Exceptions in Java

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Errors

- Errors do occur during program execution.
  - Problems opening a file, dividing by zero, accessing an out-of-bounds array element, hardware errors, and many more.

- The question becomes: What do we do when an error occurs?
  - **How** is the error handled?
  - **Where** is it handled?
  - Should the program **terminate**?
  - Can the program **recover** from the error? Should it?

- Java and many other contemporary programming languages use exceptions to provide **error-handling capabilities** for programs.

When Things Go Wrong

- When something ``goes wrong'' at a line of code:
  - An **exception object** gets created.
  - Flow of control changes to some place in your code that can handl e the exception.

- Note: usually changes in flow of control are **clearly marked** by J ava keywords
  - return, if/else, while, for, switch, call to a method, break, continue
  - Exceptions are different: they ``jump'' to somewhere else, which is often not so obvious.

ArithmeticException Example

```
public class Zero {

  public static void main(String[] args) {
    int numerator = 10;
    int denominator = 0;
    System.out.println(numerator/denominator);
    System.out.println("We never get to this statement.");
  }

}
```

After encountering the division by zero, the program **terminates** with

```
Exception in thread "main" java.lang.ArithmeticException:
 / by zero at Zero.main(Zero.java:6)
```

ArithmeticException Example

```
public static void main(String[] args) {
    int numerator = 10; int denominator = 0;
    try {
      System.out.println(numerator/denominator);
      System.out.println("We never get to this statement.");
    }
    catch (ArithmeticException e) {
      System.out.println("Division by zero occurred.");
    }
    System.out.println("After catch statement.");
}
```

- When encountering the division by zero in the try block, an exception occurs.
- The exception is handled in the catch block.
- Execution then continues with the next statement after the catch block.

Program output:

```
Division by zero occurred.
After catch statement.
```

General Format

```
try {
 statements;
}
catch (ExceptionType1 name){
 statements;
}
catch (ExceptionType2 name){
 statements;
}
finally {
 statements;
}
```

Handling Exceptions

- **try** block encloses a block of statements where an exception might be thrown.

- **catch** blocks are associated with a try statement. Contain code to handle a particular type of exception. The statements are executed if an exception of that type occurs within the try block.

- **finally** block is associated with a try statement. Contains statements that are executed **regardless** of whether or not an error occurs within the try block. Even if the try and catch block have a return statement in them, the ``finally'' block will still run.

Throwing (Raising) of Exceptions

- If an error situation brings forward an exception, we say an exception is **thrown**. Alternative term: an exception is **raised**.
- Exceptions may be thrown by the JVM in response to an error situation during program execution:
  - Example:
    - `x = y / 0;`
    - At run-time, when encountering the program's attempt to divide by zero, the JVM will throw an `ArithmeticException`.
- Exceptions may be thrown **programmatically**, using a **`throw`** statement:
  - Example:

```
if (currentToken.kind != Token.the)
    throw (new SyntaxError("Article expected!"));
```

  - In response to the `throw` statement, the JVM will instantiate the `SyntaxError` exception object and throw it.

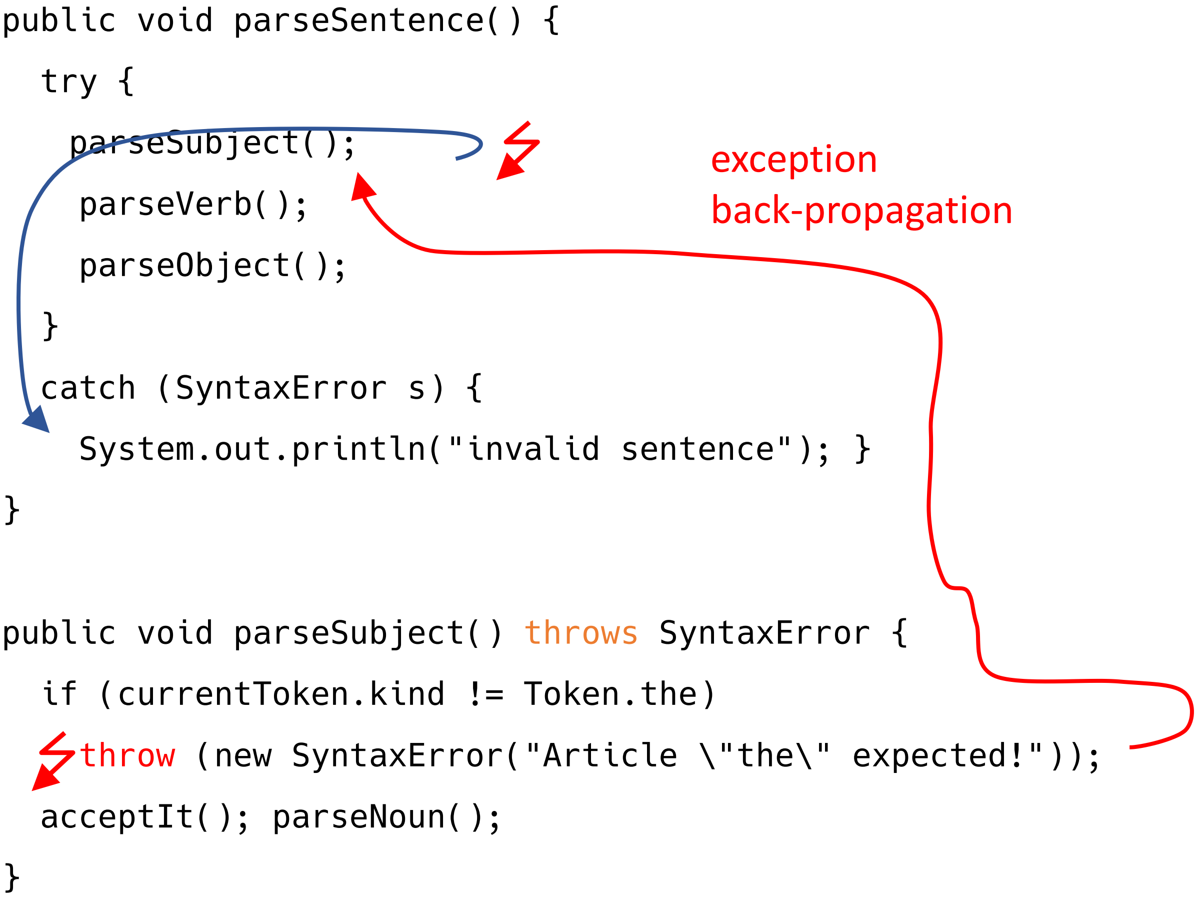
Propagation of Exceptions

```
public void parseSentence() {
  try {
    parseSubject();
    parseVerb();
    parseObject();
  }
  catch (SyntaxError s) {
    System.out.println("invalid sentence"); }
}

public void parseSubject() throws SyntaxError {
  if (currentToken.kind != Token.the)
    throw (new SyntaxError("Article \"the\" expected!"));
  acceptIt(); parseNoun();
}
```

```
Subject ::= "the" Noun
```

Propagation of Exceptions (cont.)



```
public void parseSentence( ) {
  try {
    parseSubject( );
    parseVerb( );
    parseObject( );
  }
  catch (SyntaxError s) {
    System.out.println("invalid sentence"); }
}


public void parseSubject( ) throws SyntaxError {
  if (currentToken.kind != Token.the)
    throw (new SyntaxError("Article \"the\" expected!"));
  acceptIt( ); parseNoun( );
}
```

- If a method has no catch-block for an exception type, then this method is terminated and the exception is propagated back to the caller of the method.
  - As if the caller itself had raised the exception!
- If the caller has no catch block, then the caller is terminated and the exception is propagated to the caller of the caller…
- If no handler is available at the global scope, the program is terminated (see Slide #4).

Propagation of Exceptions

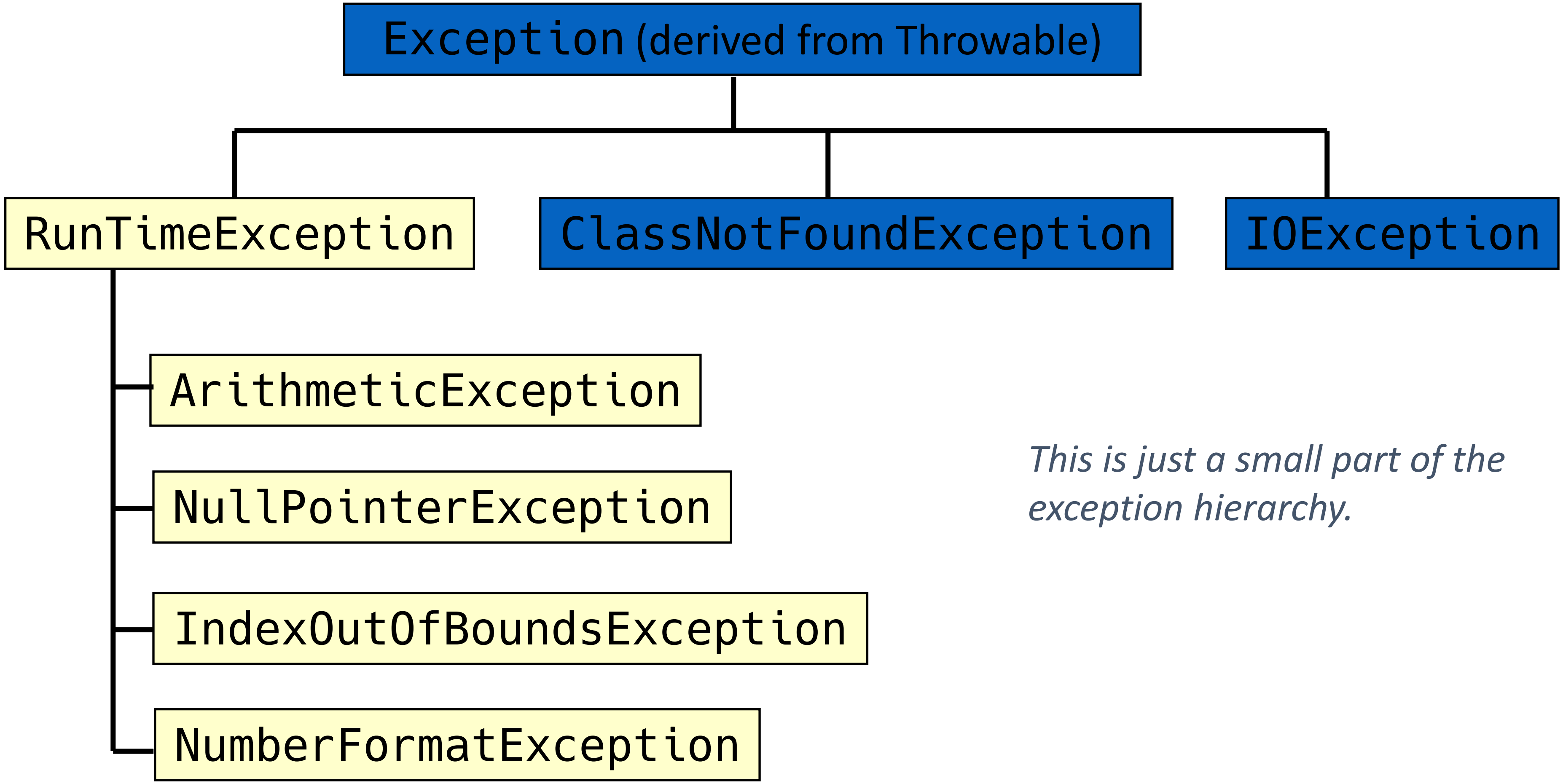
- Java methods must list the exceptions that may occur. Exceptions are listed in the method's throws clause.

- Compiler checks that **exceptions a method may throw** are
  - either listed in the throws clause, or
  - handled by the method (in a catch block).

- Exception classes `Error` and `RuntimeException` and their descendants are **unchecked exceptions** that need not be listed, and that are not verified by the compiler.

Exception Class Hierarchy

- Java exceptions are objects.
- Java has a predefined set of exceptions for errors that can occur during execution.
- Exception class hierarchy rooted at ``Throwable''; User-defined exceptions can be derived.



This is just a small part of the exception hierarchy.

Why is this useful?

- C, Fortran77 and Pascal do not support exception handling.
- Programmers then have to use ``special" return values or global error status variables:

  ```
  int foo(FILE *f) {
    ...
    if (feof(f)) return -1; // return error code -1 on EOF
    return value;           // return normal value
  }
  ```

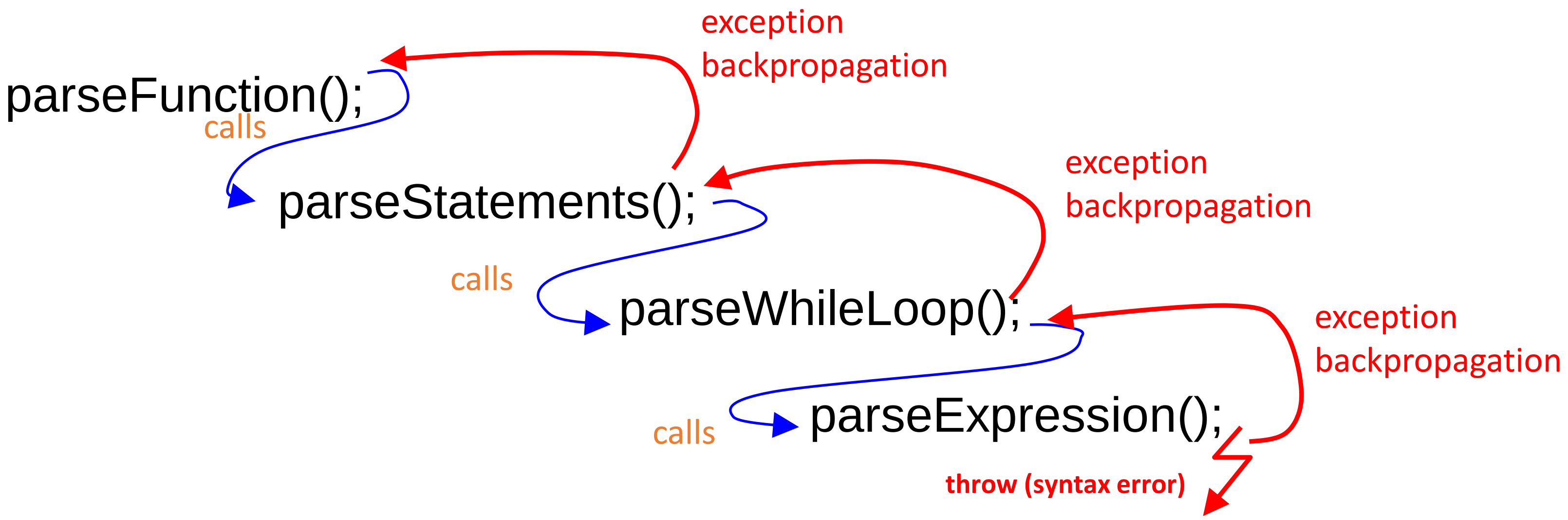
  - Every function call must be checked for return values indicating an error:

    ```
    val=foo(f);
    if(val<0) {...}
    ```

  - Tedious, results in **unreadable code**.
    - Error-handling code interspersed with the 'real' code
  - **Forgetting to check the return value** potentially leads to disaster! (Overlooking of an error condition.)

Why is this useful in a RD parser ?

- We can conveniently propagate an error condition from deep down in the call graph to a function higher up in the call graph:
  - In the below example we assume that parseStatements() and parseWhileLoop() do not have a handler for syntax error exceptions:



- parseFunction();

parseStatements();

parseWhileLoop();

parseExpression();

- Error-Recovery in parseFunction can then skip tokens past the end of the function and continue parsing the next function in the program.