Exception Hierarchy

The inheritance relationship between Throwable, Error, Exception, and RuntimeException -- how every exception descends from a single shared root, how a catch block can target any ancestor of a type (polymorphic catching), and how to query that hierarchy at runtime with instanceof. The 3rd lesson in the Exception Handling series.

Beginner 20 min

Last updated: November 9, 2026
Read online: https://www.learnforgex.com/en/topics/exception-hierarchy

---

In "Introduction to Exceptions" you saw what an exception is, how it gets thrown, and how it propagates; in "Try-Catch and Finally" you learned how to catch one and clean up after it. But so far you've treated classes like `ArithmeticException`, `NumberFormatException`, and `ArrayIndexOutOfBoundsException` as if they were independent, unrelated names. They aren't — they're all branches of a single shared tree, and knowing the shape of that tree explains which `catch` block will actually catch what, why `catch (Exception e)` can be dangerous, and why some problems (like `StackOverflowError`) should almost never be caught by an ordinary `catch` block at all.

What Is the Exception Hierarchy?

Every exception and error in Java descends from `java.lang.Throwable`. `Throwable` has exactly two direct subclasses: `Error` and `Exception`. `Exception` in turn has its own subclass, `RuntimeException`. These three classes — `Throwable`, `Error`, and `Exception`, especially `RuntimeException` — are the common ancestor of nearly every exception you've seen so far: `NumberFormatException` is an `IllegalArgumentException`, which is a `RuntimeException`, which is an `Exception`, which is a `Throwable`.

Why Does It Exist?

Without a hierarchy, every exception type would be a completely separate, unrelated class — catching 10 different exceptions a method might throw would mean writing 10 separate `catch` blocks, sharing no common behavior at all (not even `getMessage()` or `getStackTrace()`). The hierarchy provides both a common interface (every `Throwable` has a message and a stack trace) and **polymorphic catching**: a single `catch (RuntimeException e)` block can catch ANY subclass beneath `RuntimeException` in one line.

History

The `Throwable` / `Error` / `Exception` / `RuntimeException` structure has been the same since Java's very first release (1.0, 1996). The designers deliberately made `Error` a separate branch: `Exception` represents conditions application code can reasonably react to, while `Error` represents problems with the JVM itself, usually unrecoverable ones. This split is also the foundation of Java's checked-exception philosophy (covered in full in "Checked vs Unchecked Exceptions") — but the hierarchy itself is a more basic classification, independent of the checked/unchecked distinction.

Throwable: The Root of the Hierarchy

`Throwable` is the common ancestor of everything that can be caught or thrown — `Error`, `Exception`, and `RuntimeException` all included. Methods like `getMessage()`, `getStackTrace()`, `printStackTrace()`, and `getCause()` are all defined on `Throwable`, so no matter which concrete exception type you're working with, they're all shared.

```
// Every exception you've seen so far -- ArithmeticException, NumberFormatException,
// ArrayIndexOutOfBoundsException, IllegalArgumentException -- is a REAL class,
// and every one of those classes extends SOMETHING, all the way up to a single
// shared root: Throwable. This example walks that chain using reflection
// (getSuperclass(), see the Reflection lesson for the full API) to print it
// out directly, instead of just describing it.
public class ThrowableHierarchyWalkExample {
    public static void main(String[] args) {
        printHierarchy(new NumberFormatException("example"));
        System.out.println();
        printHierarchy(new ArithmeticException("example"));
        System.out.println();
        printHierarchy(new StackOverflowError());
    }

    private static void printHierarchy(Throwable throwable) {
        System.out.println("Hierarchy for " + throwable.getClass().getSimpleName() + ":");
        Class<?> current = throwable.getClass();
        while (current != null) {
            System.out.println("  " + current.getName());
            current = current.getSuperclass();
        }
        // Output for NumberFormatException, for example:
        //   java.lang.NumberFormatException
        //   java.lang.IllegalArgumentException
        //   java.lang.RuntimeException
        //   java.lang.Exception
        //   java.lang.Throwable
        //   java.lang.Object
    }
}
```

Expected Output

```
Hierarchy for NumberFormatException:
  java.lang.NumberFormatException
  java.lang.IllegalArgumentException
  java.lang.RuntimeException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object

Hierarchy for ArithmeticException:
  java.lang.ArithmeticException
  java.lang.RuntimeException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object

Hierarchy for StackOverflowError:
  java.lang.StackOverflowError
  java.lang.VirtualMachineError
  java.lang.Error
  java.lang.Throwable
  java.lang.Object
```

This example uses `getSuperclass()` (see the "Reflection" lesson) to actually walk an exception's class chain upward and print it — `NumberFormatException`'s chain climbs through `RuntimeException` and `Exception` to `Throwable`, while `StackOverflowError`'s chain never touches `Exception` at all, climbing straight through `Error` to `Throwable`. The two branches only meet at the very top.

Error: The JVM's Own Problems

`Error` and its subclasses (`StackOverflowError`, `OutOfMemoryError`, and similar) represent serious, JVM-level conditions that application code normally cannot prevent or recover from. `StackOverflowError` is thrown when the call stack overflows — typically from infinite or excessively deep recursion.

```
// StackOverflowError is a REAL, common Error every Java developer eventually
// sees -- not from broken hardware or a JVM bug, but from something as
// ordinary as a recursive method with no base case. It extends Error, NOT
// Exception (see ThrowableHierarchyWalkExample for what that chain actually
// looks like) -- a structural signal that this represents a problem with the
// RUNTIME ENVIRONMENT itself (the call stack ran out of space), not a
// recoverable application-level condition.
public class StackOverflowErrorExample {
    public static void main(String[] args) {
        try {
            recurseForever(0);
        } catch (StackOverflowError e) {
            // Technically legal -- Error extends Throwable, and any Throwable
            // CAN be caught. But see "Best Practices": catching an Error is
            // almost never the right response, because the JVM is often in a
            // degraded state by the time one is thrown (the stack was
            // already nearly exhausted). This catch exists here purely to
            // demonstrate that it's possible, not to recommend it.
            System.out.println("Caught: " + e.getClass().getName());
        }
    }

    private static void recurseForever(int depth) {
        // No base case, no termination condition -- every call adds another
        // frame to the call stack (see "Reading a Stack Trace: Propagation
        // Through the Call Chain" in "Introduction to Exceptions" for what a
        // stack frame is) until the JVM has nowhere left to put one.
        recurseForever(depth + 1);
    }
}
```

Expected Output

```
Caught: java.lang.StackOverflowError
```

This example writes `catch (StackOverflowError e)`, and it technically works — but as you'll see in "Best Practices", that's almost never the right move.

> The language permits catching `Error`, but by the time the JVM throws one, it's often already in a degraded state (the call stack may already be nearly exhausted) — so catching an `Error` and trying to continue normally usually just hides the underlying problem instead of solving it.

Exception: Application-Level Problems

Unlike `Error`, `Exception` represents conditions application code can reasonably react to — an invalid number entered by a user, a missing file, division by zero. `Exception` itself splits into two broad branches: `RuntimeException` (and everything beneath it) and everything else that is NOT a `RuntimeException`. That second group is made up of **checked exceptions**, which the compiler forces you to either declare with `throws` or catch; the `RuntimeException`

branch is **unchecked**. The distinction itself — when to use which, and why both exist — is the entire subject of "Checked vs Unchecked Exceptions"; here we're only noting where it sits in the hierarchy.

Catching the RuntimeException Subtree: catch and Polymorphism

A `catch` block doesn't have to target the exact class that was thrown — it can target any ANCESTOR of that class too, because `NumberFormatException` genuinely IS-A `RuntimeException`. This lets you catch several different concrete exception types with a SINGLE `catch` block.

```
// A catch block doesn't have to name the exact exception type that was
// thrown -- it can name any TYPE IN ITS HIERARCHY, and it will still catch
// it, because a NumberFormatException genuinely IS-A RuntimeException (that's
// what "extends" means). This example throws three completely different
// concrete exception types and catches all of them with a SINGLE
// catch (RuntimeException e) block.
public class CatchingBySupertypeExample {
    public static void main(String[] args) {
        attempt(() -> Integer.parseInt("not-a-number"));
        attempt(() -> { int[] values = new int[2]; int x = values[5]; });
        attempt(() -> { int x = 10 / 0; });
    }

    private static void attempt(Runnable riskyOperation) {
        try {
            riskyOperation.run();
        } catch (RuntimeException e) {
            // Whichever concrete type was actually thrown, e.getClass()
            // reveals it -- the catch block itself only cares that it's
            // SOME RuntimeException.
            System.out.println("Caught " + e.getClass().getSimpleName()
                    + " via catch (RuntimeException e)");
        }
    }
}
```

Expected Output

```
Caught NumberFormatException via catch (RuntimeException e)
Caught ArrayIndexOutOfBoundsException via catch (RuntimeException e)
Caught ArithmeticException via catch (RuntimeException e)
```

In this example, `NumberFormatException`, `ArrayIndexOutOfBoundsException`, and `ArithmeticException` — three completely different concrete classes — are all caught by one `catch (RuntimeException e)` block, because all three are subclasses of `RuntimeException`.

When writing multiple `catch` blocks (see "Multiple catch Blocks: Matching in Order"), put the MOST specific type first and the MOST general type last — otherwise the general block makes the more specific one after it unreachable, and the compiler flags this as an error.

Checking the Hierarchy at Runtime with instanceof

The `instanceof` operator asks, at runtime, whether an object is an instance of a given class (or any of its ancestors) — it lets you explicitly perform the same kind of matching a `catch` block does statically.

```
// instanceof asks "is this object a member of this branch of the hierarchy?"
// at runtime -- true for the exact class AND every ancestor. This is useful
// when a single catch block needs to react differently depending on how
// specific the caught exception actually is, without writing a separate
// catch clause for every possible subtype.
public class InstanceofHierarchyCheckExample {
    public static void main(String[] args) {
        describe(new NumberFormatException("example"));
        describe(new IllegalArgumentException("example"));
        describe(new ArithmeticException("example"));
        describe(new Exception("example"));
    }

    private static void describe(Exception exception) {
        // Checked broadest-relevant-to-narrowest is not required here since
        // each branch just prints a label -- but in real code, ordering
        // instanceof checks (or catch clauses) from MOST specific to LEAST
        // specific matters, or a broader check would shadow a narrower one.
        if (exception instanceof NumberFormatException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is a
NumberFormatException");
        } else if (exception instanceof IllegalArgumentException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is an
IllegalArgumentException, but not a NumberFormatException");
        } else if (exception instanceof RuntimeException) {
            System.out.println(exception.getClass().getSimpleName() + " -> is some other
RuntimeException");
        } else {
            System.out.println(exception.getClass().getSimpleName() + " -> is not a
RuntimeException at all");
        }

        // Every exception here also passes this check, since Exception sits
        // above all of them in the hierarchy -- instanceof against a common
        // ancestor is always true for every descendant.
        System.out.println("  instanceof Exception: " + (exception instanceof Exception));
    }
}
```

Expected Output

```
NumberFormatException -> is a NumberFormatException
  instanceof Exception: true
IllegalArgumentException -> is an IllegalArgumentException, but not a NumberFormatException
  instanceof Exception: true
ArithmeticException -> is some other RuntimeException
  instanceof Exception: true
Exception -> is not a RuntimeException at all
  instanceof Exception: true
```

This is especially useful inside a single, broad `catch (Exception e)` block when you need to behave differently depending on what the caught object's actual type turns out to be — though in most cases, separate `catch` blocks achieve the same result more readably.

Best Practices

- Be as SPECIFIC as possible when writing a `catch` block — catch the type you actually expect (`catch (NumberFormatException e)`) rather than defaulting to `catch (RuntimeException e)` ; a broad type only makes sense when you genuinely want to treat several types the same way.
- Avoid catching `Error` (or `Throwable` directly) — the JVM is almost always already in an unrecoverable state, and catching it just hides the problem.
- Use your IDE's "go to superclass" feature or the `getSuperclass()` chain (as shown in this lesson's first example) to check the hierarchy instead of trying to memorize it.
- When writing multiple `catch` blocks, put the most specific type first — the compiler will already reject the wrong order, but building the habit correctly from the start improves readability.

Common Mistakes

- Writing `catch (Exception e)` and doing nothing meaningful inside it — this silently swallows every checked AND unchecked exception (though NOT `Error` , since `Exception` doesn't cover it), making debugging nearly impossible.
- Writing `catch (Throwable t)` — this also covers `Error` , and is almost never the right choice.
- Catching a `StackOverflowError` and trying to continue normal execution — with the stack already nearly exhausted, this can lead to new, less predictable failures.
- Confusing the hierarchy and assuming `RuntimeException` is a SEPARATE branch from `Exception` — in reality, `RuntimeException` is a subclass of `Exception` itself, not a sibling of it.

Summary, Cheat Sheet, and Glossary

Summary

- Every exception and error descends from `Throwable`, which has two branches: `Error` and `Exception`.
- `Error` represents unrecoverable, JVM-level conditions (like `StackOverflowError`) — it normally shouldn't be caught.
- `Exception` represents conditions application code can react to; `RuntimeException` is a subclass of it.
- `catch` can target any ancestor of the thrown type, not just the exact type (polymorphic catching).
- `instanceof` lets you query an object's place in the hierarchy at runtime.

Cheat Sheet

```
// Throwable
//   ├── Error (StackOverflowError, OutOfMemoryError, ...)
//   └── Exception
//         ├── RuntimeException (NumberFormatException, ArithmeticException, ...)
//         └── (checked exceptions -- see Checked vs Unchecked Exceptions)

try {
    riskyOperation();
} catch (RuntimeException e) {   // covers every specific subtype
    // ...
}

if (something instanceof RuntimeException) {
    // ...
}
```

Glossary

- **Throwable**: the common ancestor of everything that can be caught or thrown.
- **Error**: a `Throwable` subclass representing serious, usually unrecoverable, JVM-level conditions.
- **Hierarchy**: the inheritance tree among classes; here, `Throwable` is the root, with `Error`/`Exception` as its branches.
- **Polymorphic catching**: a `catch` block's ability to catch not just the exact matching type, but any subclass of it too.
- **instanceof**: an operator that checks at runtime whether an object is an instance of a given class (or one of its ancestors).