Programming in Java

Exceptions and Assertions

Hua Huang, Ph.D.

Spring 2019

Objectives

- Define **exceptions**
- Use **try**, **catch**, and **finally** statements
- Describe **exception categories**
- Identify **common exceptions**
- Develop programs to **handle your own exceptions**
- Use **assertions**
- Distinguish appropriate and inappropriate uses of assertions
- Enable assertions at runtime
- **Log** facility

Relevance

- In most programming languages, **how do you resolve runtime errors**?
- If you make assumptions about the way your code works, and those assumptions are wrong, what might happen?
- Is it always necessary or desirable to expend CPU power testing assertions in production programs?

Exceptions and Assertions

- An **exception** is an **event**, which occurs during the execution of a program, that **disrupts the normal flow of the program's** instructions.

- Exceptions **handle unexpected situations** – Illegal argument, network failure, or file not found

- **Assertions** document and test **programming assumptions** – This can never be negative here

- **Assertion tests can be removed** entirely from code at runtime, so the code is not slowed down at all.

Exceptions

- Conditions that can **readily occur** in a correct program are **checked exceptions**.

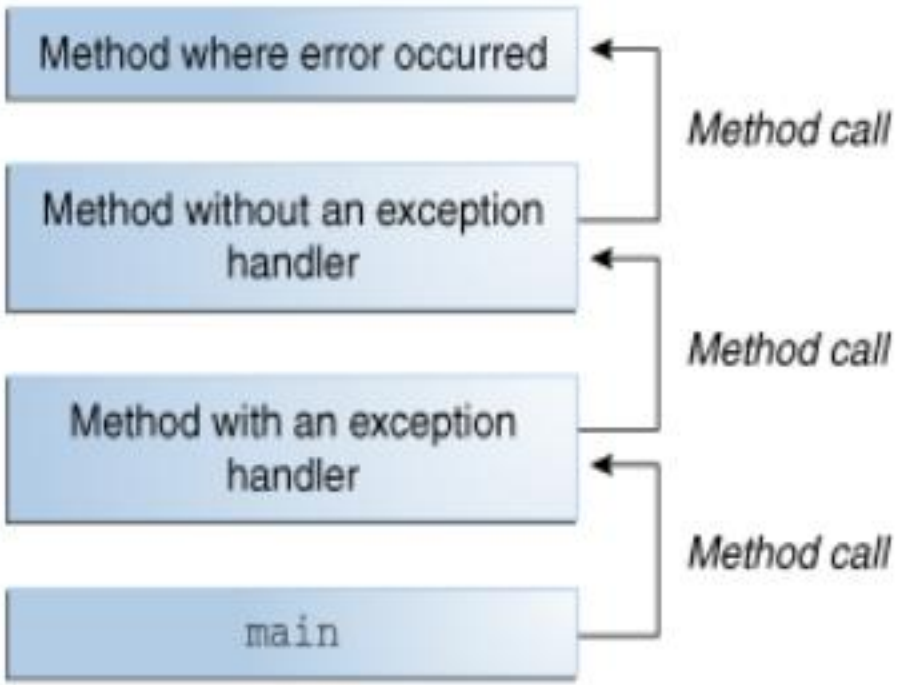
  These are represented by the `Exception` class.

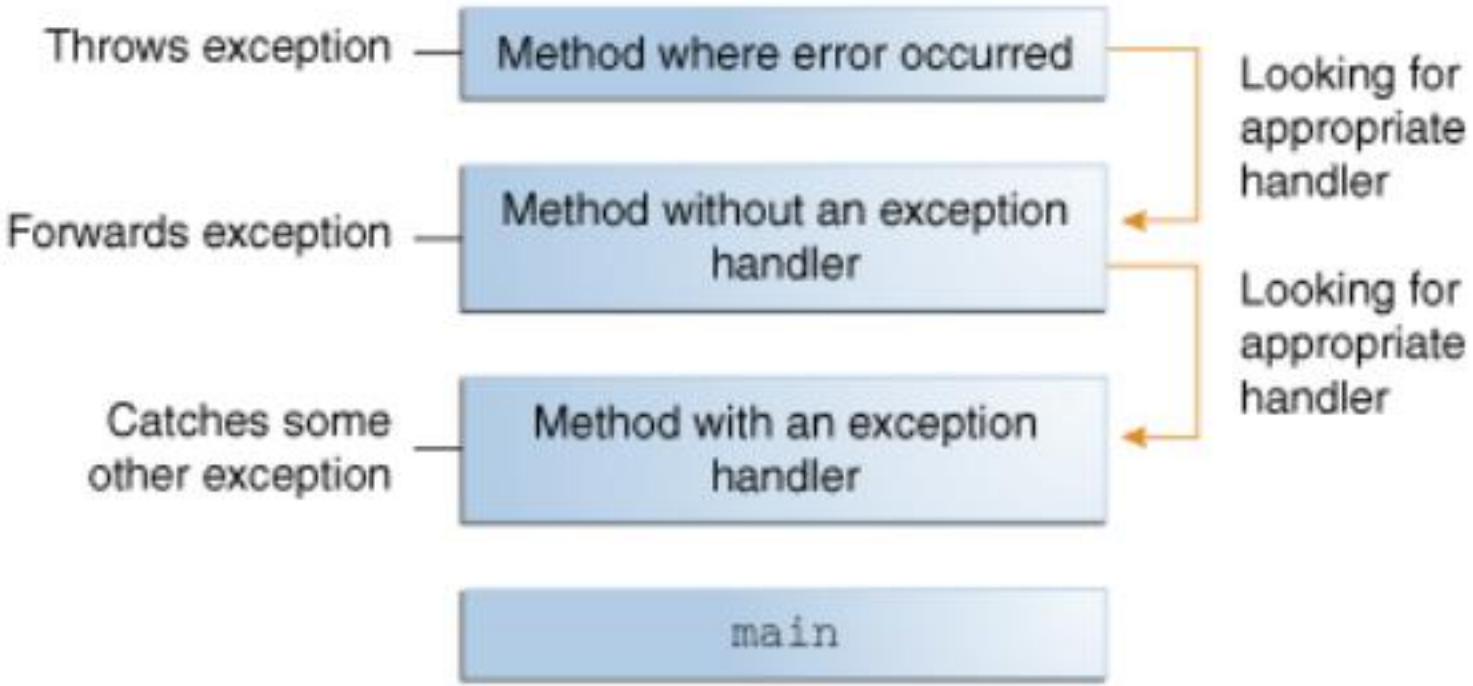
- **Severe problems** that normally are treated as **fatal** or situations that probably reflect **program bugs** are **unchecked exceptions**.

  Fatal situations are represented by the `Error` class. **Probable bugs** are represented by the `RuntimeException` class.

- The **API documentation** shows **checked exceptions** that can be thrown from a method.

Exceptions(Cont.)





Exception Example

```
01  public class AddArguments {
02    public static void main(String args[]) {
03      int sum = 0;
04      for ( String arg : args ) {
05        sum += Integer.parseInt(arg);
06      }
07      System.out.println("Sum = " + sum);
08    }
09  }
10
```

```
java AddArguments 1 2 3 4
Sum = 10
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at AddArguments.main(AddArguments.java:5)
```

The try-catch Statement

```
1    public class AddArguments2 {
2      public static void main(String args[]) {
3        try {
4          int sum = 0;
5          for ( String arg : args ) {
6            sum += Integer.parseInt(arg);
7          }
8          System.out.println("Sum = " + sum);
9        } catch (NumberFormatException nfe) {
10       System.err.println("One of the command-line "
11         + "arguments is not an integer.");
12       }
13     }
14   }
```

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

The try-catch Statement(Cont.)

```
01  public class AddArguments3 {
02    public static void main(String args[]) {
03      int sum = 0;
04      for (String arg : args) {
05        try {
06          sum += Integer.parseInt(arg);
07        } catch (NumberFormatException nfe) {
08          System.err.println("[" + arg + "] is not an "
09            + "integer and will not be included in the sum.");
10        }
11      }
12      System.out.println("Sum = " + sum);
13    }
14  }
```

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

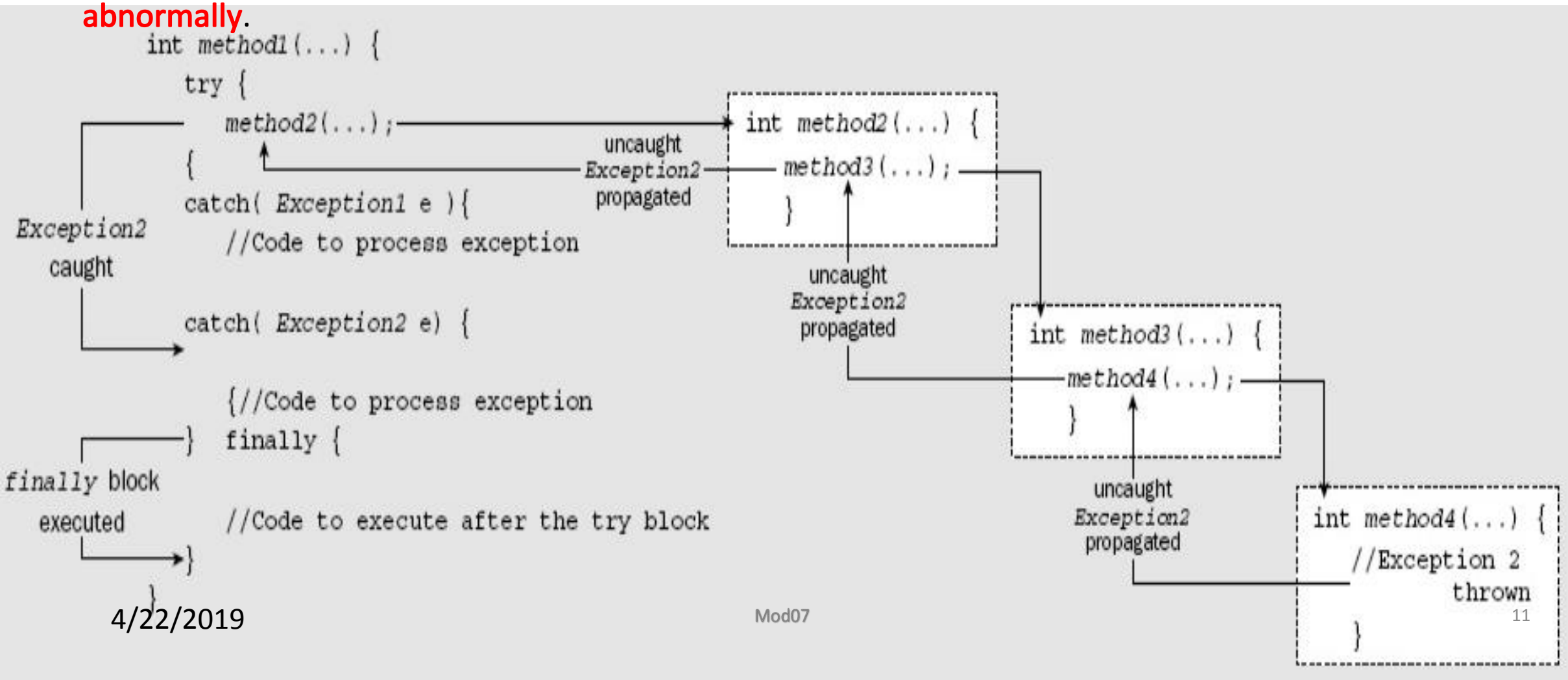
The `try-catch` Statement(Cont.)

- A `try-catch` statement can use multiple catch clauses:

```
try{
   // code that might throw one or more exceptions
} catch (MyException e1) {
   // code to execute if a MyException exception is thrown
} catch (MyOtherException e2) {
   // code to execute if a MyOtherException exception is
thrown
} catch (Exception e3) {
   // code to execute if any other exception is thrown
}
```

Call Stack Mechanism

- If an exception is **`not handled`** in the current `try-catch` block, it is **thrown to the caller** of that method.
- If the exception **gets back to the main method** and is **not handled** there, the program is **terminated abnormally**.



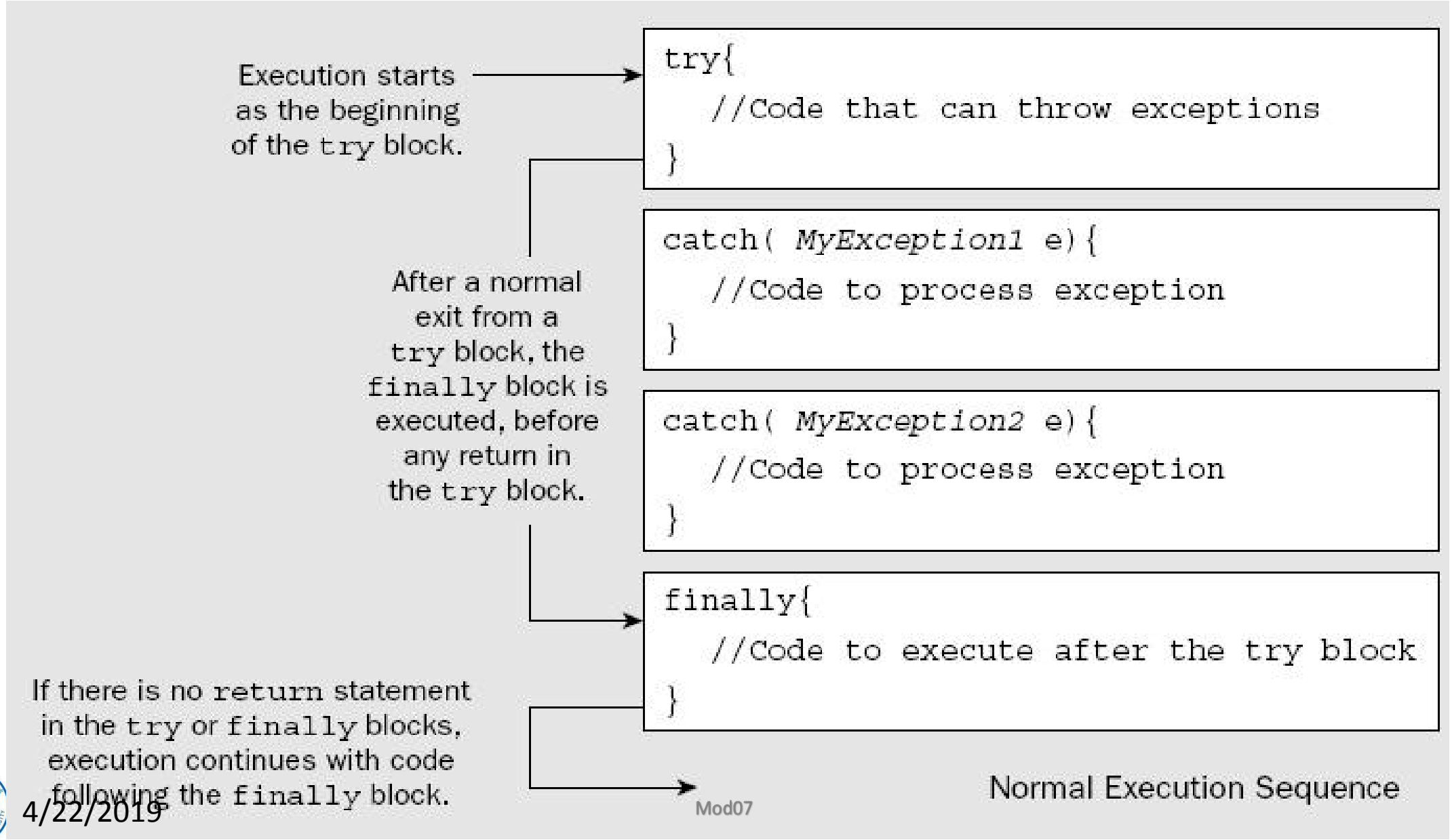
The `finally` Clause

- The `finally` clause defines a block of code that **always** executes.

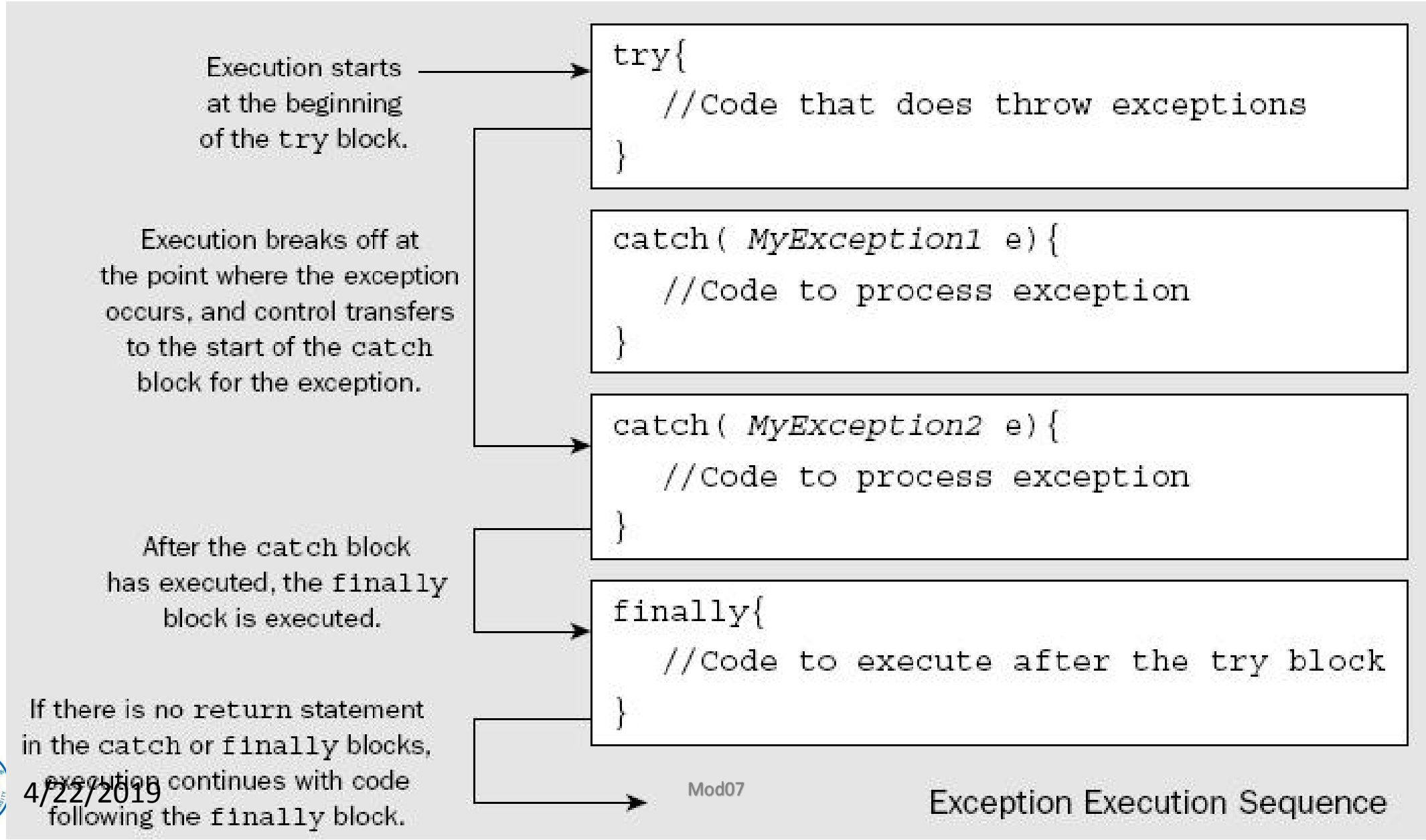
```
1   try {
2      startFaucet();
3      waterLawn();
4   } catch (BrokenPipeException e) {
5      logProblem(e);
6   }
7   finally {
8      stopFaucet();
9   }
```

- The catch clause is optional here!.

The `finally` Clause(Cont.)

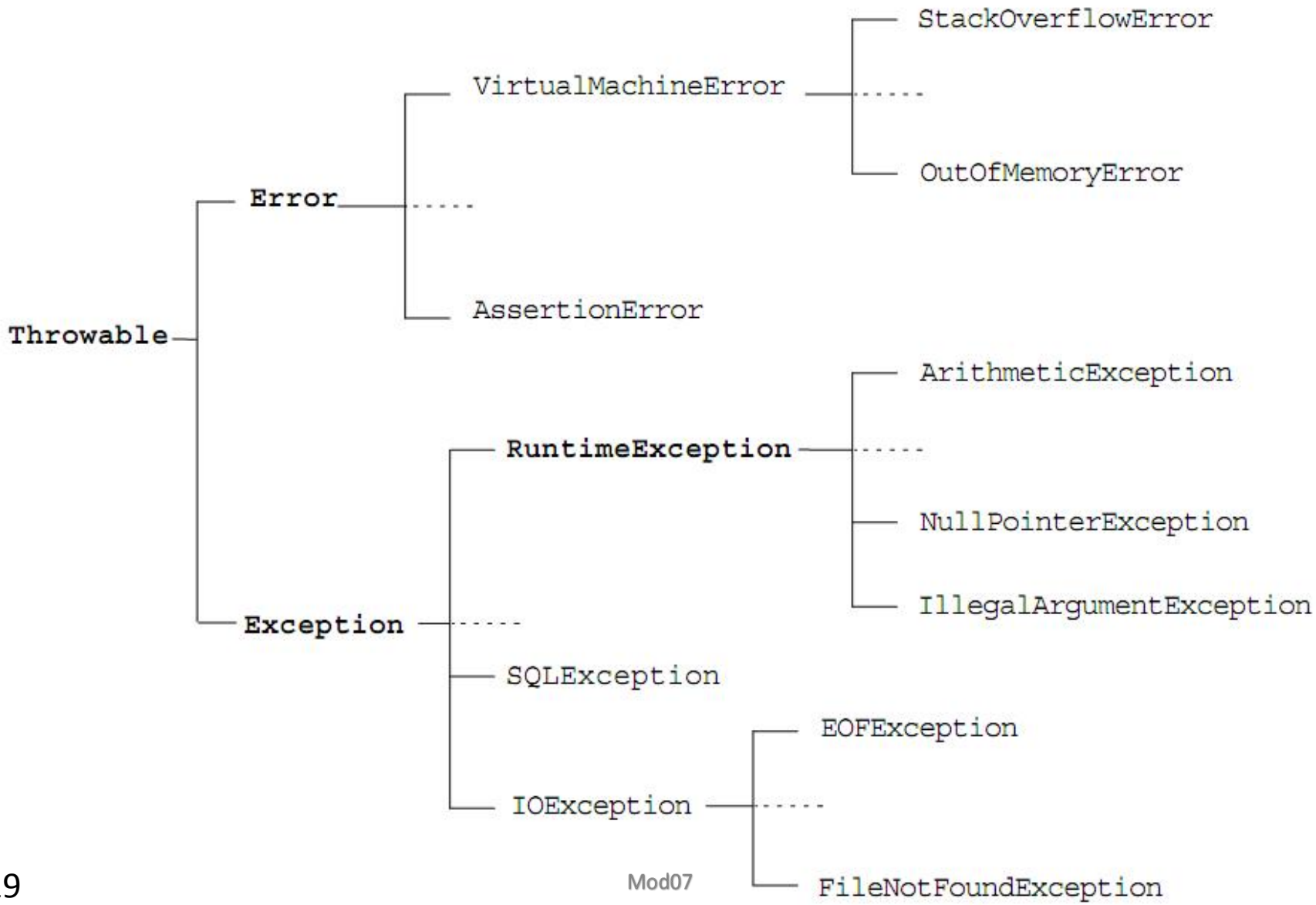


The `finally` Clause(Cont.)



Exception Execution Sequence

Exception Categories



Common Exceptions

- NullPointerException
- FileNotFoundException
- NumberFormatException
- ArithmeticException
- SecurityException
- …

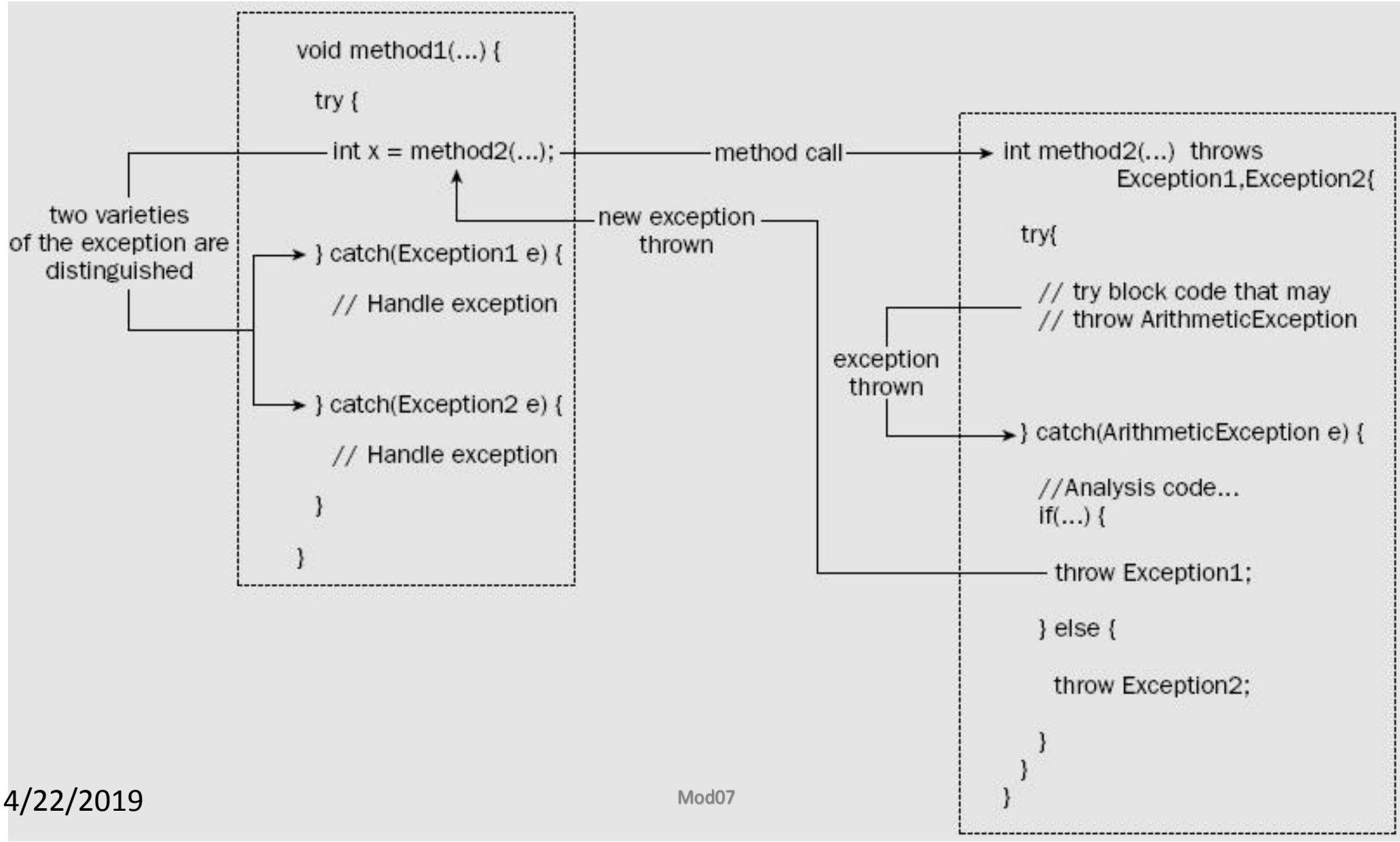
The Handle or Declare Rule

- Use the handle or declare rule as follows:
  - Handle the exception by using the **`try-catch-finally`** block.
  - **Declare** that the code **causes** an exception by using the `throws` clause.

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

- Other Principles
  - You **do not need to** declare **runtime exceptions** or **errors**.
  - You **can choose to** handle **runtime exceptions**.

Exception Handling



Method Overriding and Exceptions

- The **overriding method can throw**:
  - No exceptions
  - One or more of the **exceptions thrown by the overridden method**
  - One or more **subclasses of the exceptions** **thrown by the overridden method**
- The overriding method cannot throw:
  - **Additional exceptions** not thrown by the overridden method
  - **Superclasses of the exceptions** thrown by the overridden method

Method Overriding and Exceptions

```
01  public class TestA {
02    public void methodA() throws IOException {
03      // do some file manipulation
04    }
05  }
06
07  public class TestB1 extends TestA {
08    public void methodA() throws EOFException { //??
09      // do some file manipulation
10    }
11  }
12
13  public class TestB2 extends TestA {
14    public void methodA() throws Exception {  //??
15      // do some file manipulation
16    }
17  }
```

Creating Your Own Exceptions

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String message,int port) {
    super(message);
    this.port = port;
  }
  public int getPort() {
    return port;
  }
}
```

- Use the `getMessage` method, **inherited** from the `Exception` class, to get the reason for which the exception was made.

Handling a User-Defined Exception

- A method can throw a **user-defined**, **checked** exception:

```
01   public void connectMe(String serverName) throws
                ServerTimedOutException {
02      boolean successful;
03      int portToConnect = 80;
04
05      successful = open(serverName, portToConnect);
06
07      if ( ! successful ) {
08         throw new ServerTimedOutException(
                      "Could not connect", portToConnect);
09      }
10   }
```

Handling a User-Defined Exception

- Another method can use a `try-catch` block to capture user-defined exceptions:

```
01  public void findServer() {
02     try {
03        connectMe(defaultServer);
04     } catch (ServerTimedOutException e) {
05        System.out.println("Server timed out, trying alternative");
06        try {
07           connectMe(alternativeServer);
08        } catch (ServerTimedOutException e1) {
09           System.out.println("Error: " + e1.getMessage() +
10              " connecting to port " + e1.getPort());
11        }
12     }
13  }
```

Assertions

- Syntax of an assertion is:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

- If `<boolean_expression>` evaluates `false`, then an **`AssertionError`** is thrown.
- The second argument is converted to a string and used as descriptive text in the **`AssertionError`** message.

Recommended Uses of Assertions

- Use assertions to **document and verify the assumptions and internal logic** of a single method:
  - Internal invariants
  - Control flow invariants
  - Preconditions, Postconditions, and Class Invariants
- **Inappropriate** Uses of Assertions
  - **Do not** use assertions to **check the parameters** of a public method.
  - **Do not** use methods in the assertion check that can **cause side-effects**.

https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

Internal Invariants

- The problem is:

```
1   if (i % 3 == 0) {
2       ...
3   } else if (i % 3 == 1) {
4       ...
5   } else { // We know (i % 3 == 2)
6       ...
7   }
```

- The solution is:

```
1   if (i % 3 == 0) {
2     ...
3   } else if (i % 3 == 1) {
4     ...
5   } else {
6   assert i % 3 == 2 : i;
7     ...
8   }
```

Control Flow Invariants

- place an assertion at any location you assume will not be reached, suppose you have a method that looks like this:

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    // Execution should never reach this point!!!
}
```

Replace the final comment so that the code now reads:

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    assert false; // Execution should never reach this point!
}
```

Preconditions Invariants

- By convention, **preconditions** on public methods are enforced by explicit checks that throw particular, specified exceptions. For example:

```
public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);
  setRefreshInterval(1000/rate);
}
```

Do not use assertions to check the parameters of a public method. Use an assertion to test a nonpublic method's precondition

```
 private void setRefreshInterval(int interval) {
   // Confirm adherence to precondition in nonpublic method
   assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE :
   interval;
   ... // Set the refresh interval
}
```

Postconditions Invariants

- You can test **postcondition** with assertions in both **public** and **nonpublic** methods.

```
public BigInteger modInverse(BigInteger m) {
  if (m.signum <= 0)
    throw new ArithmeticException("Modulus not positive: " + m);
  ... // Do the computation
  assert this.multiply(result).mod(m).equals(ONE) : this;
    return result;
}
```

Class Invariants

- A **class invariant** is a type of internal invariant, can specify the **relationships among multiple attributes**, and should be true before and after any method completes.
- Example, suppose implementing a balanced tree of some sort. A class invariant might be that **the tree is balanced and properly ordered**.
- it might be appropriate to implement a private method that checked that the tree was indeed balanced.

```
// Returns true if this tree is properly balanced
 private boolean balanced() {
  ...
}
```

- The constraint that should be true before and after any method completes, each public method and constructor should contain the following line immediately prior to its return:

```
assert balanced();
```

Controlling Runtime Evaluation of Assertions

- If assertion checking is **disabled**, the code runs as fast as if the check was never there.
- Assertion checks are **disabled by default**. Enable assertions with the following commands:

```
java -enableassertions MyProgram
```

or:

```
java -ea MyProgram
```

- Assertion checking can be controlled on **class**, **package**, and **package hierarchy** bases, see: docs/guide/language/assert.html

Log

- Package java.util.logging
- Create or Get the Logger:

```
public static Logger getLogger(String name)
```

- Log level
  - java.util.logging.Level
  - **SEVERE** 最高, WARNING, INFO, CONFIG, FINE, FINER, **FINEST** 最低
- Logger.GLOBAL_LOGGER_NAME vs. System.out
  - TestLogger.java

Log(Cont.)

- Example code snippet :

```
try {
  Handler handler = new FileHandler("OutFile.log");
  Logger.getLogger("").addHandler(handler);
} catch (IOException e) {
  Logger logger = Logger.getLogger("package.name");
  StackTraceElement elements[] = e.getStackTrace();
  for (int i = 0, n = elements.length; i < n; i++) {
    logger.log(Level.WARNING, elements[i].getMethodName());
  }
}
```

Questions or Comments?