## Exceptions

- Used to signal that something went wrong:
  - `throw new EOFException(“Uneven number of ints”);`
- Can be caught by exception handler
  - Recovers from error
  - Or exits gracefully

## A Checkered Past

- Java has two sorts of exceptions

- Checked exceptions: compiler makes sure that calling code doesn't ignore the problem if it occurs.
  - Used for expected problems

- Unchecked exceptions: compiler lets us ignore these if we want
  - Used for fatal or avoidable problems
  - Are subclasses of *RunTimeException* or *Error*

## Hierarchy of Exception Classes

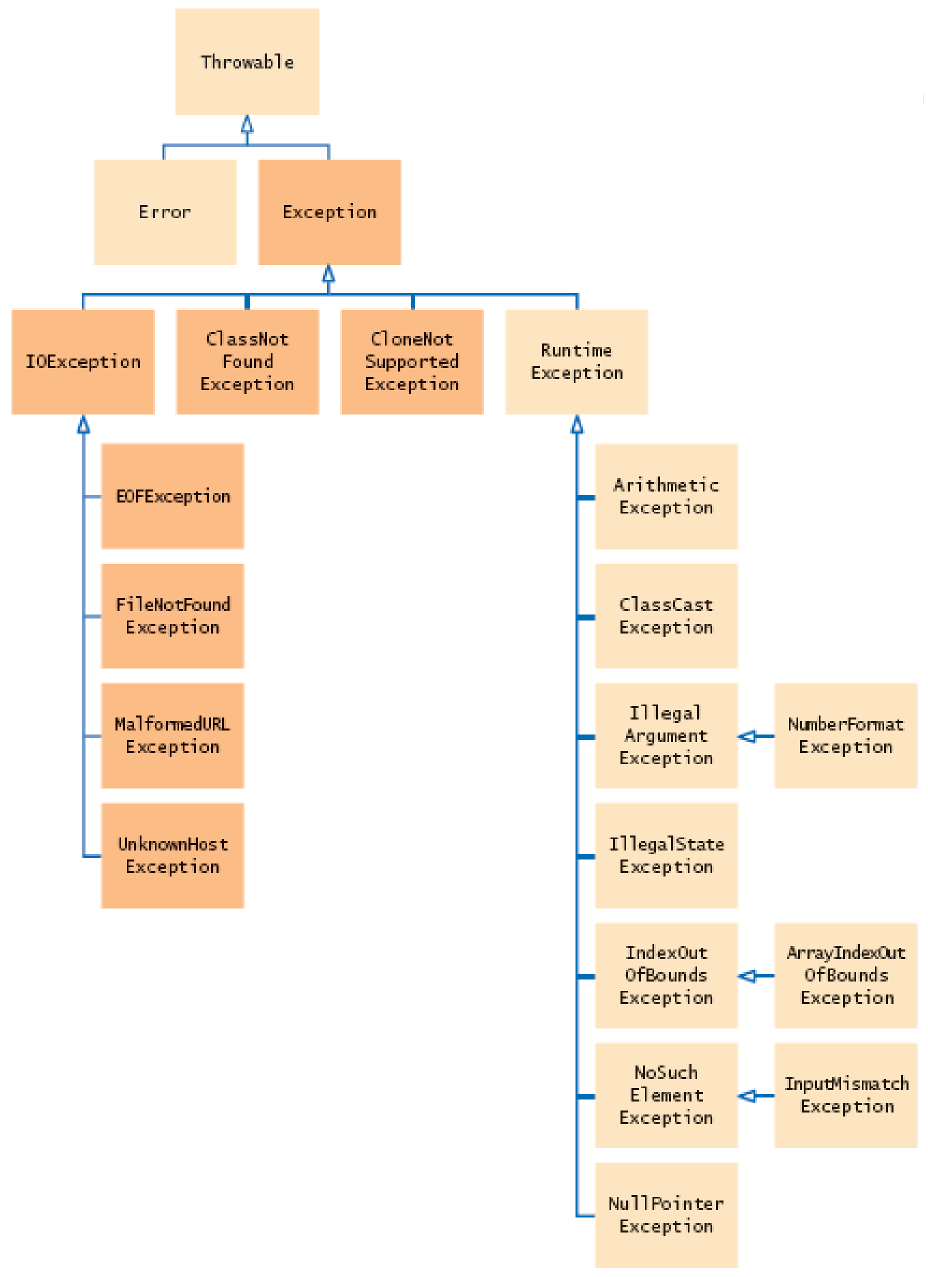

**Figure 1** The Hierarchy of Exception Classes

# A Tale of Two (and a half) Choices

- Dealing with checked exceptions
  - Can propagate the exception
    - Just declare that our method will pass any exceptions along
    - *`public void loadGameState() throws IOException`*
    - Used when our code isn't able to rectify the problem
  - Can handle the exception
    - Used when our code can rectify the problem
  - Can do both
    - Do what we can to handle the exception, and then throw the same (or a different) exception

## Handling Exceptions

- Use try–catch statement:

  -
    ```
    try {
        // potentially "exceptional" code
    } catch (ExceptionType var) {
        // handle exception
    }
    ```

Can repeat this part for as many different exception types as you need.

- Related, try–finally for clean up:

  -
    ```
    try {
        // code that requires "clean up"
    } finally {
        // runs even if exception occurred
    }
    ```

# An example of try/catch

```
@Test
public void testBigRationalBigIntegerBigInteger() {
    try {
        // Test 1: try to construct a BigRational whose denominator is zero.
        new BigRational(new BigInteger("7"), new BigInteger("0"));
        fail("Constructor did not throw ArithmeticException when denominator was zero");
    } catch (ArithmeticException exception) {
        // Test 1 succeeded if it gets here
    } catch (Exception exception) {
        exception.printStackTrace();
        fail("Constructor threw a " + exception.toString()
                + " when it should have thrown an ArithmeticException");
    }

    try {
        // Test 2: construct a BigRational whose denominator is NOT zero.
        new BigRational(new BigInteger("0"), new BigInteger("7"));
    } catch (ArithmeticException exception) {
        fail("Constructor threw ArithmeticException when denominator was NOT zero");
    } catch (Exception exception) {
        exception.printStackTrace();
        fail("Constructor threw a " + exception.toString()
                + " when it should not have thrown any Exception");
    }
    // Test 2 succeeded if it gets here
}
```

# Exceptions – Summary

- Can be thrown to signal that something went wrong:
  - `throw new EOFException("Uneven number of ints");`
- Can be propagated to the calling method:
  - `public void loadGameState() throws IOException`
- Can be caught by exception handler
  - Recovers from error
  - Or exits gracefully

    -
      ```
      try {
            // potentially "exceptional" code
      } catch (ExceptionType var) {
            // handle exception
      }
      ```