

All the classes of exception are present in java.lang package

A **Partially Checked Exception** is:

A parent exception class whose subclasses include both checked and unchecked exceptions.

In Java, the only class that fits this definition is:

java.lang.Exception

**🔍 Why is Exception Partially Checked?**

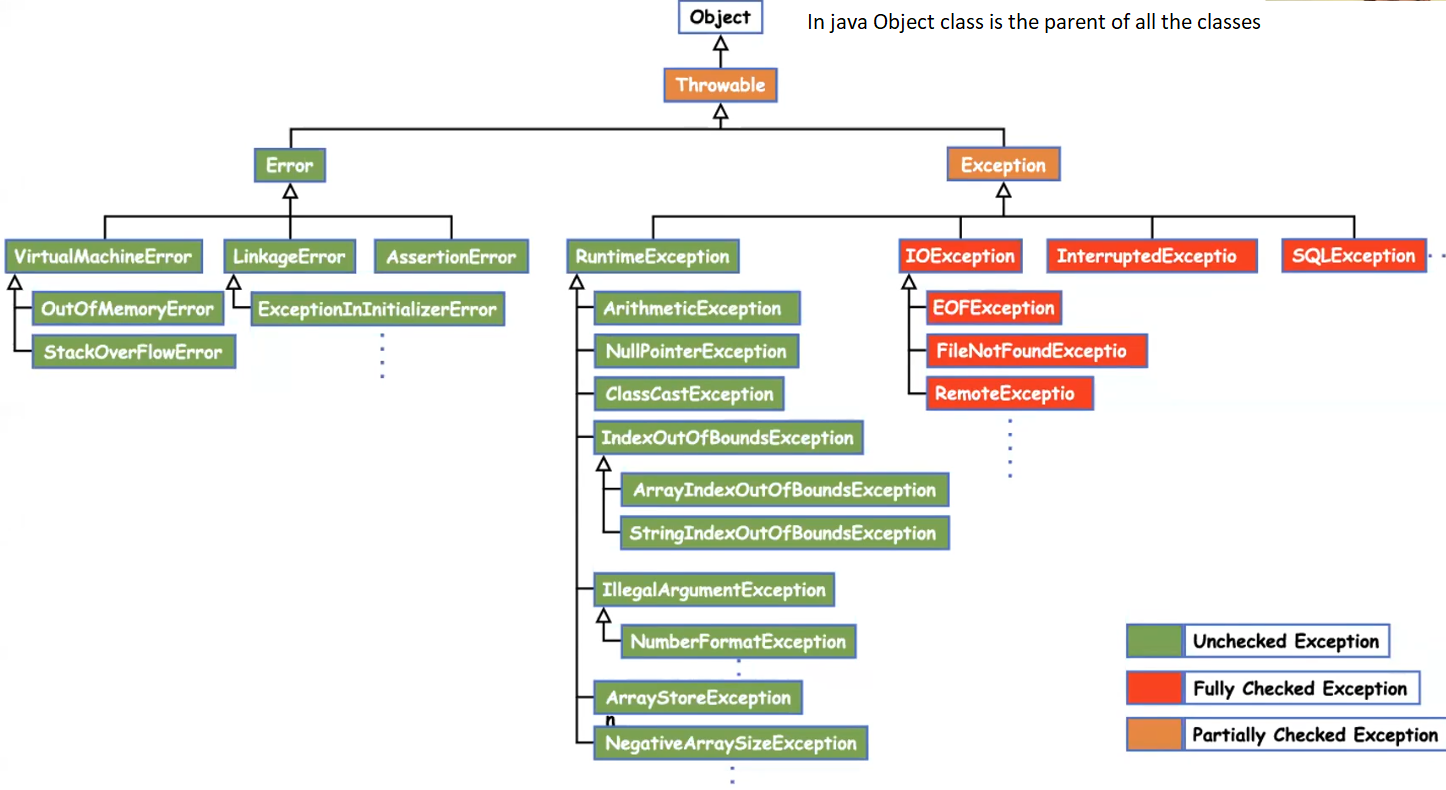
Because:

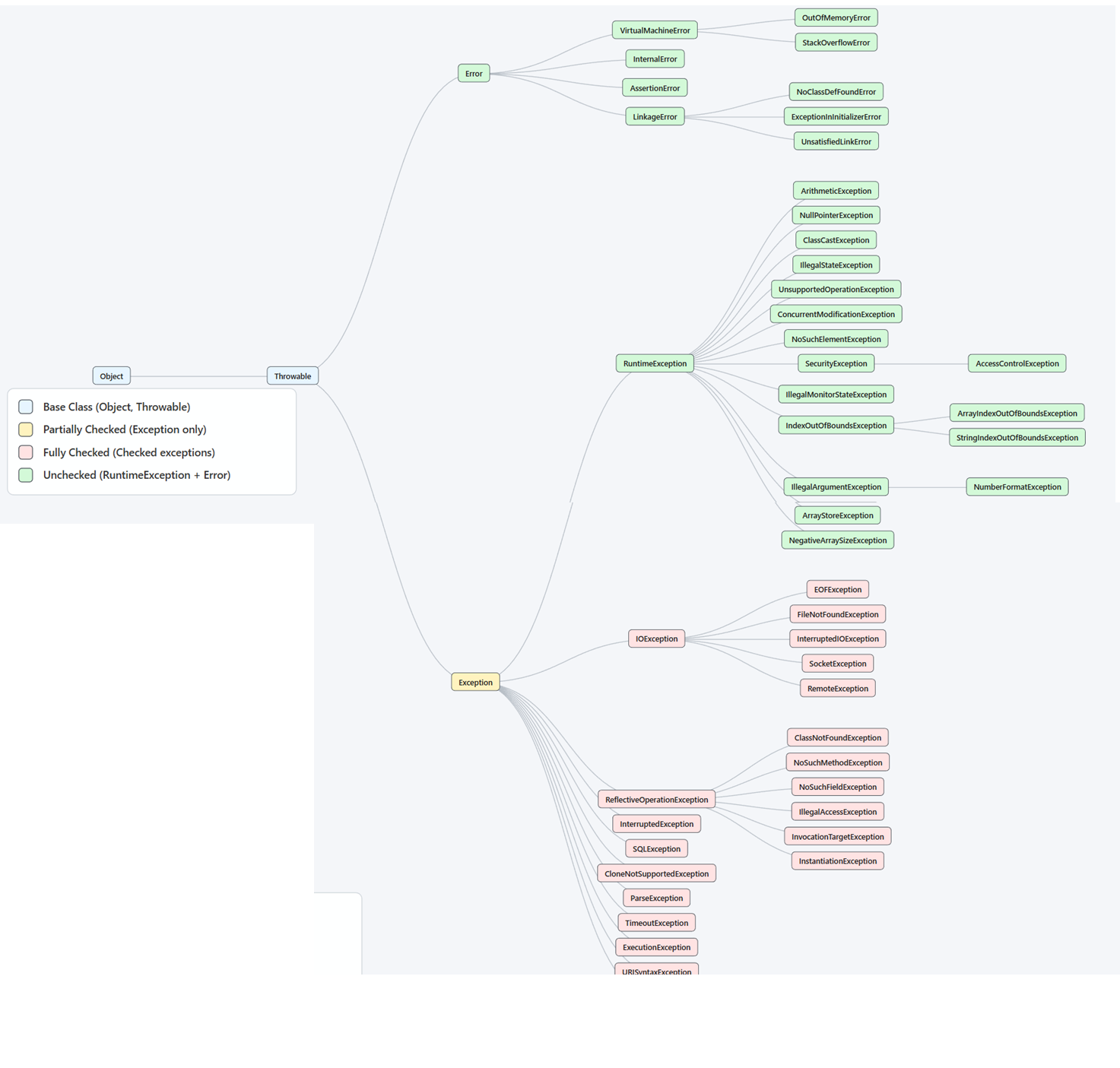
Throwable  
 └── Exception ← Partially Checked  
 ├── RuntimeException ← Unchecked  
 └── IOException ← Checked

So:

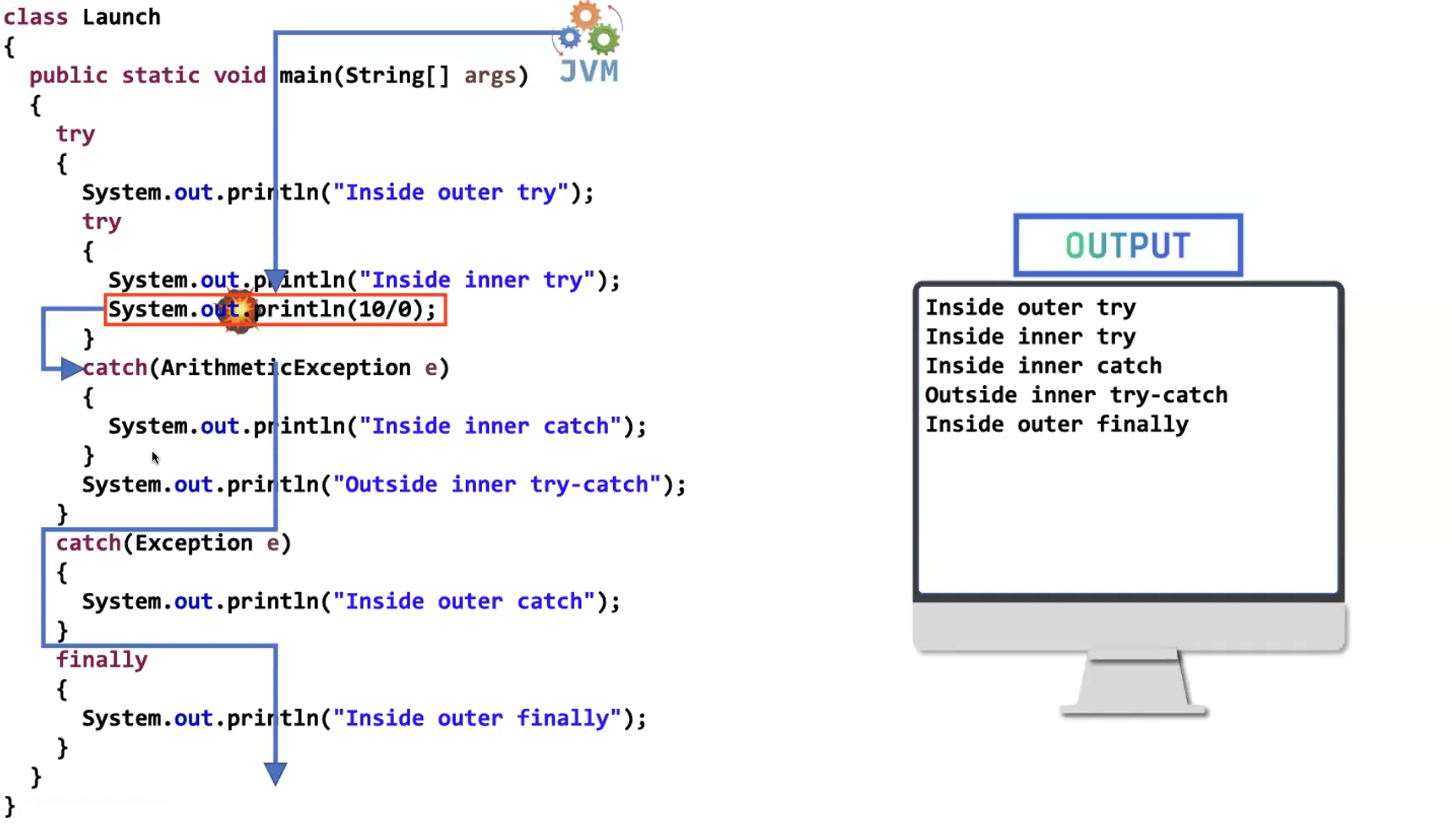
* Some children are **checked**
* Some children are **unchecked**

That’s why Exception is called **partially checked**.





Nested try-catch example



**✅ Correct Definition of Partially Checked Exception**

Your statement:

It is such a parent Exception class where its child consists of both checked and unchecked exception classes.

✔ This is almost correct.  
Let me refine it professionally:

**🔹 Correct Definition**

A **Partially Checked Exception** is:

A parent exception class whose subclasses include both checked and unchecked exceptions.

In Java, the only class that fits this definition is:

java.lang.Exception

**🔍 Why is Exception Partially Checked?**

Because:

Throwable  
 └── Exception ← Partially Checked  
 ├── RuntimeException ← Unchecked  
 └── IOException ← Checked

So:

* Some children are **checked**
* Some children are **unchecked**

That’s why Exception is called **partially checked**.

**❌ Now the Important Correction**

You wrote:

If it is a partially checked exception it should be handled compulsorily

This statement needs correction.

**🔴 Correct Rule:**

If you directly use Exception, it must be handled or declared.

Example:

void test() throws Exception

Compiler forces handling.

Why?

Because Exception may represent a checked exception.

**But Important Clarification**

Not every subclass of Exception must be handled.

For example:

NullPointerException

This extends:

RuntimeException → Exception

But it is **unchecked**, so:

✔ No compulsory handling.

**🎯 Final Correct Understanding**

| **Type** | **Must Handle?** |
| --- | --- |
| Exception (parent) | ✅ Yes |
| Checked exceptions (like IOException) | ✅ Yes |
| RuntimeException | ❌ No |
| Error | ❌ No |

**🧠 Interview-Level Explanation (Short Version)**

Exception is called partially checked because it has both checked and unchecked subclasses.  
When declared directly, it must be handled because it may represent a checked exception.

**🔥 Final Corrected Statement (You Can Write This)**

A partially checked exception is a parent exception class whose subclasses include both checked and unchecked exceptions. In Java, Exception is partially checked. When used directly, it must be handled or declared because it may represent a checked exception.

If you want, I can now give:

* 🔥 5 tricky interview questions on this
* 🔥 Memory trick to never forget this
* 🔥 Code examples showing compiler behavior

Tell me what you want next 💪

We can write another try-catch in “inside inner try”. Similarly we can write another try catch in “inside inner catch”. Even in the finally block also we can write try-catch. We can write any number of times nested try - catch there is no restriction.

Good topic Pavan 🔥  
This is called **Nested try–catch**.

Your statements are mostly correct ✅  
Let’s make them **100% technically accurate + interview ready**.

**✅ 1️⃣ Can we write try–catch inside another try?**

✔ YES.

Example:

class NestedTryDemo {  
  
 public static void main(String[] args) {  
  
 try {  
 System.out.println("Outer try started");  
  
 try {  
 int a = 10 / 0; // ArithmeticException  
 }   
 catch (ArithmeticException e) {  
 System.out.println("Inner catch handled: " + e);  
 }  
  
 System.out.println("Outer try continues");  
  
 }   
 catch (Exception e) {  
 System.out.println("Outer catch handled: " + e);  
 }  
  
 System.out.println("Program ends");  
 }  
}

✔ Inner exception handled by inner catch  
✔ Outer catch not executed

**✅ 2️⃣ Can we write try–catch inside catch block?**

✔ YES.

Example:

class TryInsideCatch {  
  
 public static void main(String[] args) {  
  
 try {  
 int a = 10 / 0; // ArithmeticException  
 }   
 catch (ArithmeticException e) {  
  
 System.out.println("Outer catch executed");  
  
 try {  
 String s = null;  
 System.out.println(s.length()); // NullPointerException  
 }   
 catch (NullPointerException ex) {  
 System.out.println("Inner catch inside catch handled: " + ex);  
 }  
 }  
 }  
}

✔ Completely valid.

**✅ 3️⃣ Can we write try–catch inside finally block?**

✔ YES.

Example:

class TryInsideFinally {  
  
 public static void main(String[] args) {  
  
 try {  
 int a = 10 / 0;  
 }   
 catch (ArithmeticException e) {  
 System.out.println("Handled arithmetic exception");  
 }   
 finally {  
  
 try {  
 String s = null;  
 System.out.println(s.length());  
 }   
 catch (NullPointerException e) {  
 System.out.println("Handled exception inside finally");  
 }  
  
 System.out.println("Finally block executed");  
 }  
 }  
}

✔ Allowed  
✔ Works fine

**✅ 4️⃣ Can we write unlimited nested try–catch?**

✔ YES.

There is **no restriction in Java language**.

But ⚠ Important:

Just because we can, doesn't mean we should.

Deep nesting:

* Reduces readability
* Makes debugging hard
* Not recommended in real projects

**🧠 Important Execution Rule**

When exception occurs:

1. JVM first checks **current try block**
2. If not handled → moves to outer try
3. If still not handled → JVM default handler

This is called:

👉 **Exception Propagation**

**🎯 Interview-Ready Correct Statement**

You can write nested try–catch blocks inside:

* try
* catch
* finally

There is no language restriction on nesting depth.  
However, excessive nesting is discouraged because it reduces readability and maintainability.

**⚠ Important Caution**

If you write:

try {  
 // some code  
}  
finally {  
 // no catch  
}

This is valid.

But:

finally {  
 try {  
 }  
}

Is also valid.

But:

try {  
}

Alone is NOT valid.

Must have:

* catch  
  OR
* finally

**🔥 Industry Reality**

In real Spring Boot projects:

Deep nested try–catch is avoided.

Instead we use:

* Global exception handling
* @ControllerAdvice
* Custom exceptions

