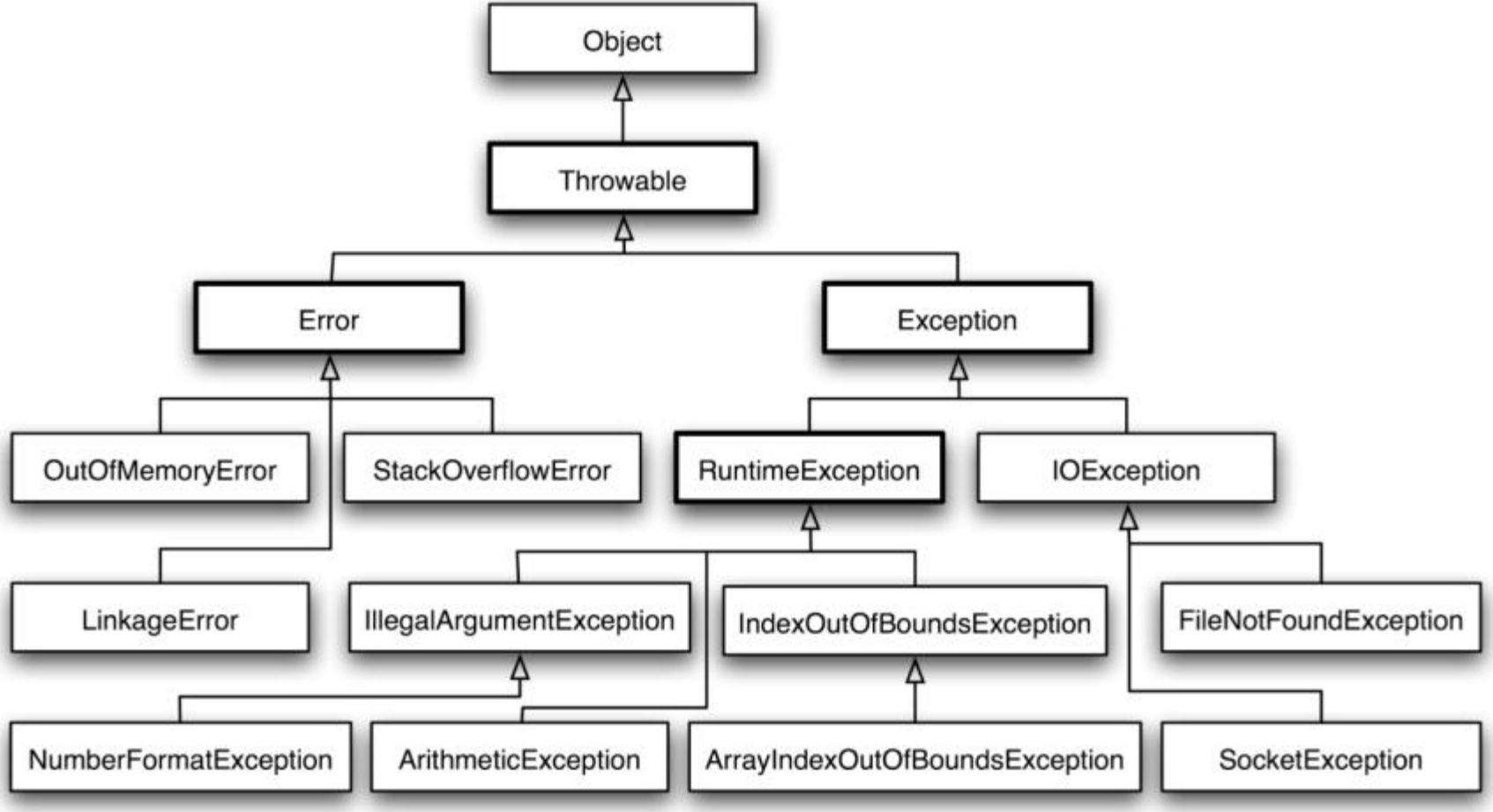
دستور کار ۸

Exceptions

Exceptions and outputs have distinct purposes in programming. Exceptions are used to handle and communicate unexpected situations or errors, allowing for centralized error handling and maintaining program integrity. Outputs, on the other hand, represent the expected results or values produced by a program. Exceptions are suitable for exceptional cases, while outputs convey regular results. Both have their roles in different scenarios and should be used accordingly.


Object
Throwable
Error
Exception
OutOfMemoryError
StackOverflowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException

Exception handeling

There are two approaches to exception handling:

- Try-Catch
- Using "throws" in the method signature.

RAM vs HDD

- RAM refers to the temporary memory in a computer that is used for actively running programs and storing data that is actively being accessed. It is a type of volatile memory, meaning its contents are lost when the computer is powered off
- On the other hand, HDD refers to a non-volatile storage device used for long-term data storage. It consists of rotating magnetic disks that store data using read/write heads. HDDs offer larger storage capacities at a lower cost compared to other storage options.

RAM vs HDD

As you know, in Java, we use variables and objects to store data in memory. The memory is divided into two main types: primary memory and secondary memory. Primary memory is fast but limited in size, while secondary memory, such as a hard disk drive (HDD), has larger storage capacity but is slower.

When we close programs or turn off the computer, the data stored in primary memory will be erased, and we won't be able to access them. However, there are times when we need to store data for a longer period. In such cases, we can save our data as files on secondary memory, like an HDD. In Java, this can be achieved using the java.io package. The package provides the necessary classes for this purpose

Java I/O

In Java, the concept of I/O (Input/Output) is implemented through a concept called streams. A stream is a flow of data that is created between a source and a destination, facilitating the transfer of data.
When a Java program is executed, three streams are automatically created that you are familiar with:

- System.in: The input stream
- System.out: The output stream
- System.err: The error stream

Generally, there are two types of streams: byte-based streams, which handle input and output in a binary format, and character-based streams, which work with sequences of characters.

System.in System.out System.err

In Java, `System.in`, `System.out`, and `System.err` are predefined objects of the `java.lang.System` class that represent the standard input, standard output, and standard error streams, respectively.

1. `System.in`: This is the standard input stream, which is typically connected to the keyboard. It allows you to read input from the user. You can use methods like `Scanner` or `BufferedReader` to read data from this stream.
2. `System.out`: This is the standard output stream, which is typically connected to the console or terminal. It allows you to display output or results to the user. You can use the `System.out.println()` method to print text or values to this stream.
3. `System.err`: This is the standard error stream, also connected to the console or terminal. It is used to display error messages or exceptional events. By default, error messages are printed in red or with some distinguishing format to differentiate them from regular output.

Byte-based streams vs Char-based streams

- Byte-based streams, also known as binary streams, operate at the lowest level and deal with data in its raw binary form. They read and write data in the form of individual bytes. Examples of byte-based streams include InputStream and OutputStream. These streams are suitable for handling non-textual data, such as images, audio, video, and binary files.
- On the other hand, character-based streams are designed to handle textual data. They provide higher-level abstractions and work with characters rather than individual bytes. These streams automatically handle the conversion between characters and bytes using character encoding schemes, such as UTF-8 or UTF-16. Examples of character-based streams include Reader and Writer classes. They are suitable for reading and writing text files, processing strings, and working with textual data.

Byte-based streams vs Char-based streams

Note that in Java, we have two types of streams: input streams and output streams, and we cannot perform both reading and writing operations with a single stream. If we need to read from and write to a file simultaneously, we must create separate input and output streams for that file, where each stream represents a different file. FileWriter serves as the output stream, and FileReader serves as the input stream.

- `FileReader` and `FileWriter` as Char-based streams
- `ObjectInputStream` and `ObjectOutputStream` as Byte-based streams

FileReader and FileWriter

- if the file does not exist, the FileWriter will build the file.
- by default the contents of the existing file will be deleted, use append mode in order to retain the existing file contents. `FileWriter writer = new FileWriter(fileName, true);`
- To read a character from a file in Java, you can use the read() method of FileReader. However, you should note that the return type of this method is int. If the returned value is equal to -1, it means we have reached the end of the file and the reading process is complete. Otherwise, you can convert the returned value to a char to display the read character.

Note: close FileReader and FileWriter after you're done.

Serialization in java

In Java, it is possible to write an object to a file or read from a file in its entirety. To accomplish this, the desired object needs to be written as a sequence of bytes, including both the object's data and information about its class. This process is referred to as object serialization in Java. When writing an object to a file, it needs to be serialized, and when reading it, it needs to be deserialized.

Serializable

- This is an interface that does not have any methods to implement and is used only for marking classes that are intended to be stored in a file. A class that we want to store its objects in a file must implement this interface; otherwise, it will not be possible to store the objects. Note that in addition to the object itself, all its fields that need to be stored in the file (some fields may not need to be stored) must be serializable.
- You can import `Serializable` interface from `java.io` library

Note: All primitive data types are serializable by default. Other classes need to be checked separately. For this purpose, you can refer to the Java documentation.