Feel free to contact **santhosh_arena@yahoo.co.in**

Collective Info.

santhosh

SCJP java Programmer certification Exam And Training. This site is entirely independent of Sun Microsystems Inc, The Sun Certified Java Programmers Exam (SCJP) is the internationally recognized java certification exam. I am offering free mock Exam preparation questions, practice tests, tutorials, certification faq and sample code. This page contains a very detailed tutorial organized by topic. At the end of the tutorial you can learn from your mistakes and apply your concepts on the free mock exams java certification practice tests.

Chapter 5 : Flow Control and Exceptions

Unreachable statements produce a compile-time error.

```
while (false) { x = 3; } // won't compile

for (;false;) { x =3; } // won't compile

if (false) {x = 3; } // will compile, to provide the ability to conditionally compile the code.
```

- **Local variables already declared in an enclosing block, therefore visible in a nested block cannot be re-declared inside the nested block.**
- **A local variable in a block may be re-declared in another local block, if the blocks are disjoint.**
- **Method parameters cannot be re-declared.**

1. Loop constructs

- **3 constructs - for, while, do**
- **All loops are controlled by a boolean expression.**
- **In while and for, the test occurs at the top, so if the test fails at the first time, body of the loop might not be executed at all.**
- **In do, test occurs at the bottom, so the body is executed at least once.**
- **In for, we can declare multiple variables in the first part of the loop separated by commas, also we can have multiple statements in the third part separated by commas.**
- **In the first section of for statement, we can have a list of declaration statements or a list of expression statements, but not both. We cannot mix them.**
- **All expressions in the third section of for statement will always execute, even if the first expression makes the loop condition false. There is no short -circuit here.**

2. Selection Statements

- if takes a boolean arguments. Parenthesis required. else part is optional. else if structure provides multiple selective branching.
- switch takes an argument of byte, short, char or int.(assignment compatible to int)
- case value should be a constant expression that can be evaluated at compile time.
- Compiler checks each case value against the range of the switch expression's data type. The following code won't compile.

```
byte b;
switch (b) {
  case 200: // 200 not in range of byte
  default:
}
```

- We need to place a break statement in each case block to prevent the execution to fall through other case blocks. But this is not a part of switch statement and not enforced by the compiler.
- We can have multiple case statements execute the same code. Just list them one by one.
- default case can be placed anywhere. It'll be executed *only if* none of the case values match.
- switch can be nested. Nested case labels are independent, don't clash with outer case labels.
- Empty switch construct is a valid construct. But any statement within the switch block should come under a case label or the default case label.

3. Branching statements

- break statement can be used with any kind of loop or a switch statement or just a labeled block.
- continue statement can be used with only a loop (any kind of loop).
- Loops can have labels. We can use break and continue statements to branch out of multiple levels of nested loops using labels.
- Names of the labels follow the same rules as the name of the variables.(Identifiers)
- Labels can have the same name, as long as they don't enclose one another.
- There is no restriction against using the same identifier as a label and as the name of a package, class, interface, method, field, parameter, or local variable.

4. Exception Handling

- An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- There are 3 main advantages for exceptions:

  1. Separates error handling code from "regular" code

  2. Propagating errors up the call stack (without tedious programming)

  3. Grouping error types and error differentiation

- An exception causes a jump to the end of try block. If the exception occurred in a method called from a try block, the called method is abandoned.

- If there's a catch block for the occurred exception or a parent class of the exception, the exception is now considered handled.

- At least one 'catch' block or one 'finally' block must accompany a 'try' statement. If all 3 blocks are present, the order is important. (try/catch/finally)

- finally and catch can come only with try, they cannot appear on their own.

- Regardless of whether or not an exception occurred or whether or not it was handled, if there is a finally block, it'll be executed always. (Even if there is a return statement in try block).

- System.exit() and error conditions are the only exceptions where finally block is not executed.

- If there was no exception or the exception was handled, execution continues at the statement after the try/catch/finally blocks.

- If the exception is not handled, the process repeats looking for next enclosing try block up the call hierarchy. If this search reaches the top level of the hierarchy (the point at which the thread was created), then the thread is killed and message stack trace is dumped to System.err.

- Use throw new xxxException() to throw an exception. If the thrown object is null, a NullPointerException will be thrown at the handler.

- If an exception handler re-throws an exception (throw in a catch block), same rules apply. Either you need to have a try/catch within the catch or specify the entire method as throwing the exception that's being re-thrown in the catch block. Catch blocks at the same level will not handle the exceptions thrown in a catch block - it needs its own handlers.

- The method fillInStackTrace() in Throwable class throws a Throwable object. It will be useful when re-throwing an exception or error.

- The Java language requires that methods either *catch* or *specify* all checked exceptions that can be thrown within the scope of that method.

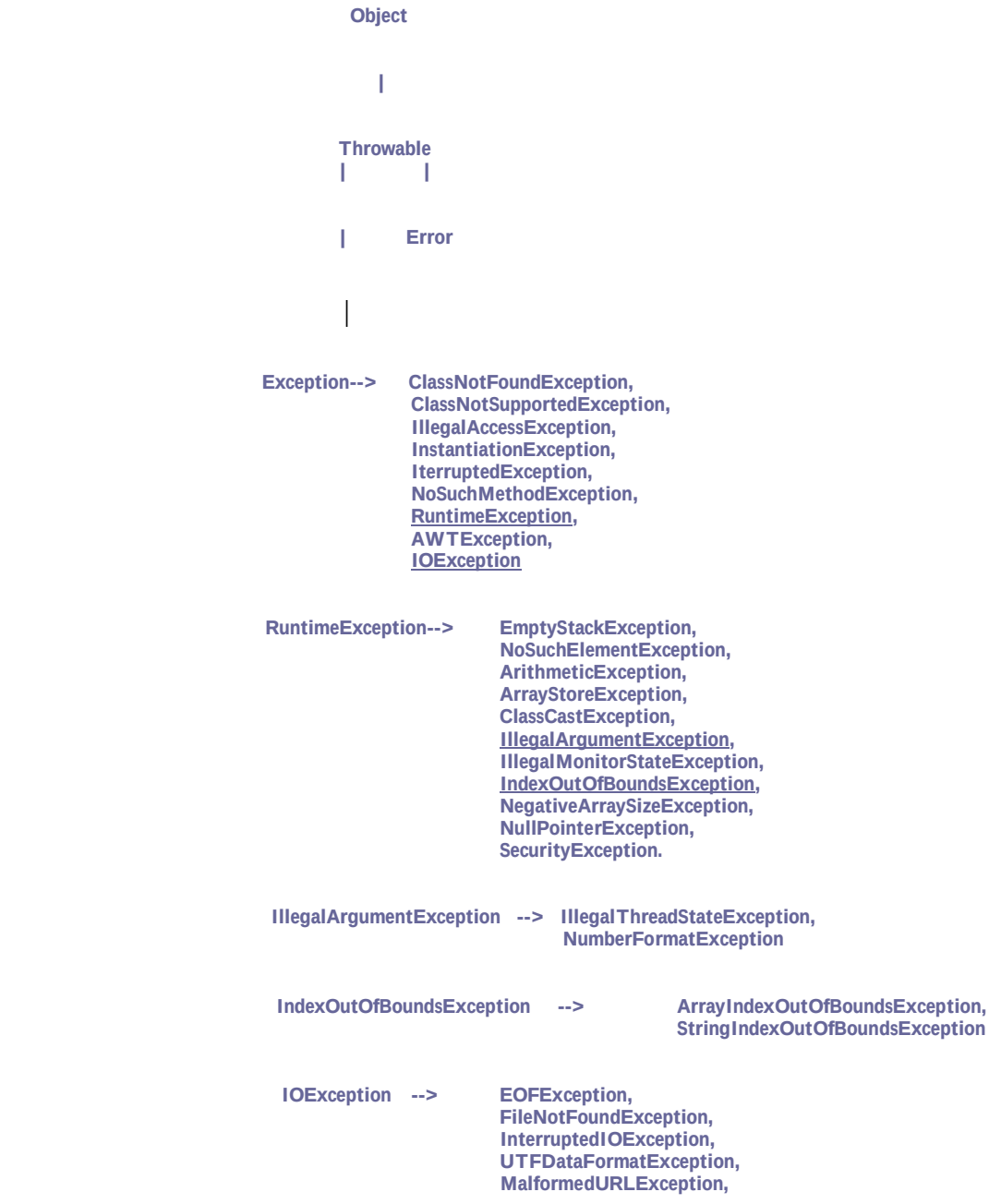
- All objects of type java.lang.Exception are checked exceptions. (Except the classes under java.lang.RuntimeException) If any method that contains lines of code that might throw checked exceptions, compiler checks whether you've handled the exceptions or you've declared the methods as throwing the exceptions. Hence the name checked exceptions.

- If there's no code in try block that may throw exceptions specified in the catch blocks, compiler will produce an error. (This is not the case for super-class Exception)

- Java.lang.RuntimeException and java.lang.Error need not be handled or declared.

- An overriding method may not throw a checked exception unless the overridden method also throws that exception or a super-class of that exception. In other words, an overriding method may not throw checked exceptions that are not thrown by the overridden method. If we allow the overriding methods in sub-classes to throw more general exceptions than the overridden method in the parent class, then the compiler has no way of checking the exceptions the sub-class might throw. (If we declared a parent class variable and at runtime it refers to sub-class object) This violates the concept of checked exceptions and the sub-classes would be able to by-pass the enforced checks done by the compiler for checked exceptions. This should not be allowed.

Here is the exception hierarchy.



ProtocolException,
SockException,
UnknownHostException,
UnknownServiceException.