COMPUTER CODING CLUB
MOTILAL NEHRU NATIONAL INSTITUTE OF TECHNOLOGY, PRAYAGRAJ

SOFTABLITZ

Exception Handling, Collections and Streams.

Exception Handling

- In Java, an Exception is an object that describes an exceptional condition in a piece of code. It is the equivalent of error-codes in C/C++.

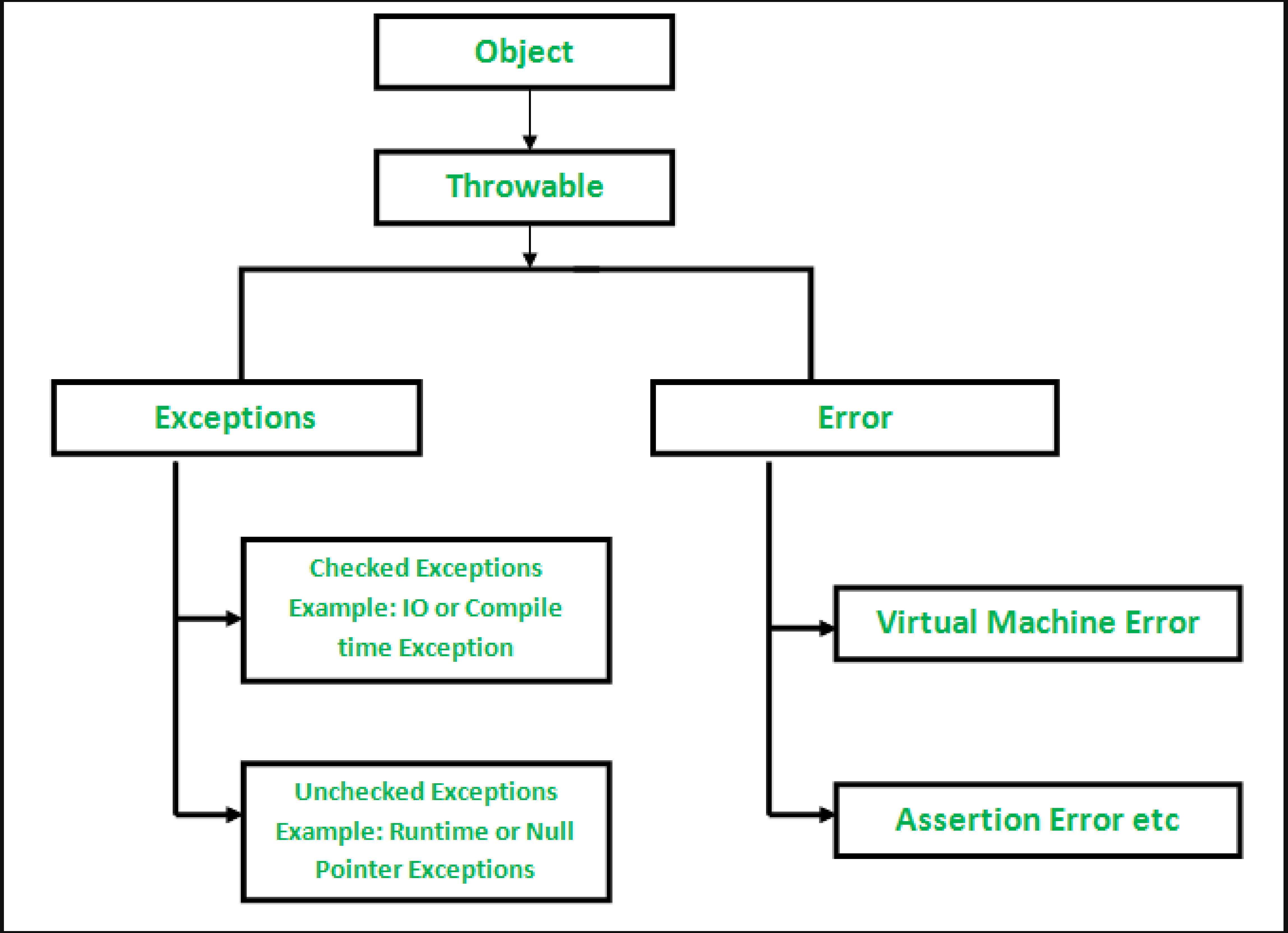
- An exceptional condition is when Java program has to tell other parts of the program that something bad has happened and it has failed. For example incorrect inputs, unexpected termination, invalid permissions, etc.

- Exceptions are generated by the Java runtime and can also be thrown by your code.

- Exception handling is the way of processing these exception objects in Java.

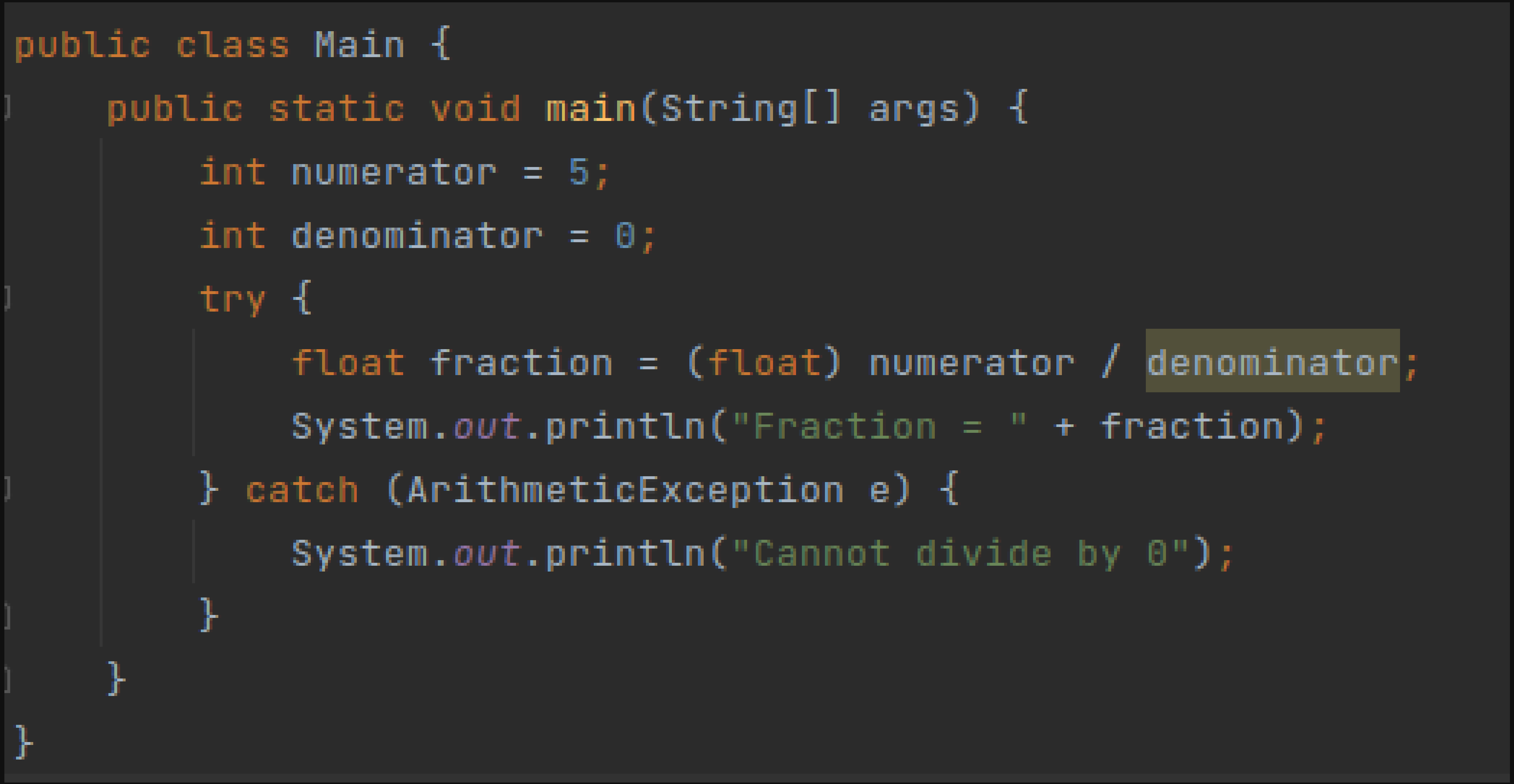
- Exception handling is managed by five keywords **try**, **catch**, **throw**, **throws** and **finally**.

Exception Class Hierarchy


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

try-catch

- The code that can throw an Exception is written under the **try block**.
- Exceptions are caught in the **catch block** and are handled in it in some rational manner.

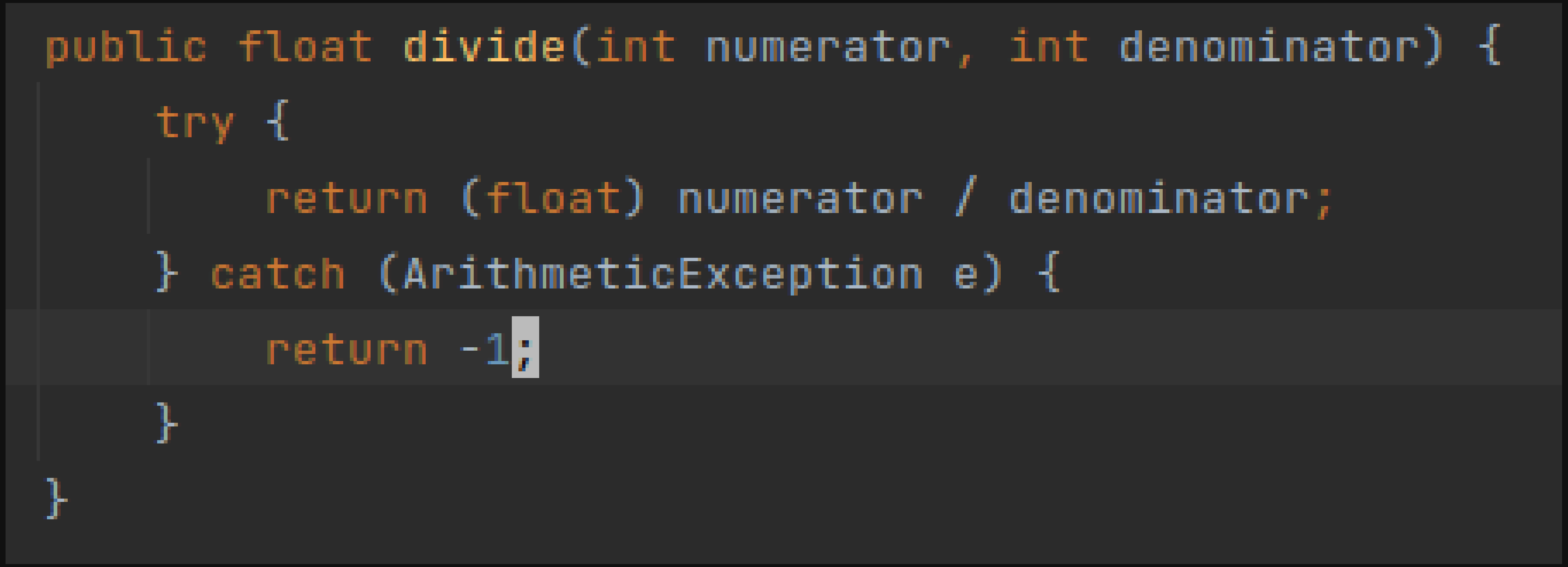


```java
public class Main {
    public static void main(String[] args) {
        int numerator = 5;
        int denominator = 0;
        try {
            float fraction = (float) numerator / denominator;
            System.out.println("Fraction = " + fraction);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by 0");
        }
    }
}
```

throw

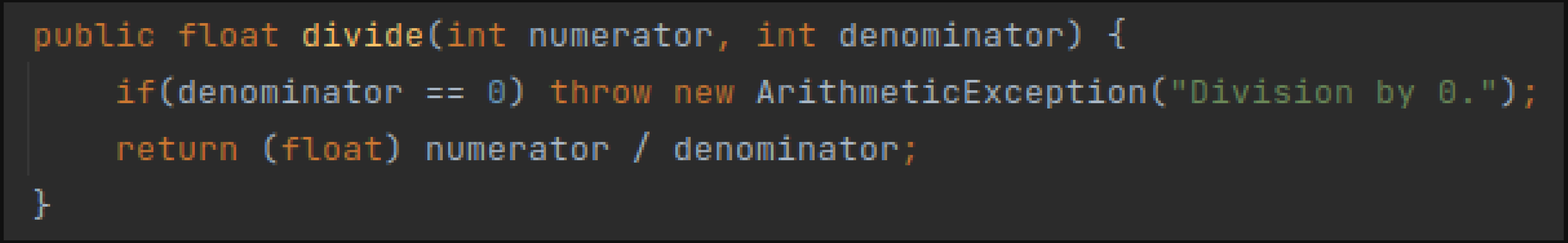
- In your codebase, you may have written code that might throw exceptions or you might have cases in which you would like to throw exceptions.
- The **throw** keyword is used to throw an exception by a Java program.
- Consider the throw keyword like the **return** keyword.
- The throw keyword can also be considered as a special-case return keyword.
- You can throw user defined or pre-defined exceptions.
- Thrown exceptions have to be handled by some exception somewhere else.
- Throwing an Exception is a way of transferring the responsibility of handling an Exception to some other part of the codebase.

Transfer of responsibility example



```java
public float divide(int numerator, int denominator) {
    try {
        return (float) numerator / denominator;
    } catch (ArithmeticException e) {
        return -1;
    }
}
```

v/s

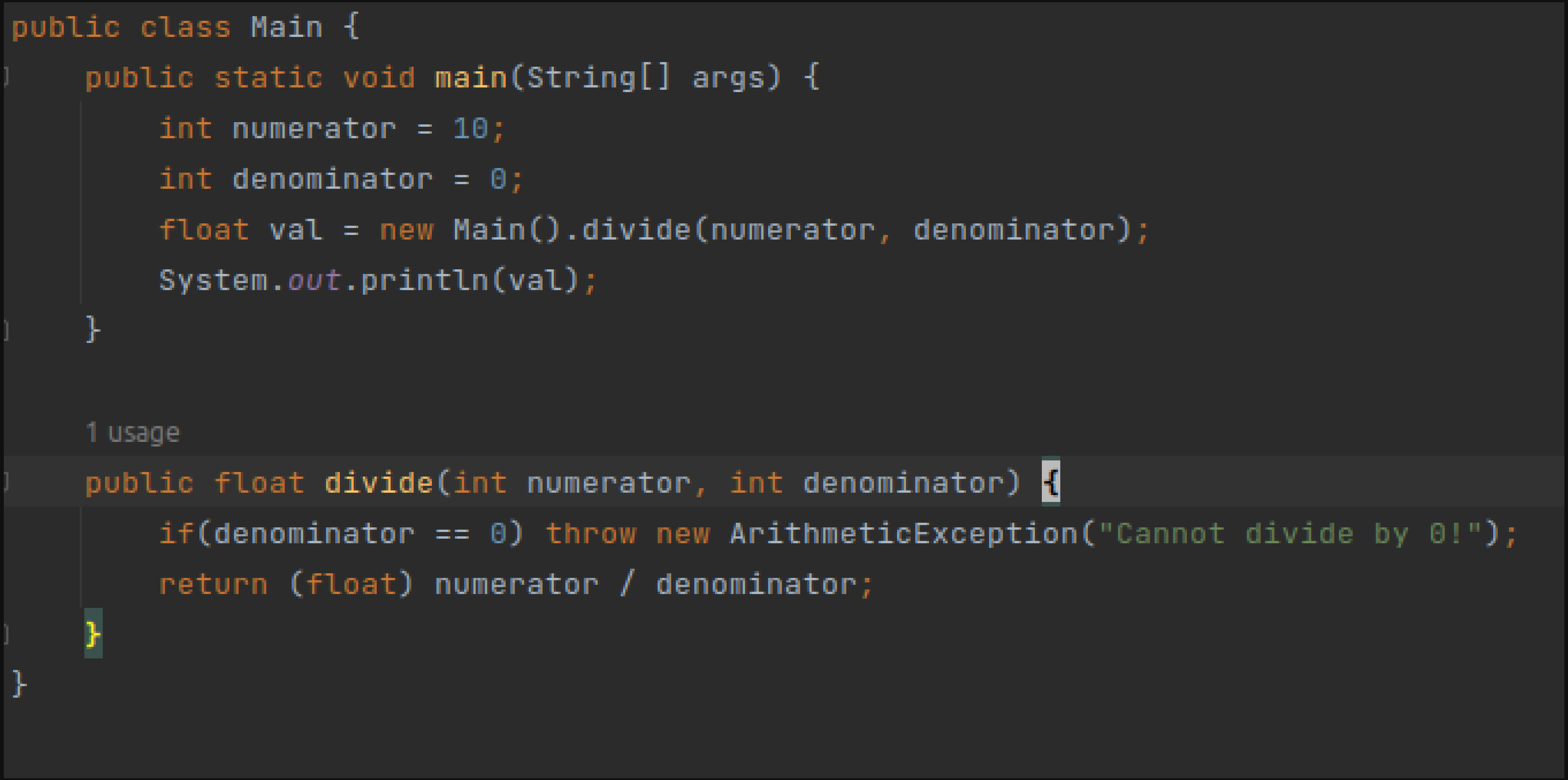


```java
public float divide(int numerator, int denominator) {
    if(denominator == 0) throw new ArithmeticException("Division by 0.");
    return (float) numerator / denominator;
}
```

throws

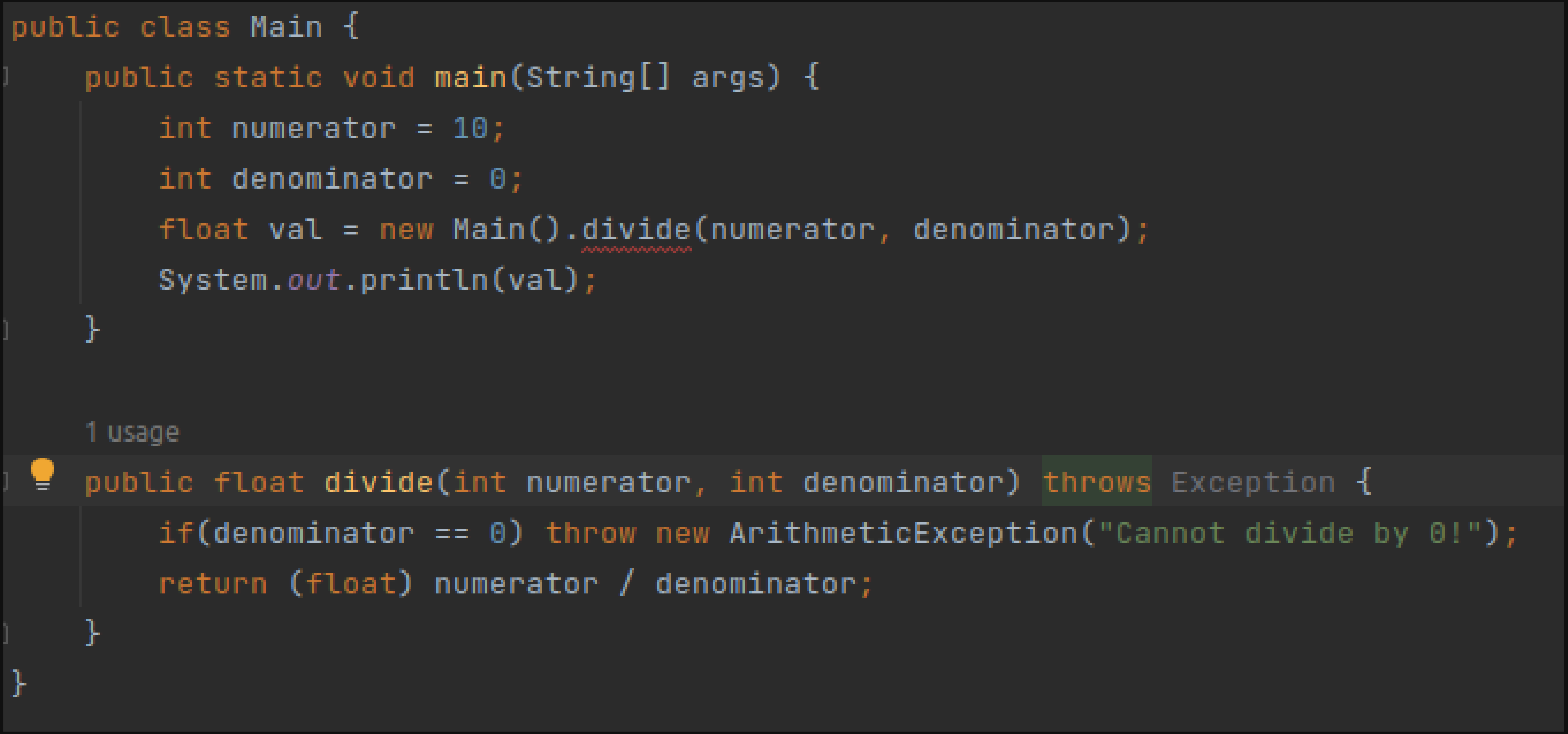
- In your code, you might want to notify other pieces of code when a method throws and Exception.
- The **throws** keyword is used to achieve the same, declare that the method throws an exception.
- Consider throws keyword similar to the return type in a function signature.
- When you declare a method with the throws keyword, you are making the acknowledgement of the Exception by the caller code compulsory.
- These two keywords (throw and throws) work in synergy.
    - throw - we are *telling a piece of code to throw* an Exception.
    - throws - we are *telling that a piece of code throws* an Exception.

Forcing the acknowledgment of an Exception



```java
public class Main {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;
        float val = new Main().divide(numerator, denominator);
        System.out.println(val);
    }

    1 usage
    public float divide(int numerator, int denominator) {
        if(denominator == 0) throw new ArithmeticException("Cannot divide by 0!");
        return (float) numerator / denominator;
    }
}
```

Forcing the acknowledgment of an Exception



```java
public class Main {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;
        float val = new Main().divide(numerator, denominator);
        System.out.println(val);
    }

    1 usage
    public float divide(int numerator, int denominator) throws Exception {
        if(denominator == 0) throw new ArithmeticException("Cannot divide by 0!");
        return (float) numerator / denominator;
    }
}
```

Question

There are two ways to remove this error. What are they?

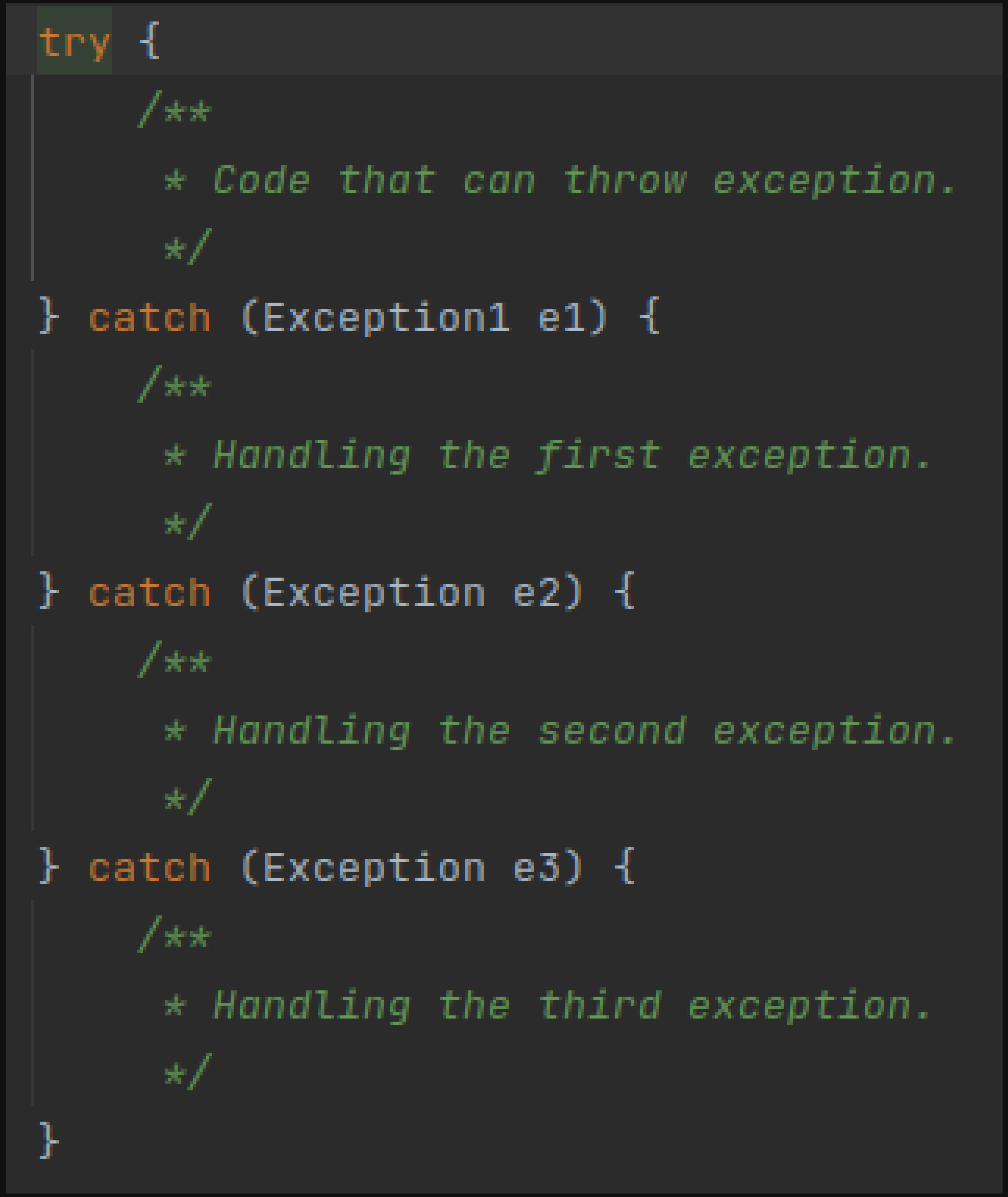
Question

There are two ways to remove this error. What are they?

- Handling the error with a try-catch block
- Transferring the responsibility even further using the throws statement.

Multiple catch blocks

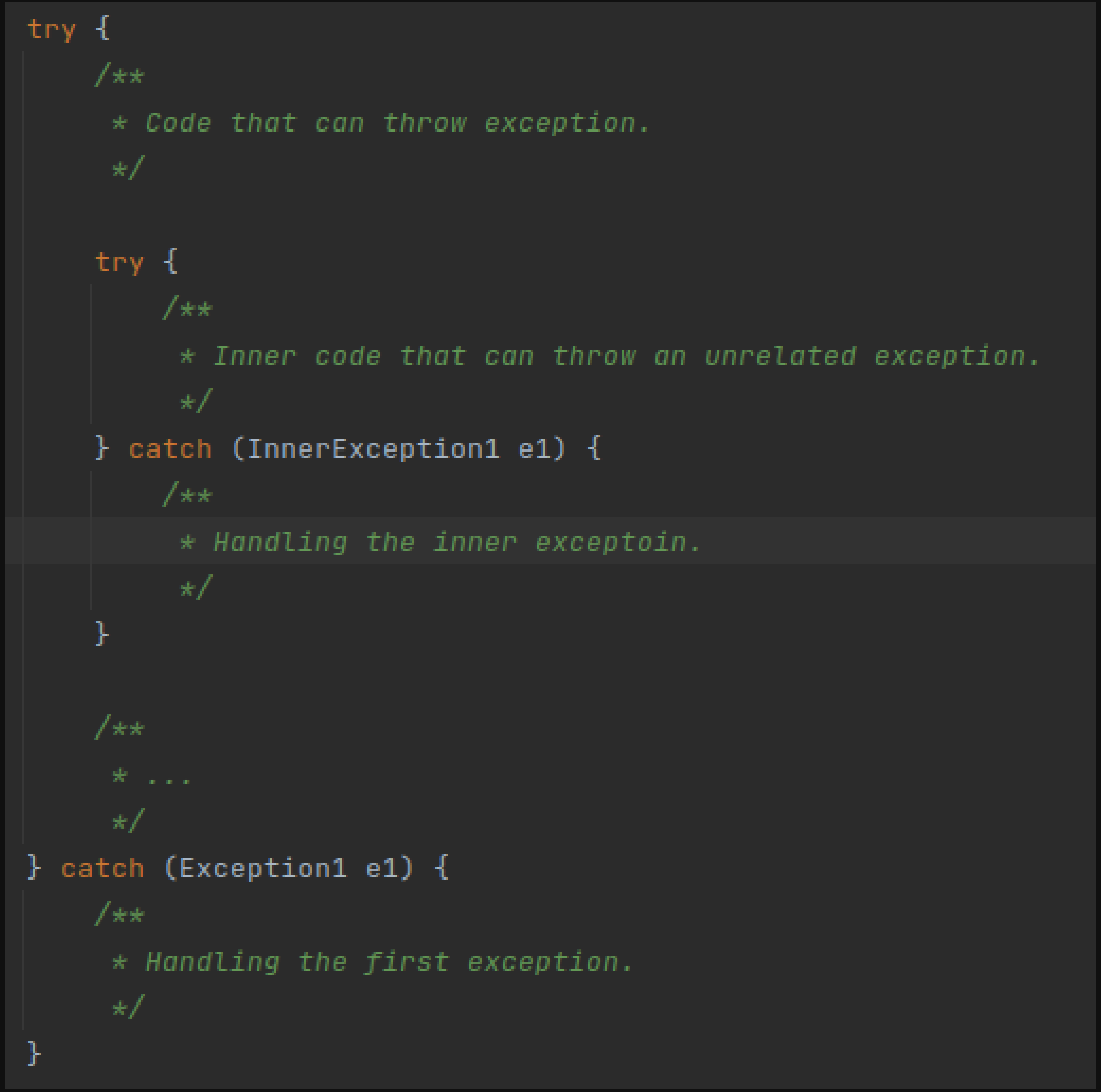
- A piece of code can throw more than one Exception.
- Different thrown exceptions might require different handling.
- We can provide multiple catch blocks to catch different kinds of exceptions and provide the logic of handling each of them.
- **Imp** - Care has to be taken of ordering these catch statements. Sub-classes should be caught before super-classes.



```
try {
    /**
     * Code that can throw exception.
     */
} catch (Exception1 e1) {
    /**
     * Handling the first exception.
     */
} catch (Exception e2) {
    /**
     * Handling the second exception.
     */
} catch (Exception e3) {
    /**
     * Handling the third exception.
     */
}
```

Nested try-catch

- A try (or a catch) block can contain another try-catch block within it.
- Think of it like nested if-else statements.



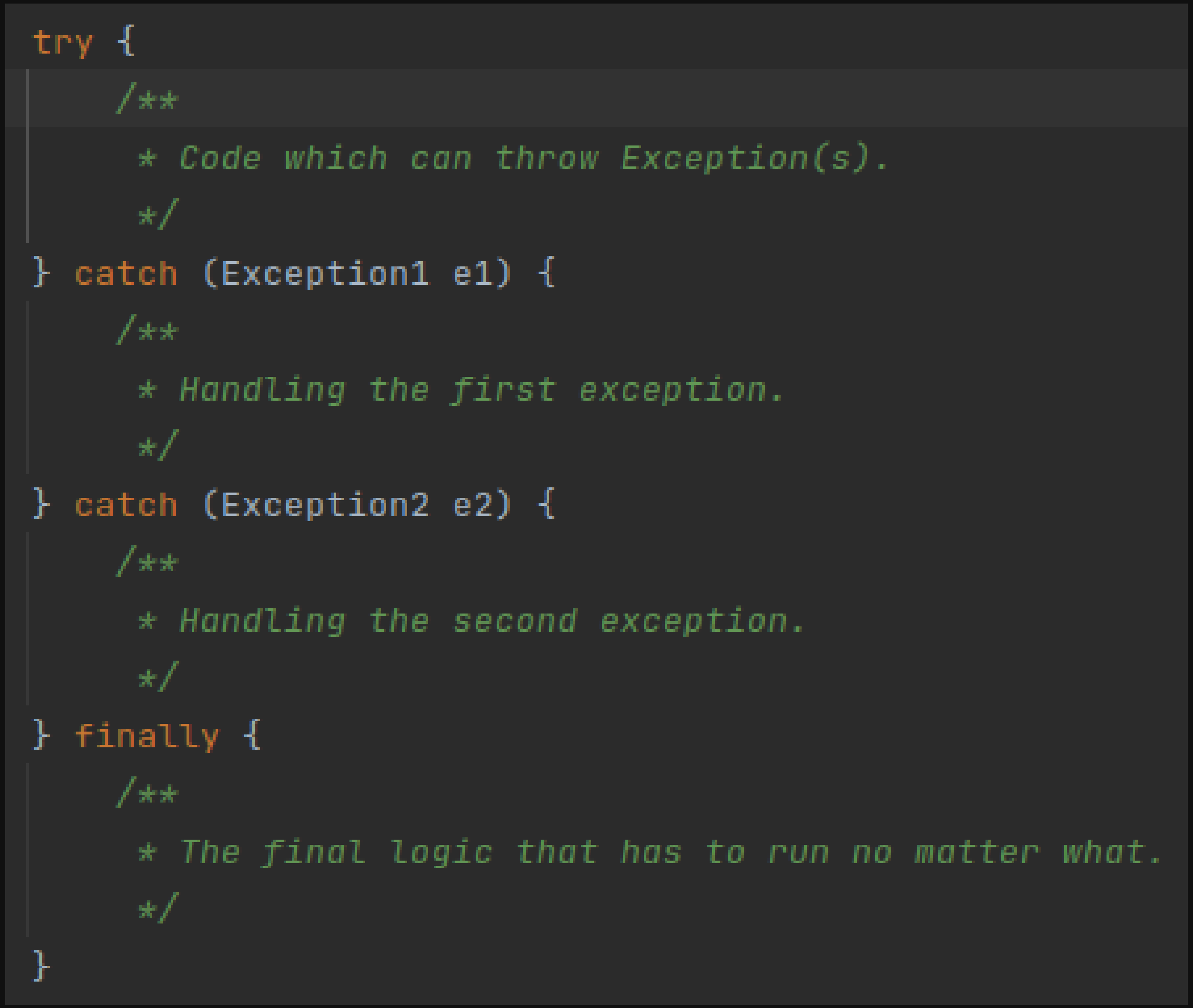
```java
try {
    /**
     * Code that can throw exception.
     */

    try {
        /**
         * Inner code that can throw an unrelated exception.
         */
    } catch (InnerException1 e1) {
        /**
         * Handling the inner exceptoin.
         */
    }

    /**
     * ...
     */
} catch (Exception1 e1) {
    /**
     * Handling the first exception.
     */
}
```

finally

- After an Exception is throw, no matter what catch statement has caught it, you might want to run some piece of logic at the very end.

- These might include, closing of Files, Streams or Sockets.

- To achieve this, we add a **finally** block at the end of our try-catch block. Which will contain the statements that we want to execute no matter what.



```
try {
    /**
     * Code which can throw Exception(s).
     */
} catch (Exception1 e1) {
    /**
     * Handling the first exception.
     */
} catch (Exception2 e2) {
    /**
     * Handling the second exception.
     */
} finally {
    /**
     * The final logic that has to run no matter what.
     */
}
```

Custom Exceptions

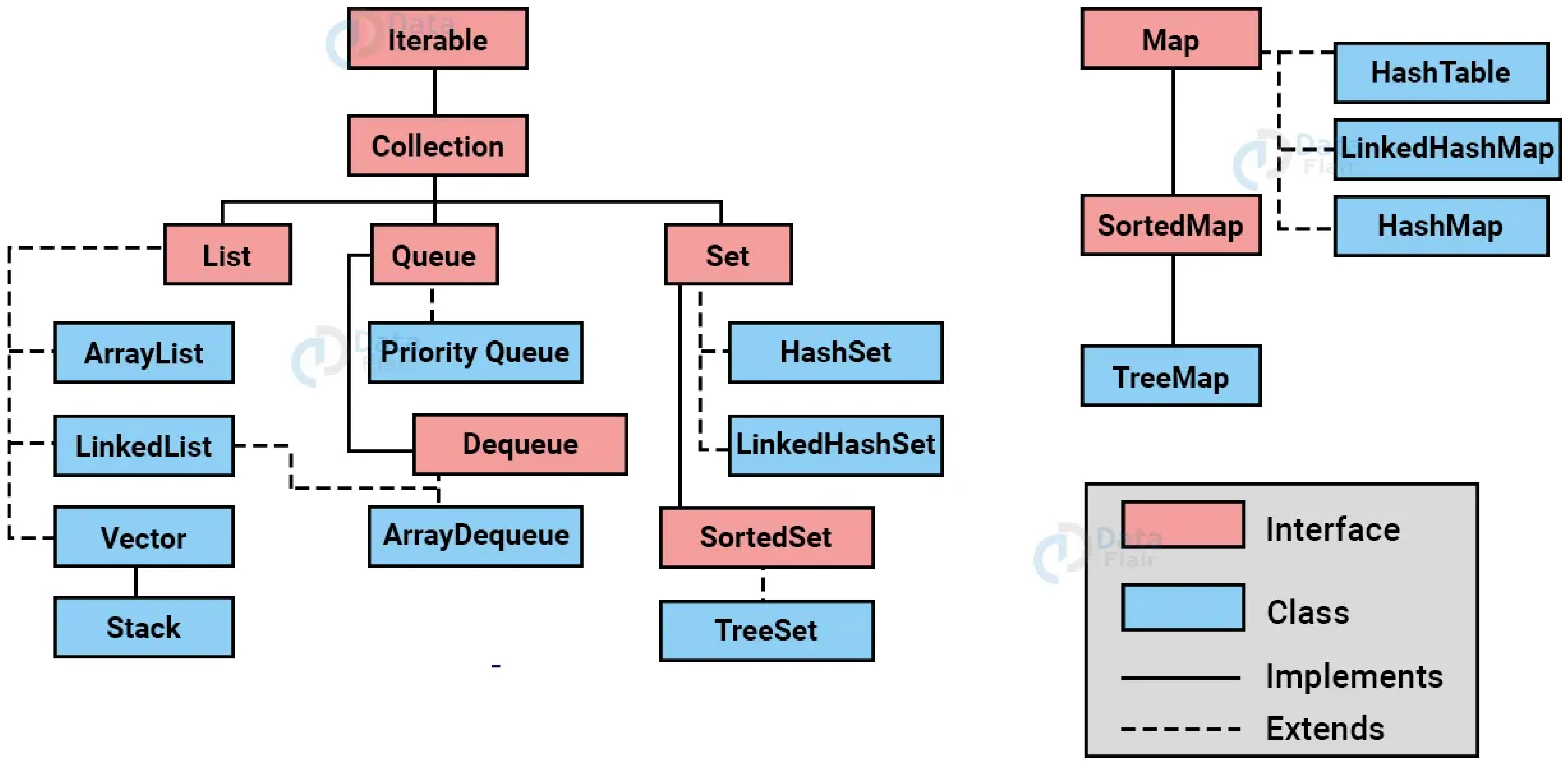
- In many cases, you might need to declare your own Exceptions to increase the clarity of your code.
- Many libraries throw custom Exceptions to inform their users of any exceptional behavior.
- For example, the JavaFx library throws a LoadException when it is unable to load an FXML file.
- You might also need custom exceptions in your own codebase to make it more readable and for easier debugging.
- You can create custom Exceptions by simply extending the Exception class and then use them as you would any pre-defined Exception.

Questions?

Collections Framework

- The Collections Framework is a set of classes and interfaces that reside in the java.util package and provide functionality for handling groups of classes.
- The framework was added later to the Java standard to ensure uniformity between different types of collections and also to provide extensibility for user defined collections.
- The collections framework in Java is very similar to the Standard Template Library in C++.
- The main interfaces/classes in the Collection framework are,
    - **List, Queue, Set and Map**
    - **Iterator**
    - **Collections**
    - **Comparator**
    - **Comparable**

Hierarchy of Collection Framework in Java



The Collection Interface

- This is the base interface that all the Collections (except Map) extend.
- The Collection interface has the following functions
  - **add(E Element)** : Boolean, inserts a new element.
  - **remove(E Element)** : Boolean, removes an element from the collection.
  - **isEmpty()** : Boolean, returns true if the collection has no elements.
  - **size()** : Returns how many elements we have stored in the collection.
  - **clear()**: void, removes all elements from the collection.
  - **contains(E Element)**: Boolean, verifies if some element is stored in the collection.
- All the functions that extend the Collection interface provide an implementation for the above methods.

The List Interface

- The List interface declares a behavior of a collection that stores a sequence of elements.
- Elements can be accessed or inserted by their position in the list using a zero-based indexing.
- A list may contain duplicate elements.
- The List interface defines some additional methods like,
  - indexOf(E object) - to get the index of an element.
  - remove(int index) - to remove an element at a given index.
  - get(int index) - to get an element at a given index.
- Two concrete classes that extend the List interface are ArrayList and LinkedList.

The Set Interface

- The List interface declares a behavior of a collection that does not allow duplicate elements.
- There are two additional interfaces that implement the Set interface which are,
  - SortedSet - Represent sets which have a concept of ordering.
  - NavigableSet - Represents sets which support retrieval of elements based on the closest match to a given value.
- The Set interface provides an *of()* static method that returns a set of elements.
- Two concrete classes that extend the Set interface are HashSet and TreeSet.

The Queue Interface

- The Queue interface declares a behavior of a collection that allows accessing elements in a first-in first-out fashion.
- There is an additional Deque interface that extends a Queue and declares the behavior of a doubly-ended queue.
- The extra functions provided by the queue interface include,
    - peek() - Returns the element at the top of the queue. Does not remove the element. If there is no element, null is returned.
    - poll() - Returns the element at the top of the queue while popping it from the queue. If there is no element, null is returned.
    - element() - Returns the element at the top of the queue. If there is no element, NoSuchElementException is thrown.
    - remove() - Removes and returns the element at the top of the queue. If there is no element, NoSuchElementException is thrown.
- A concrete class that extends the Queue interface is PriorityQueue.

The Iterator Interface

- The Iterator interface allows us to access elements of the collection and is used to iterate over the elements in the collection
- Iterator can be used to iterate only in the forward direction.
- Iterator behaves like a cursor in a text editor and thus is also known as a universal cursor for Collection API.
- The Iterator interface provides the next() and hasNext() functions.
    - next() - returns the reference to the next object in the collection.
    - hasNext() - returns whether there are elements further in the collection or not.
- The Iterator interface along with the Iterable interface can be used to define custom Collections of objects.

The Collections Class

- The Collections class is a utility class in the Collections framework.
- It provides various algorithms which act on objects of the Collection interface.
- Some functions in the Collections class include,
    - addAll() - To add the provided elements to the passed collection.
    - sort() - To sort the collection. Only accepts parameters of the List type.
    - binarySearch() - To search elements in a collection. Only accepts parameters of the List type.
    - and many more...

The Comparable Interface

- The Comparable interface is implemented by classes that are comparable and have a concept of ordering.
- For example, a Student class can implement the Comparable interface where the Students can be ordered by their roll numbers.
- The classes that implement the Comparable interface have to implement the *compareTo(E obj)* method.
- The method returns,
    - A negative number, the current object should come before the passed object.
    - Zero, if the current object and the passed objects are equal.
    - A positive number, the current object should come after the passed object.

The Comparator Interface

- A comparator is an object that is used to compare two objects and decide which one is larger and smaller.
- A comparator is applied externally to a class, i.e. the classes which have to be compared are passed to the *compare(Object a, Object b)* function which compares the objects on the basis of some criteria.
- A comparator can be passed to algorithms in the Collections class which can be used to make comparisions between objects for various algorithms (like sorting).
- A comparator extends the Comparator interface which implements the *compare(Object a, Object b)* function. The function returns,
    - A negative number, if the first object is less than the second object.
    - Zero, if the objects are equal.
    - A positive number, if the first object is greater than the first object.

Working with Maps

- A map is a data structure that holds key-value pairs.
- The keys are unique values that have a corresponding value.
- Maps do not implement the Iterable interface, which means we cannot iterate over the elements of the Map using a for loop.
- The Map interface provide two basic operations, *get()* and *put()*.
- The Map interface also provides the *of()* method that returns a Map of passed objects.
- The implementations of Map include,
  - TreeMap - Uses a red-black tree to implement the key store.
  - HashMap - Uses hashing to implement the key store.

Legacy Data Structures

- Before the Collection framework existed, there were data structures in Java.
- They included the following classes - **Vector, Stack, Dictionary, Hashtable and Properties.**
- The problem with these data structures was that every data structure had it's own API and there was no standard that they agreed upon.
- These data structures still exist in Java to provide backwards compatibility.
- Using these data structures is not recommended and you should avoid them too.

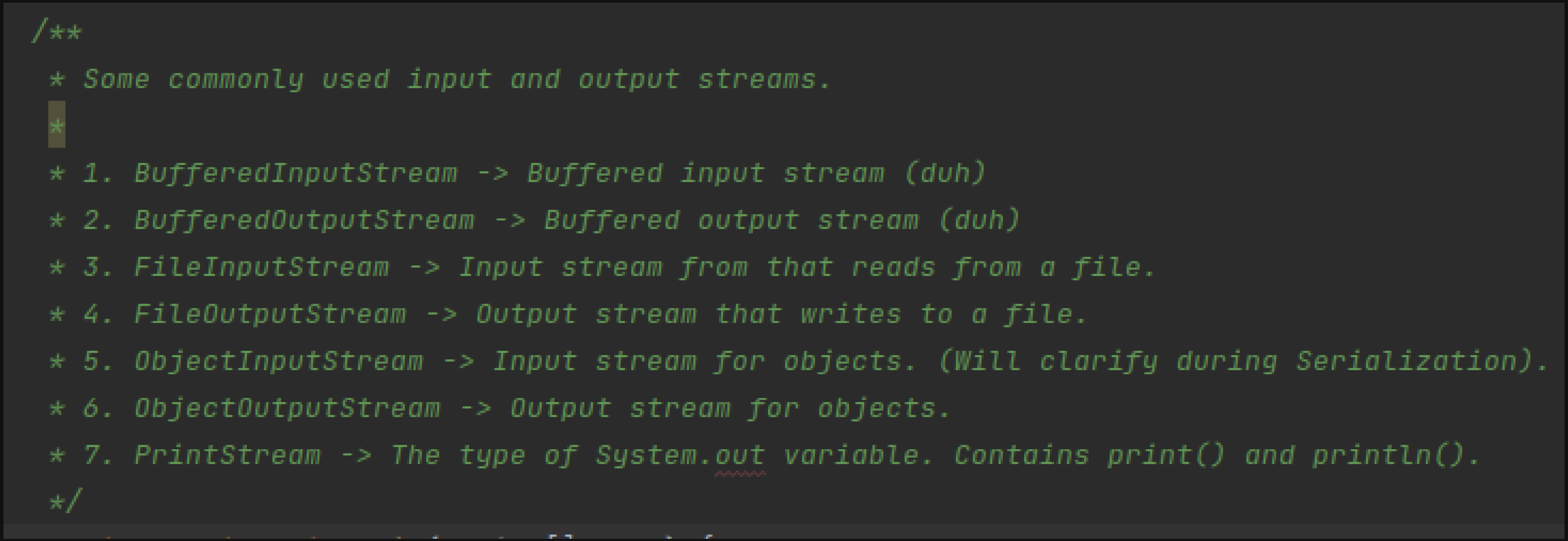
Questions?

The Stream API

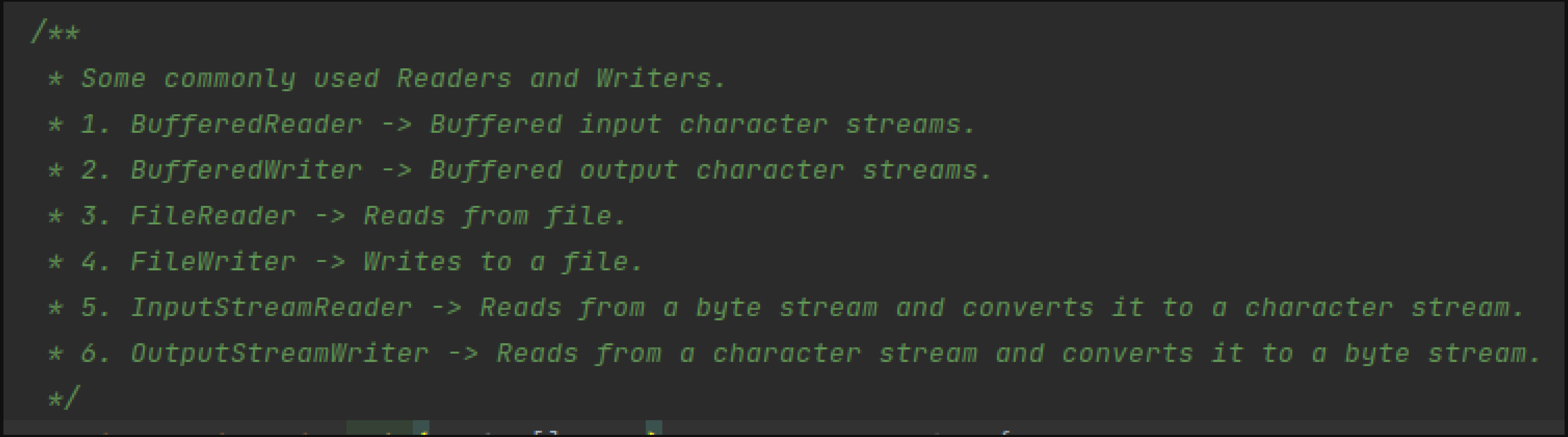
- Java programs perform IO through streams.
- A stream is an abstraction that either produces or consumes data.
- A stream is like a sequence of data in the form of bits (or characters).
- There are input and output streams. Input streams read data from an external source, like a file, socket or the standard input. Output streams writie data to an external source, like disk file or standard output.
- Streams abstract the source and destination, so the program you use the data in does not have to worry about the source of the data.
- There are two types of streams based on the basic unit of the stream.
    - Byte Oriented - These make handling of binary objects convenient. For example, serialized objects, files, etc.
    - Character Oriented - These make handling of textual data convenient. They use Unicode encoding to make character handling much more convenient.

I/O Streams and Readers and Writers

- Streams can be categorized on the basis of two parameters, whether they are byte oriented or character oriented and whether they are reading from a source or writing to a destination.
- Byte oriented streams - There are two types,
    - InputStreams
    - OutputStreams
- Character oriented streams - There are two types,
    - Readers
    - Writers
- There are several concrete implementations of these classes. You'll be using a lot of them in your code.



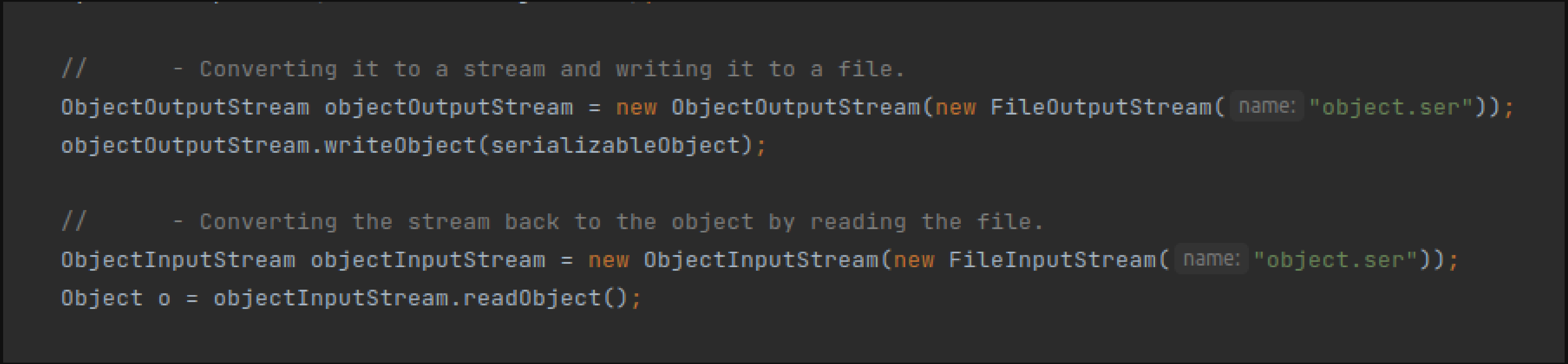
```
/**
 * Some commonly used input and output streams.
 *
 * 1. BufferedInputStream -> Buffered input stream (duh)
 * 2. BufferedOutputStream -> Buffered output stream (duh)
 * 3. FileInputStream -> Input stream from that reads from a file.
 * 4. FileOutputStream -> Output stream that writes to a file.
 * 5. ObjectInputStream -> Input stream for objects. (Will clarify during Serialization).
 * 6. ObjectOutputStream -> Output stream for objects.
 * 7. PrintStream -> The type of System.out variable. Contains print() and println().
 */
```



```java
/**
 * Some commonly used Readers and Writers.
 * 1. BufferedReader -> Buffered input character streams.
 * 2. BufferedWriter -> Buffered output character streams.
 * 3. FileReader -> Reads from file.
 * 4. FileWriter -> Writes to a file.
 * 5. InputStreamReader -> Reads from a byte stream and converts it to a character stream.
 * 6. OutputStreamWriter -> Reads from a character stream and converts it to a byte stream.
 */
```

Serialization

- Serialization has great use in the real world.
- Objects exist in memory, but there should be some way to persist them. Maybe store them in a database or write them to a file.
- Similarly, a mechanism to create objects from stored data must exist as well.
- This is where serialization comes in.
- Formally, serialization is the process of converting an object to a byte stream.
- Similarly, de-serialization is the process of converting a byte stream to an object.
- After we have a byte stream, we can pretty much do anything with it, send it over a socket, write it to a file or print it on the console.
- These streams can be provided to the ObjectInputStream and ObjectOutputStream objects which can read objects from the stream or write objects to the stream.



```java
//      - Converting it to a stream and writing it to a file.
ObjectOutputStream objectOutputStream = new ObjectOutputStream(new FileOutputStream( name: "object.ser"));
objectOutputStream.writeObject(serializableObject);

//      - Converting the stream back to the object by reading the file.
ObjectInputStream objectInputStream = new ObjectInputStream(new FileInputStream( name: "object.ser"));
Object o = objectInputStream.readObject();
```

Fin.