Contents

Throwable

This is the top level class which is used in pretty much all exception handling and error handling in the Java language, used by the `JVM` itself, and all exceptions - checked and unchecked extend off of it. Further more the special type of Error, which is mostly used by the `JVM`, internally, is also extending off of it.

Every thrown object from java Code must be a subclass of the `Throwable` class, or one of its sub-classes, exception handling constructs in the language such as throw statements, throws clause, and the catch clause, deal only with `Throwable` and its sub-classes. There are three important sub-classes of `Throwable` - `Error`, `Exception` and `RuntimeException`.

The general hierarchy of the java exception model is such that: `Object -> Throwable -> Excetpion -> RuntimeException`

- `Exceptions` of type Exception in Java are known as check exceptions. If code can throw an Exception you must handle it using a catch block or declare that the method throws that exception forcing the caller of that method to handle that exception
- `RuntimeException` is a derived class of the Exception class. The exceptions deriving from `RuntimeException` are known as unchecked exceptions. It is optional to handle unchecked exceptions. If a code segment in a method can throw an unchecked exception, it is not mandatory to catch that exception or declare that exception in the throws clause of that method.
- `Error` - when the `JVM` detects a serious abnormal condition in the program, it raises an exception of type Error. Exception of type Error indicate an abnormal condition, in the program. There is no point in catching these exceptions and pretend nothing has happened. At this point if an Error exception occurs it is a really bad practice to do so! These errors signal that some irrecoverable state has been reached.

Throwing

```
public static void main(String []args) {
    if(args.length == 0) {
        // this branch of the code declares that a RuntimeException
           exception is thrown, as mentioned above these types
```

```
        // of exceptions are not mandatory to be handled, and the program
           will simply exit and the call stack will unwind
        // accordingly.
        throw new IllegalArgumentException("No input passed to echo
           command");
    }
    else {
        for(String str : args) {
            // command-line arguments are separated and passed as an array
            // print them by adding a space between the array elements
            System.out.print(str + " ");
        }
    }
```

Since there is no explicit user defined handler for the exception thrown above, the `JVM` will itself make sure that this exception is caught at the moment the program terminates, the reason being, to at the very least log out the error, so the user can obtain some sort of information about the invalid state that has occurred.

```
public static void main(String [] args) {
    System.out.println("Type an integer in the console: ");
    Scanner consoleScanner = new Scanner(System.in);
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
}
```

Here is another example with an unchecked exception which is possible to occur in user code, the method `nextInt` which tries to read the next integer from `stdin`, can throw an exception if the input is not a valid integral type, this means that the method can throw an unchecked exception in that case. However this is also documented in the method documentation, therefore the user code has the ability to, if desired, catch the exception, even though it is unchecked and handle that case gracefully, instead of completely killing the program.

Another useful feature in the try-catch block is that an exception can be wrapped in another one in the catch block and re-thrown, instead of handling it immediately in the catch block this is usually useful in practice, since swallowing an exception (leaving the catch block empty, or adding some generic print statement which simply loses information about the cause and reason for the exception is a bad idea). Wrapping an exception is a good way to preserve the initial cause, if it can not be handled at the time of the catch statement. And can be used to be wrapped in a more generic exception which other layers in the program execution know how to handle.

Chain throwing exceptions is most usually done to do what is called exception translation - translating internal system exceptions coming from libraries or the java ones, to a more generic business level exception which is easier to handle and control.

Catching

Besides creating exceptions the user code can also catch them, this is the companion action to throwing exceptions. The example below extends the example from above, which reads integer from `stdin`, and tries to correctly handle the case when the integer is not an integer, or it is unable to parse a valid integer from `stdin`. In this case the program will simply just print some message on `stdout`, but in a more robust solution, it can be put in a `while(true)` loop and require a user entry until a valid integral type is read from `stdin`.

```
public static void main(String [] args) {
```

```
    System.out.println("Type an integer in the console: ");
    Scanner consoleScanner = new Scanner(System.in);
    try {
        System.out.println("You typed the integer value: " +
           consoleScanner.nextInt());
    } catch(InputMismatchException ime) {
        // nextInt() throws InputMismatchException in case anything other
           than an integer is typed in the console; so
        // handle it, this is crude example since the exception is
           swallowed, and some very generic error is printed out
        // in practice this is not a good idea, since information is being
           lost, namely the information from the
        // exception, or that an exception is occurred
        System.out.println("Error: You typed some text that is not an
           integer value...");
    }
}
```

The try-catch block has two elements to it, the code in the try block will be the one which will be examined for exceptions, once an exception occurs, the code in the catch block will be triggered, and if the exception in the catch block matches the catch exception statements, then whichever matches will be invoked. Note that multiple catch statements are allowed, it is also possible that no exception is caught, if none of the catch statements match the exception being thrown. If in the example above, something else than `InputMismatchException` then the catch block will not be triggered, and the exception will be handled by the `JVM` itself

Multi-catch catching

```
public static void main(String [] args) {
    String integerStr = "non-integral-type";
    System.out.println("The string to scan integer from it is: " +
       integerStr);
    Scanner consoleScanner = new Scanner(integerStr);
    try {
        System.out.println("The integer value scanned from string is: " +
            consoleScanner.nextInt());
    } catch(InputMismatchException ime) {
        System.out.println("Error: Cannot scan an integer from the given
           string");
    } catch(NoSuchElementException nsee) {
        System.out.println("Error: Cannot scan an integer from the given
           string");
    } catch(IllegalStateException ise) {
        System.out.println("Error: nextInt() called on a closed Scanner
           object");
    }
}
```

In the example above, calling `nextInt`, will fail with `NoSuchElementException`, why is that ? Simple, because the scanner is initialized with an invalid string from which to read/parse int, in this case a string `"non-integral-type"`. This also shows that the order in which the exceptions are caught also matters, in this case the `InputMismatchException` will not trigger before the `NoSuchElementException`, however

`InputMismatchException` extends from `NoSuchElementException`, therefore it is not possible two do the following

```
// note that the catch statement order matters, especially when the
   exceptions are such that they have intersecting and
// common hierarchy, in this case the NoSuchElementException is parent
   class of the InputMismatchException
try {
    System.out.println("The integer value scanned from string is: "
        + consoleScanner.nextInt());
} catch(NoSuchElementException nsee) {
    System.out.println("Error: Cannot scan an integer from the given
       string");
} catch(InputMismatchException ime) {
    System.out.println("Error: Cannot scan an integer from the given
       string");
}
```

Note that the `NoSuchElementException` which is a super class of the `InputMismatchException` is caught first, that is not possible, the child / sub classes must come first, otherwise this is considered a compile time error, why is that ? If that was not an error during run-time the exception that will be caught will always be `NoSuchElementException`, even if the actual instance is `InputMismatchException`, meaning that this is potentially a logical error, and there is no way to catch that during run-time, that is why the compiler is helpful, and prompts with an error during compilation, suggesting something is logically wrong, not syntactically.

`When providing multiple catch handlers, handle specific exceptions before handling general exceptions. If you provide a derived class exception, catch handler after a base class exception handler, the compiler will issue a compile time error.`

The language also provides something known as multi catch blocks, which allows one to express the catch statement such that it matches multiple exception in a single catch statement.

```
// note that a binary or operator (pipe) is used, however, this is
   technically, a compiler error because in multi-catch
// block the exceptions can not be of related types, in this case one is a
   parent of the other
try {
    System.out.println("The integer value scanned from string is: " +
        consoleScanner.nextInt());
} catch(NoSuchElementException | IllegalStateException multie) {
    // this is not a valid multi-catch, since the two exceptions are
       related,
    System.out.println("Error: An error occured while attempting to scan
       the integer");
}
```

`In multi-catch block, one cannot combine catch handlers for two exceptions that share a base - derived class relationship. Only unrelated exceptions can be listed and combined in a multi-catch.`

One might notice that both multiple catch statements and multi-catch statements fill a very similar role, which one makes more sense, depends on the situation. If the exceptions are thrown for the same or similar reason, then a multi-catch makes more sense, makes code more readable and maintainable. However if the different

exceptions are handled very differently in their catch blocks, then a multiple catch statements approach is required

General catching

As already discussed all exceptions extend from the `Throwable` class, and further more from the Exception class as well, that means that in the catch statement one can specify a higher level Exception type, `Throwable` even, to make sure all types of exceptions which a given statement can emit within a try-catch block are caught, not the best idea, but sometimes it can be useful. For example many types of different exceptions are being thrown when one interacts with the I/O API operations. If one tries to handle each and every case, trying to be as pedantic as possible, the code will quickly become unreadable and non really maintainable, in in such general situations, it is simply better to catch `Exception` type instead, for the cases that are not as important, and a general error can be reported.

```
try {
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
} catch(InputMismatchException ime) {
    // if something other than integer is typed, we'll get this exception,
       so handle it
    System.out.println("Error: You typed some text that is not an integer
       value...");
} catch(Exception e) {
    // catch IllegalStateException, and anything else here which is
       unlikely to occur...
    System.out.println("Error: Encountered an exception and could not read
       an integer from
        the console... ");
}
```

So in the example above the most important exception is being handled, exclusively, which is the `InputMismatchException`, which will occur in case the scanner can not read a valid integral type on the input, however everything else, is relegated to just the `catch(Exception)` block

Finalizing

All of the examples above have one big issue, that is the fact that the Scanner object is not closed, however this can be mitigated by calling the close method on the scanner object, this can be done in several ways, but the try-catch block in java have two main methods to achieve this - try-with-resources (introduced in Java 8) and finally (the classic approach)

Finally

The finally block is a block which is executed, regardless of the fact that an exception is occurred. The code inside the finally block will be executed AFTER the code in both the try and catch, even if an exception is not caught, the finally block will always be executed. This provides a deferred safe way to clean up resources after their usage has expired

```
System.out.println("Type an integer in the console: ");
Scanner consoleScanner = new Scanner(System.in);
try {
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
```

```
} catch(Exception e) {
    // call all other exceptions here ...
    System.out.println("Error: Encountered an exception and could not read
       an
        integer from the console... ");
    System.out.println("Exiting the program - restart and try the program
       again!");
} finally {
    // note that close itself, as documented can throw an
       IllegalStateException, in case one tries to call close
    // multiple times on the same scanner object
    System.out.println("Done reading the integer... closing the Scanner");
    consoleScanner.close();

    // if the call to close above, throws, anything below it will not be
       executed, that is why it is not recommended to
    // use the old fashioned finally block to close, closeable objects,
       since that could still cause a resource leak in
    // some cases
    System.out.println("Will not be called in caes close() above throws an
       exception");
}
```

It is also possible to have internal try-catch inside a finally block, that is actually the old school way of closing file stream objects, since their close method, does in fact throw a checked exception and it has to be handled, it is not optional. Note that however

`If a call to System.exit()is done inside a method, it will abnormally terminate the program. So if the calling method has any finally blocks, they will not be called, and resources may leak. For this reason it is a bad practice to call System.exit(), to terminate or exit from a program`

Some very weird caveats that can occur within the usage of finally block

```
// since the finally block is always executed, this will `always` return
   false, therefore it is not really advised to
// return anything from finally blocks
boolean returnTest() {
    try {
        return true;
    }
    finally {
        return false;
    }
}
```

Try-with-resources

This is a new feature in Java 8, which is meant to circumvent all the usual boilerplate and possible issues that might occur using the usual finally syntax, the try-with-resources used to close all resources which extend from the `AutoCloseable` interface, also introduced in Java 8, this interface signals to the `JVM`, that a resource is `auto-closeable`, and the resource is declared within the try-with-resources statement block the close method of that resource will be automatically closed.

The syntax of the try-with-resources is quite simple, the resources are declared and initialized in the try block, multiple ones can be declared by using a semi colon to split them up.

```
try(Scanner consoleScanner = new Scanner(System.in)) {
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
} catch(Exception e) {
    // catch all other exceptions here ...
    System.out.println("Error: Encountered an exception and could not read
       an
        integer from the console... ");
    System.out.println("Exiting the program - restart and try the program
       again!");
}
```

Note the way the resource is being declared, in the example above there is no finally block, however internally the compiler will translate this into an actual try-catch-finally block when during the code generation phase, the try-with-resources is mostly a syntax sugar for a regular finally block, this however makes sure that the code is functionally correct, and there is no logical issues.

To declare more than one resources within the try-with-resource statement as mentioned above one can use a ; to separate them, all of them are explicitly defined as effectively final

```
// note that the two declarations are split with a semi-colon, in the
   try-with-resources statement
try (ZipOutputStream zipFile = new ZipOutputStream(new
   FileOutputStream(zipFileName));
                                FileInputStream fileIn = new
                                   FileInputStream(fileName)) {
    // putNextEntry can throw IOException
    zipFile.putNextEntry(new ZipEntry(fileName));
    // the variable to keep track of number of bytes sucessfully read
    // copy the contents of the input file into the zip file
    int lenRead = 0;
    while((lenRead = fileIn.read(buffer)) > 0) { // read can throw
       IOException
        zipFile.write(buffer, 0, lenRead); // write can throw IOException
    }
    // the streams will be closed automatically because they are within
       try-with-
    // resources statement
}
```

Note that all resources declared within the try-with-resources block are effectively final, meaning that they can not be re-assigned, this is because the original reference will be lost, and the resources would effectively leak. However what is possible is the following

```
// note that in this example  the console scanner is declared and
   initialized outside the try-with-resources block,
// however another variable captures the original reference in the
   try-with-resources block effectively doing the same as
// the example above, but it does show that something like that is
   possible and not considered an error.
```

```
Scanner consoleScanner = new Scanner(System.in);
try (Scanner scan = consoleScanner) {
    // use the scan instance
} catch (Exception e) {
    // do some catching
}
```

The main benefit of the try-with-resources statement is that it simplifies the code and makes it less verbose, however it also ensures correctness, by not having to manually provide finally blocks which can be either forgotten or functionally or logically wrong.

Note that it is also possible to provide a try-with-resources block that has no catch statement, this is not really recommended however it is allowed in the language

```
try(Scanner consoleScanner = new Scanner(System.in)) {
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
}
```

```
If a finally and try-with-resources are used together, the resources inside the try-with-resour
statement will be closed BEFORE the finally block is called, meaning that closing them
again in the finally block is most definitely wrong and undefined behavior, they will
likely throw since the close method is already closed, in that case the finally block can
be used to log or notify the outside world that all resources are cleaned up correctly
and finalized.
```

Throws

The other component of the Exception framework model in Java, is the throws clause, this clause defines that a certain method throws an exception, if a method throws an unchecked exception using the throws clause is optional, however if a method is throwing a checked exception the throws clause is not optional, and it has to declare all the checked exceptions that this method might throw. It is usually a good practice to also document which exception is thrown from a method, based on which condition or erroneous state.

```
// this demonstrates how a checked exception is declared with the throws
   clause, the exception is propagated from the
// new File() method, which itself declares throws clause, in this case
   the exception is not handled in the main method,
// therefore it is propagated outisde of the main method, however it has
   to be declared that the main method throws,
// otherwise a compiler error will occur
public static void main(String []args) throws FileNotFoundException {
    System.out.println("Reading an integer from the file 'integer.txt': ");
    Scanner consoleScanner = new Scanner(new File("integer.txt"));
    System.out.println("You typed the integer value: " +
       consoleScanner.nextInt());
}
```

‘When a method which declares at least one checked exception is used through the throws clause, within an outer method, the outer method has only two possibilities - either handle the exception or simply re-declare the throws clause so another method which invokes it will handle it or re-throw it.

The throws clause has some additional properties which need to be considered, one of which is what happens if a base method declares throws clause, and a method in a sub-class has to override it. The rule of thumb,

and generally the accepted approach is that the contract from the base class HAS to be adhered to, it is the contract after all, so if a sub-class overrides a method, it can NOT change the throws clause such that more general checked exceptions are thrown from the overridden method, and it can not add more checked exceptions to be thrown form the overridden method, both of these actions result in compiler error (however a checked exception can be changed from the base method, only if the checked exception defined in the overriding method is part of the hierarchy, or in other words is a sub-type of the one defined in the base method)

```
interface IntReader {
    int readIntFromFile() throws IOException;
}
class InvalidThrowsClause implements IntReader {
    // this is allowed, changing the type of the checked exception with
       another checked exception, only if it is
    // part of the type hierarchy is possible, in this case
       FileNotFoundException extends from IOException
    public int readIntFromFile() throws FileNotFoundException {
        Scanner consoleScanner = new Scanner(new File("integer.txt"));
        return consoleScanner.nextInt();
    }
}
```

`The throws declaration in an overridden method can only be changed or modified to add unchecked exception to the list of exceptions being thrown, any modification to the list of the checked exception where the checked exception is not a sub-type of the original one being thrown will result in a compiler error`

Documentation

It is as mentioned already a good practice to use the @throws `JavaDoc` tag, to document the specific situations or causes in which an exception - checked or unchecked exception might be thrown from a method. Here is an actual example from the actual implementation of the `nextInt` method form Scanner

```
/**
* Scans the next token of the input as an int.
*
* <p> An invocation of this method of the form
* nextInt() behaves in exactly the same way as the
* invocation nextInt(radix), where radix
* is the default radix of this scanner.
*
* @return the int scanned from the input
* @throws InputMismatchException
* if the next token does not match the Integer
* regular expression, or is out of range
* @throws NoSuchElementException if input is exhausted
* @throws IllegalStateException if this scanner is closed
*/
public int nextInt() {
    return nextInt(defaultRadix);
}
```

The way the exceptions are listed, by convention is in alphabetical order, when a method can throw more than one exception in the documentation. They are not listed by severity or in base - child class relationship

Miscellaneous

- If a method does not have a throws clause, ti does not mean it cannot throw any exceptions; it just means it cannot throw any checked exceptions.
- Static block initializers cannot throw any check exceptions, this is because static initialization blocks are invoked when the class is loaded, so there is no way to handle the thrown exceptions in the caller. Further more there is no way to declare the checked exceptions in a throws clause using static initializer
- Non-static initializer blocks can throw checked exceptions however all the constructors should declare those exceptions, in their throws clause. This is because the compiler merges the code in the non-static initalizer block and the constructors during the code generation phase, hence the throws clause of the constructor can be used for declaring the checked exceptions that a non-static init block can throw.
- An overriding method cannot declare more checked exceptions in the throws clause than the list of exceptions declared in the throws clause of the base method
- An overriding method can declare more specific exceptions than the exception listed in the throws clause of the base method; in other words, one can still declare derived exception in the throws clause of the overriding method.
- If a method is declared in two or more interfaces, and if that method declares to throw different exceptions in the throws clause, the method implementations must declare and list all of these exception in their throws clause, the methods throws clauses are effectively merged together

User-exceptions

Custom exceptions can be declared by users by extending either from the `Exception` class or the `RuntimeException`, it is bad practice to extend from the `Throwable` or `Error` classes. As already mentioned, based on the type of exception - checked or unchecked, one extends from `Exception` or `RuntimeException` respectively

The Exception class and by proxy the `RuntimeException` provide the following constructors

Method	Description
Exception()	Default constructor of the Exception class with no additional (or detailed) information on the exception.
Exception(String)	Constructor that takes a detailed information string about the constructor as an argument.
Exception(String, Throwable)	In addition to a detailed information string as an argument, this exception constructor takes the cause of the exception (which is another exception) as an argument.
Exception(Throwable)	Constructor that takes the cause of the exception as an argument.

Method	Description
String getMessage()	Returns the detailed message (passed as a string when the exception was created).
Throwable getCause()	Returns the cause of the exception (if any, or else returns null).
Throwable[] getSuppressed()	Returns the list of suppressed exceptions (typically caused when using a try-with-resources statement) as an array.

Method	Description
void printStackTrace()	Prints the stack trace (i.e., the list of method calls with relevant line numbers) to the console (standard error stream). If the cause of an exception (which is another exception object) is available in the exception, then that information will also be printed. Further, if there are any suppressed exceptions, they are also printed.

```
// the class declaration is an example of a custom user exception, which
   is in this case unchecked one, since it extends
// from the RuntimeException, by default the custom user exceptions are
   not required to provide any body, since the parent
// class RuntimeException and Exception in this case have default non-arg
   constructors
class InvalidInputException extends RuntimeException {

    public InvalidInputException() {
    }

    public InvalidInputException(String message) {
        super(message);
    }

    public InvalidInputException(String message, Throwable throwable) {
        super(message, throwable);
    }
}
```

Assertions

The assert statement is used to check or test your assumptions about the program. The keyword, assert provides support for assertions in Java. Each assertion statement contains a Boolean expression. If the result of the Boolean expression is true, it means the assumption is true, so nothing happens. However if the Boolean result is false, then the assumption you had about the program holds no more, and the `AssertionError` is thrown. Remember that the Error class and its derived classes indicate serious `runtime` errors and are not meant to be handled. In the same way, if an `AssertionError` is thrown the best course of action is not to catch the exception and to allow the program to terminate. After that the assertion has to be examined, for the reason it failed.

Asserts are quite useful tool for ones program, it allows users to make certain assumptions in the code, and that is quite helpful to discover when these assumptions fail.

A very important detail to remember is that assertions are by default disabled in the run-time. To enable them use the `-ea` switch (or its longer form of `-enableasserts`). To disable assertions at `runtime` use a `-da` switch. If assertions are disabled by default at runtime then what is the use of `-da` switch ? There are many uses. For example, if you want to enable assertions for all classes within a given package and want to disable asserts in a specific class in that package. Then a -da switch is useful.

Command-Line Argument	Short Description
-ea	Enables assertions by default (except system classes).
-ea:	Enables assertions for the given class name.

Command-Line Argument	Short Description
-ea:...	Enables assertions in all the members of the given package .
-ea:...	Enable assertions in the given unnamed package.
-esa	Short for -enablesystemsassertions; enables assertions in system classes. This option is rarely used.
-da	Disable assertions by default (except system classes).
-da:	Disable assertions for the given class name.
-da:...	Disables assertions in all the members of the given package .
-da:...	Disable assertions in the given unnamed package.
-dsa	Short for -disablesystemsassertions; disables assertions in system classes. This option is rarely used.

Summary

Try-catch and throw statements

- When an exception is thrown from a try block, the `JVM` looks for a matching catch handler from the list of catch handlers in the method call-chain. If no matching handler is found, that unhandled exception will result in crashing the application.
- While providing multiple exception handlers (stacked catch handlers), specific exception handlers should be provided before general exception handlers.
- You can programatically access the stack trace using the methods such as `printStackTrace()` and `getStackTrace()`, which can be called on any exception object.

Catch, multi-catch, and finally

- A try block can have multiple catch handlers. If the cause of two or more exceptions is similar, and the handling code is also similar, you can consider combining the handlers and make it into a multi-catch block.
- A catch block should either handle the exception or `rethrow` it. To hide or swallow an exception by catching an exception and doing nothing is really a bad practice.
- You can wrap one exception and throw it as another exception. These two exceptions become chained exceptions. From the thrown exception, you can get the cause of the exception.
- The code inside a finally block will be executed irrespective of whether a try block has successfully executed or resulted in an exception.

Try-with-resources statement

- Forgetting to release resources by explicitly calling the `close()` method is a common mistake. You can use a try-with-resources statement to simplify your code and auto-close resources.
- You can auto-close multiple resources within a try-with-resources statement. These resources need to be separated by semicolons in the try-with-resources statement header.
- If a try block throws an exception, and a finally block also throws exception(s), then the exceptions thrown in the finally block will be added as suppressed exceptions to the exception that gets thrown out of the try block to the caller.

Custom exceptions

- It is recommended that you derive custom exceptions from either the Exception or `RuntimeException` class.
- A method's throws clause is part of the contract that its overriding methods in derived classes should obey.
- An overriding method can provide the same throw clause as the base method's throws clause or a more specific throws clause than the base method's throws clause.
- The overriding method cannot provide a more general throws clause or declare to throw additional checked exceptions when compared to the base method's throws clause.
- For a resource to be usable in a try-with-resources statement, the class of that resource must implement the `java.lang.AutoCloseable` interface and define the `close()` method.

Invariant with asserts

- Assertions are condition checks in the program and should be used for explicitly checking the assumptions you make while writing programs.
- The assert statement is of two forms: one that takes a Boolean argument and one that takes an additional string argument.
- If the Boolean condition given in the assert argument fails (i.e., evaluates to false), the program will terminate after throwing an `AssertionError`. It is not advisable to catch and recover from when an `AssertionError` is thrown by the program.
- By default, assertions are disabled at runtime. You can use the command-line arguments of -ea (for enabling asserts) and `-da` (for disabling asserts) and their variants when you invoke the `JVM`.