C5 Flow Control and Exceptions

Java 8 Associate

1Z0-808

Link

https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387
https://docs.oracle.com/javase/specs/jls/se8/html
https://docs.oracle.com/javase/tutorial
http://www.java2s.com

https://enthuware.com
https://github.com/carlosherrerah/OCA

JUAN CARLOS HERRERA HERNANDEZ
carlos.herrera@upa.edu.mx

Contenido

>>

0. CERTIFICATION SUMMARY

The `if` statement and the `switch` statement are types of conditional/decision controls that allow your program to behave differently at a "fork in the road," depending on the result of a logical test.

The switch statement can be used to replace multiple if-else statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`); or it can evaluate `enum`s; and as of Java 7, it can evaluate `String`s.
If there is no match, then the `default` case will execute, if there is one.

three looping constructs available: `for, while, do`
The `break` and `continue` statements can be used in either a labeled or unlabeled fashion

Java provides an elegant mechanism in exception handling.
Exception handling allows you to isolate your error-correction code into separate blocks so the main code doesn't become cluttered by error-checking code.

```
try {
  throw new Exception();
} catch (Exception e) {
  System.out.println("Exception");
} finally {
  System.out.println("Finally");
}
```

```
try {
  throw new Exception();
} catch (Exception e) {
  System.out.println("Exception");
}
```

```
try {
  // No throw Exception();
} finally {
  System.out.println("Finally");
}
```

Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

To declare that an exception may be thrown, the `throws` keyword is used in a **method** definition, along with a list of all checked exceptions that might be thrown.

- **Runtime exceptions** are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they **do not need to be handled or declared**, and thus are known as **"unchecked" exceptions**.
- **Errors** are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they **do not need to be handled or declared**.
- **Checked exceptions** include any exception types that are not of type `RuntimeException` or `Error`.

>>

1. Using if and switch Statements

- The only legal expression in an `if` statement is a `boolean` expression in other words, an expression that resolves to a `boolean` or a `Boolean` reference.

- Watch out for `boolean` assignments (=) that can be mistaken for `Boolean` equality (==) tests:
```
boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```

- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.

- `switch` statements can evaluate only to `enum`s or the `byte`, `short`, `int`, `char`, and, as of Java 7, `String` data types. You can't evaluate `long`, `float`, and `double` :
```
long s = 30;
switch(s) { }
```

- The `case` constant must be a literal or a compile-time constant, including an `enum` or a `String`. You cannot have a case that includes a nonfinal variable or a range of values.

- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.

- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.

- The `default` block can be located a**nywhere** in the `switch` block, so if no preceding `case` matches, the `default` block will be entered; if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

- It's also illegal to have more than one `case` label using the same value.
- It ***is*** legal to leverage the power of boxing in a `switch` expression. `switch(new Integer(4))`

```
switch (expression) {
case constant1: code block
case constant2: code block
default: code block
}
```

```
// ¿Cual es el resultado?
```

```
if (booleanExpression) {
      System.out.println("Inside");
}
```

```
int x = 3
if (x = 3) {
   System.out.println("x is 3");
} else {
   System.out.println("x is not 3");
}
```

boolean boo = false;
if (boo = true) { }
You might think one of three things:
1. The code compiles and runs fi ne, and the if test fails because boo is false.
2. The code won't compile because you're using an assignment (=) rather than an equality test (==).
3. The code compiles and runs fi ne and the if test succeeds because boo is SET to true (rather than TESTED for true) in the if argument!

```
// Indentacion del programa
```

```
if (x > 3) {
   y = 2;
}
z += 8;
a = y + x;
```

```
if (x > 3) // bad practice
   y = 2;
z += 8;
a = y + x;
```

```
if (x > 3)
   y = 2;
   z += 8;
   a = y + x;
```

```
// Evitar anidamiento
    if (price < 300) {
      buyProduct();
    } else {
      if (price < 400) {
        getApproval();
      } else {
        dontBuyProduct();
      }
    }
```

```
    if (price < 300) {
      buyProduct();
    } else if (price < 400) {
      getApproval();
    } else {
      dontBuyProduct();
    }
```

```
¿Cuál es el comportamiento de cada uno de ellos?
```

```
if (exam.done())
  if (exam.getScore() < 0.61)
    System.out.println("again");
  else
   System.out.println("Master");
```

```
if (exam.done()) {
   if (exam.getScore() < 0.61) {
      System.out.println("again");
   } else {
    System.out.println("Master");
   }
}
```

```
if (exam.done())
   if (exam.getScore() < 0.61)
      System.out.println("again");
else
   System.out.println("Master");
```

```
¿Cual es el resultado?
```

```
int y = 5;
int x = 2;
if (((x > 3) && (y < 2)) | doStuff()) {
   System.out.println("true");
}
```

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | doStuff()) {
   System.out.println("true");
}
```

>>

2. Creating Loops Constructs

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop—in other words, code below the `for` loop won't be able to use the variable.
- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be comma separated.
- An enhanced `for` statement (new as of Java 5) has two parts: the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- Unlike with C, you cannot use a number or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will **always** enter the body of the loop at least once.

2.1. Using break and continue

- An unlabeled `break` statement will cause the current iteration of the innermost loop to stop and the line of code following the loop to run.
- An unlabeled `continue` statement will cause the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.

- If the `break` statement or the `continue` statement is labeled, it will cause a similar action to occur on the labeled loop, not the innermost loop.
- The `break` and `continue` keywords are used to stop either the entire loop (`break`) or just the current iteration (`continue`).
- The difference between them is whether or not you continue with a new iteration or jump to the first statement below the loop and continue from there.
- ***continue* statements must be inside a loop; otherwise, you'll get a compiler error. *break* statements must be used inside either a loop or a *switch* statement.**

```
int x = 1;
while (x) { }       // Won't compile; x is not a boolean
while (x = 5) { }   // Won't compile; resolves to 5
while (x == 5) { }  // Legal, equality test
while (true) { }    // Legal
```

```
for (/*Initialization*/ ; /*Condition*/ ; /* Iteration */) {
    /* loop body */
}
```

```
for (int x = 10, y = 3; y > 3; y++) { }

for( ; ; ) {
  System.out.println("Inside an endless loop");
}
```

TABLE 5-1 Causes of Early Loop Termination

Code in Loop	What Happens
`break`	Execution jumps immediately to the first statement after the `for` loop.
`return`	Execution jumps immediately back to the calling method.
`System.exit()`	All program execution stops; the VM shuts down.

You must first make sure the code isn't violating any fundamental rules that will lead to compiler error, and then look for possible exceptions. Only after you've satisfi ed those two, should you dig into the logic and fl ow of the code in the question.

The Enhanced for Loop (for Arrays)
for(declaration : expression)

```
// tabla de multiplicar de un numero
int num = 5;
for (int j = 1; j <= 10; j++) {
  if (j == 6) {
    break  /* continue */;
  }
  System.out.println(num + " * " + j + " = " + (num * j));
}
```

2.2. Labeled Statements

Both the `break` statement and the `continue` statement can be unlabeled or labeled. Although it's far more common to use `break` and `continue` unlabeled.

```
    // etiquetas
    System.out.println("Etiquetas");
    boolean isTrue = true;
    outer: for (int i = 0; i < 2; i++) {
      while (isTrue) {
        System.out.println("Hello");
        break outer;   /* continue outer; */
      } // end of inner while loop
      System.out.println("Outer loop."); // Won't print
    } // end of outer for loop
    System.out.println("Good-Bye");
```

>>

3. Handling Exceptions

- Some of the benefits of Java's exception-handling features include organized error-handling code, easy error detection, keeping exception handling code separate from other code, and the ability to reuse exception-handling code for a range of issues.

- Exceptions come in two flavors: checked and unchecked.

- Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.

- Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws` or handle the exception with an appropriate `try/catch`.

- Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.

- A `finally` block will always be invoked, regardless of whether an exception is thrown or caught in its `try/catch`.

- The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.

- Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.

- Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()` and `main()` is "ducking" the exception by declaring it).

- You can almost always create your own exceptions by extending `Exception` or one of its checked exception subtypes. Such an exception will then be considered a checked exception by the compiler. (In other words, it's rare to extend `RuntimeException`.)

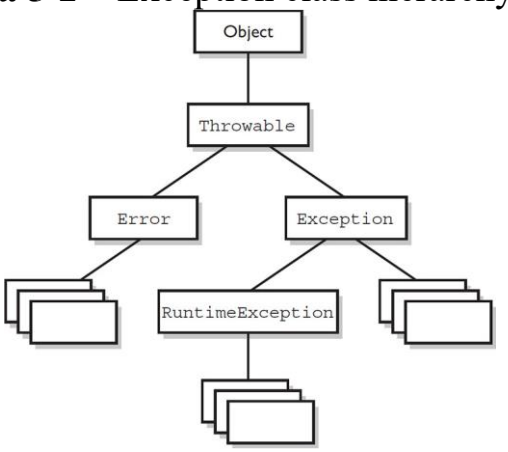
- All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes!

- Some exceptions are created by programmers and some by the JVM.

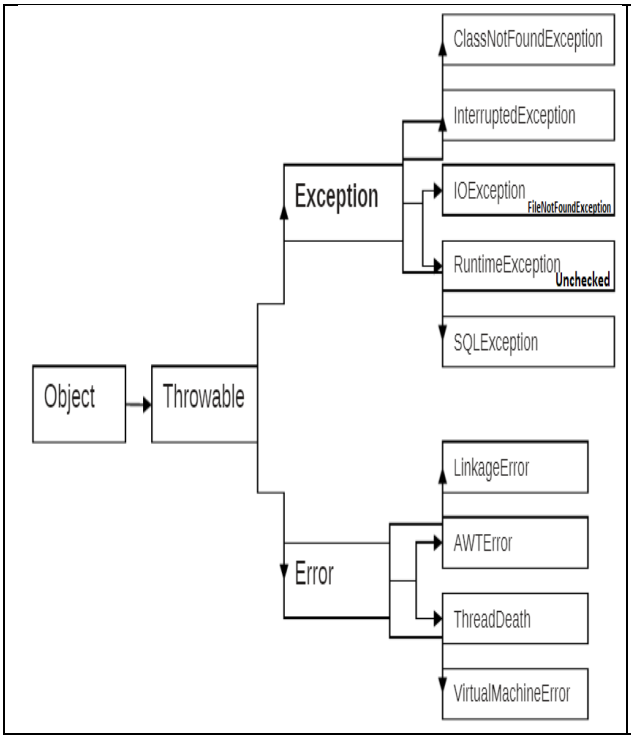
It is illegal to use a try clause without either a catch clause or a finally clause.

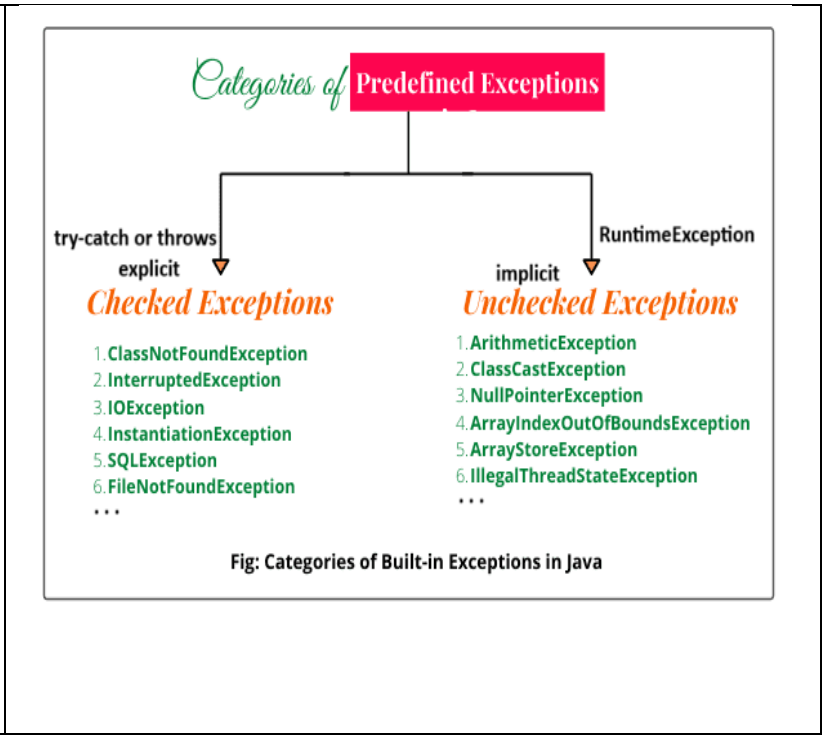
```
try {
  // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}
```

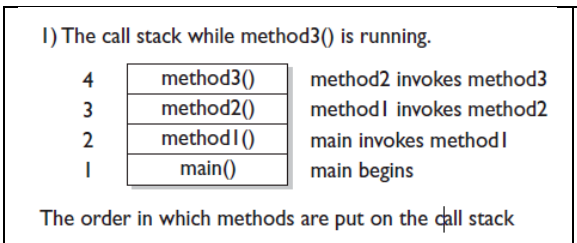
Figura 5-2 Exception class hierarchy



`printStackTrace()`








```
1  public class P3TryCatch {
2
3    static void doStuff() {
4      doMoreStuff();
5    }
6
7    static void doMoreStuff() {
8      int x = 5 / 0; // ArithmeticException is thrown
9    }
0
1    public static void main(String[] args)  {
2       doStuff();
3    }
4  }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at scjp.c5.P3TryCatch.doMoreStuff(P3TryCatch.java: 8)
    at scjp.c5.P3TryCatch.doStuff(P3TryCatch.java:4)
    at scjp.c5.P3TryCatch.main(P3TryCatch.java:12)
```

3.1. Exception Declaration and the Public Interface

Each method must either handle all checked exceptions by supplying a `catch` clause or list each unhandled checked exception as a thrown exception.

A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception somewhere, your code will not compile.

To create your own exception, you simply subclass `Exception` (or one of its subclasses) as follows:

```
class MyException extends Exception { }
```

>>

4. Common Exceptions and Errors

8.5 Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)

Let's define two broad categories of exceptions and errors:

- **JVM exceptions.** Those exceptions or errors that are either exclusively or most logically thrown by the JVM.
- **Programmatic exceptions.** Those exceptions that are thrown explicitly by application and/or API programmers.

JVM exceptions

```
package scjp.c5;

public class P5CommonException {
  static String s;

  static void go() { // recursion gone bad
    go();
  }

  public static void main(String[] args) {
    go();  // StackOverflowError
    System.out.println(s.length());//NullPointerException

    String a = "123G";
    int b  = Integer.parseInt(a);
  }
}
```

Programmatic Exceptions

```
static int parseInt(String s) throws NumberFormatException {
    boolean parseSuccess = false;
    int result = 0;
    // do complicated parsing
    if (!parseSuccess) // if the parsing failed
      throw new NumberFormatException();
    return result;
  }
```

Tabla 5- 2. Descriptions and Sources of Common Exceptions.

summarizes the ten exceptions and errors that are most likely a part of the OCA 8 exam.

Exception	Description	Typically Thrown
`ArrayIndexOutOfBoundsException` (this chapter)	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
`ClassCastException` (Chapter 2)	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
`IllegalArgumentException`	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
`IllegalStateException`	Thrown when the state of the environment doesn't match the operation being attempted—for example, using a scanner that's been closed.	Programmatically
`NullPointerException` (Chapter 3)	Thrown when attempting to invoke a method on, or access a property from, a reference variable whose current value is `null`.	By the JVM
`NumberFormatException` (this chapter)	Thrown when a method that converts a `String` to a number receives a `String` that it cannot convert.	Programmatically
`ArithmeticException`	Thrown when an illegal math operation (such as dividing by zero) is attempted.	By the JVM
`ExceptionInInitializerError` (Chapter 2)	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
`StackOverflowError` (this chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
`NoClassDefFoundError`	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

>>