EE3206 Java Programming and Applications

Lecture 6 Exceptions and Assertions

Intended Learning Outcomes

- To know what is exception and how to handle exceptions
- To distinguish checked and unchecked exceptions
- To declare, throw and catch exceptions
- To use try-catch-finally clause
- To understand the advantages of using exceptions
- To apply assertions to ensure program correctness

Three Kinds of Programming Errors

- **Syntax errors** arise because the rules of the language have not been followed. They are detected by the compiler.
- **Runtime errors** occur while the program is running if the environment detects an operation that is impossible to carry out.
- **Logic errors** occur when a program doesn't perform the way it was intended to.

- The ideal time to catch an error is at compile time, because badly formed code will not be run. However, only syntax errors can be detected at compile time. The rest of the problems must be handled at runtime through a formal mechanism - **Exception Handling**.
  - An **exception** is an event object, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- Generally speaking, reading, writing, and debugging becomes easier with exceptions handling

Traditional Error Handling

- Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
- In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
  open the file;
  determine its size;
  allocate that much memory;
  read the file into memory;
  close the file;
}
```

- This function seems simple enough, but it ignores all the following potential errors.
  - What happens if the file can't be opened?
  - What happens if the length of the file can't be determined?
  - What happens if enough memory can't be allocated?
  - What happens if the read fails?
  - What happens if the file can't be closed?

Separating Error-Handling Code from “Regular” Code

```
readFile {
  open the file;
  determine its size;
  allocate that much memory;
  read the file into memory;
  close the file;
}
```

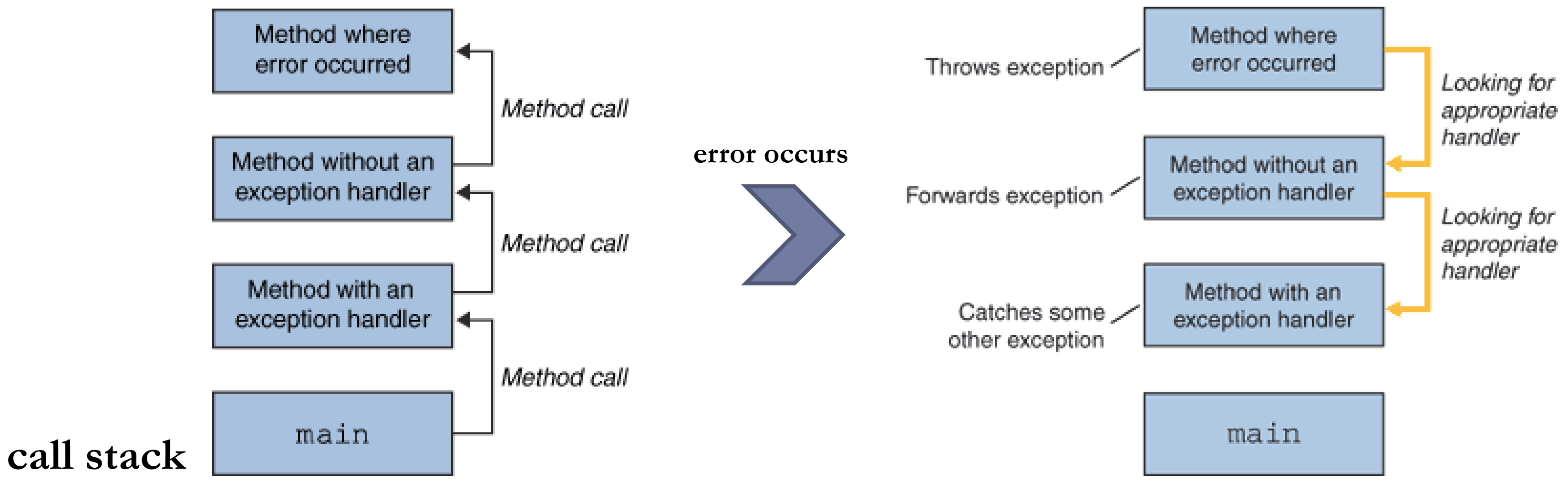
```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;

    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exceptions enable you to express the main flow of your code and then to deal with the exceptional cases elsewhere.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exception Handling – How it works?

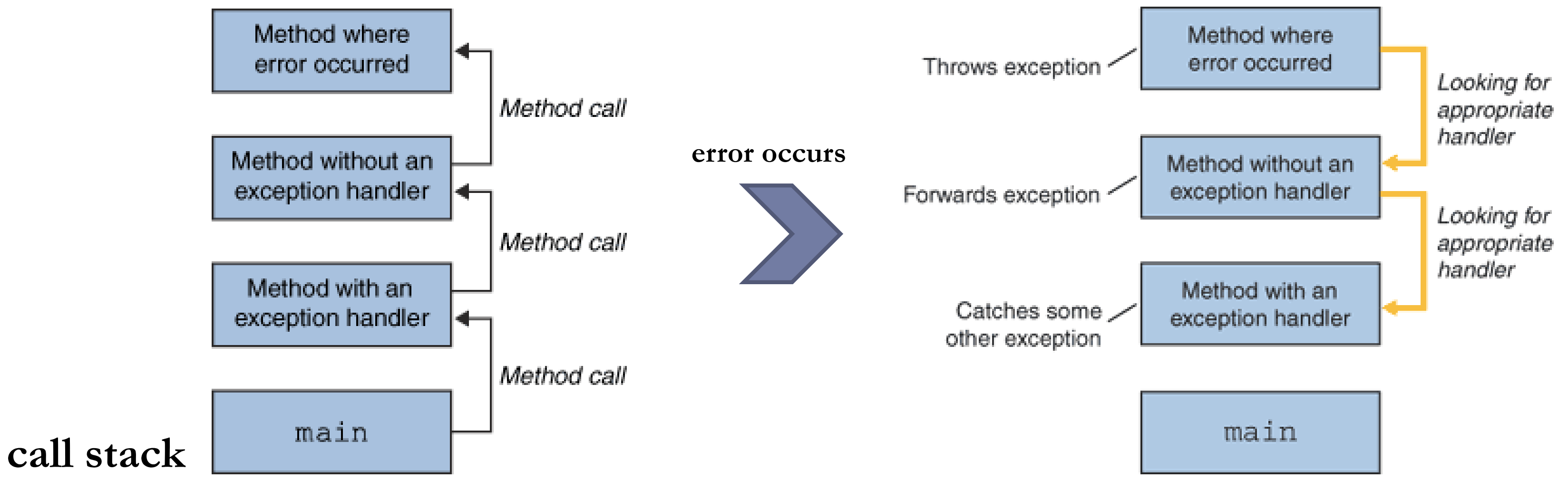
- When an error occurs within a method, the method creates an object and hands it off to the runtime system. This object is called an *exception object*, which contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.
- After a method throws an exception, the runtime system attempts to find something to handle it. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.



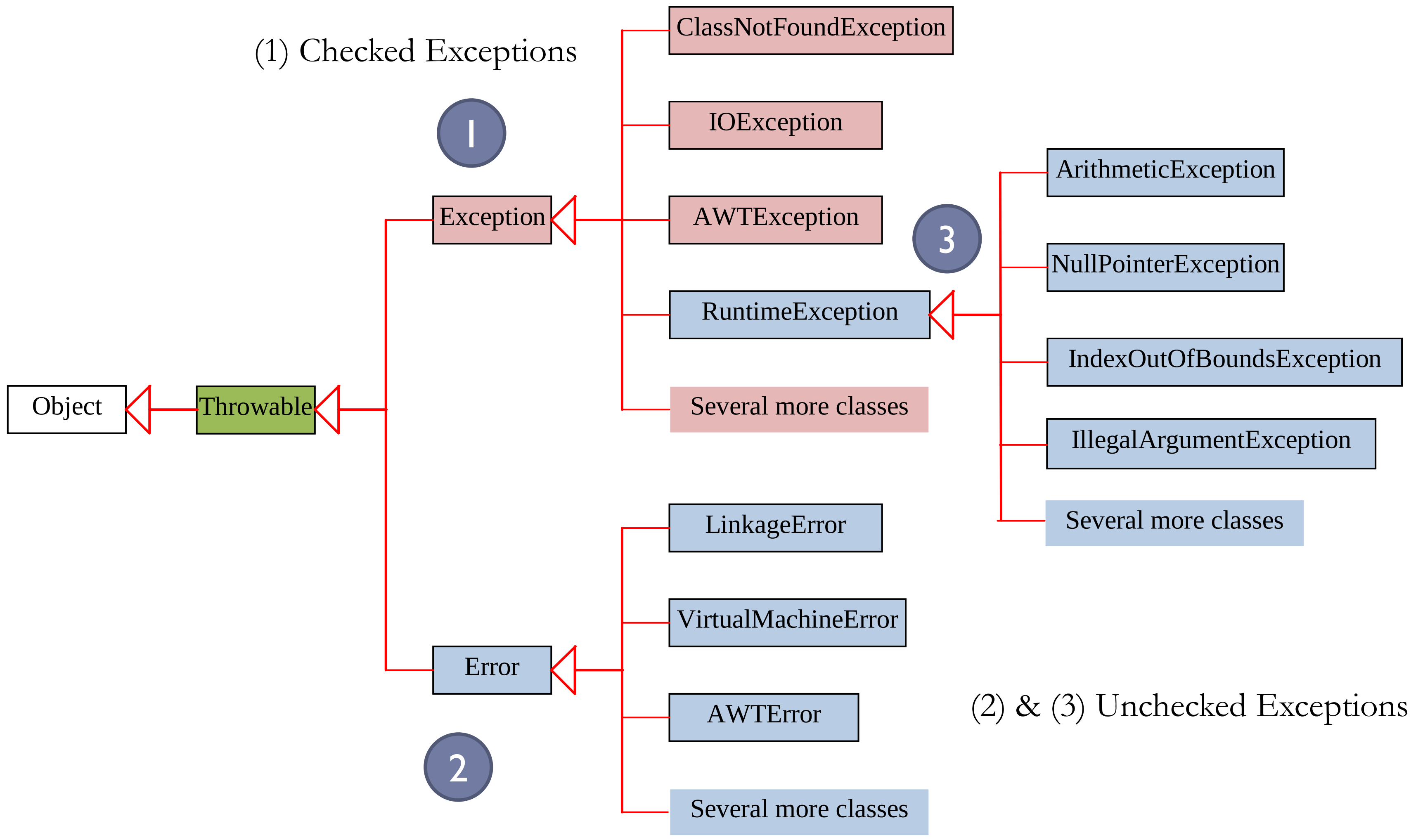
Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main
call stack
error occurs
Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Exception Handling – How it works?

- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler.
- An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. The exception handler chosen is said to *catch the exception*.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the program) terminates.


Method where error occurred
Method without an exception handler
Method with an exception handler
main
Method call
Method call
Method call
call stack
error occurs
Throws exception
Forwards exception
Catches some other exception
Method where error occurred
Method without an exception handler
Method with an exception handler
main
Looking for appropriate handler
Looking for appropriate handler

Checked and Unchecked Exceptions



Three Kinds of Exceptions

- The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from.
  - For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor of java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

- The second kind of exception is the *error.* These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
  - For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Three Kinds of Exceptions

- The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API.
  - For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

- Errors and runtime exceptions are collectively known as *unchecked exceptions*.

- In Java, programmers are forced to deal with *checked exceptions* only. It includes:
  1. Catch the checked exceptions from improper code and provide a handler for them
  2. Specify all kinds of checked exceptions that a method may throw

Declaring Exceptions

- Every method must state/specify the types of <u>checked exceptions</u> it might throw. This is known as declaring exceptions.
- For examples:

```
public void myMethod() throws IOException {
        // the code here may throw IOException
        // see next page for how to throw exceptions
        // ……
}

public void myMethod() throws IOException, ClassNotFoundException {
        // the code here may throw two types of exceptions
        // ……
}
```

Throwing Exceptions

- Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from the JDK, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the *throw* statement.
- When a program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as throwing an exception. Here are two examples:

```
throw new Exception();
```

```
Exception ex = new Exception();
throw ex;
```

```
 /** Set a new radius */
public void setRadius(double newRadius) throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException("Radius cannot be negative");
}
```

Catching Exceptions – Try Block

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a **try block**. In general, a try block looks like the following.

```
try {
  code
}
catch and finally blocks . . .
```

- The segment in the example labeled *code* contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next slides.)

Catching Exceptions – Catch Block

- You associate exception handlers with a **try block** by providing one or more **catch blocks** directly after the try block.

```
try {
  code                    // may throw exceptions of ExceptionType1 and ExceptionType2
} catch (ExceptionType1 name) {
  code                    // handle exception of ExceptionType1
} catch (ExceptionType2 name) {
  code                    // handle exception of ExceptionType2
}
```

- Each catch block is an exception handler and handles the particular type of exception indicated by its argument. The type of exception must be a subclass of the *Throwable* class.
- The runtime system invokes the exception handler when the handler is the first one whose *ExceptionType* matches the type of the exception thrown.
- The system considers it a match if the thrown object can legally be assigned to the exception handler's argument. (upcasting is possible!)

Example - Without Catching Runtime Errors

```java
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

    // Display the result
    System.out.println(
      "The number entered is " + number);
  }
}
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.

ExceptionDemo

Example - Catching Runtime Errors

```java
import java.util.*;

public class HandleExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        scanner.nextLine(); // discard input
      }
    } while (continueInput);
  }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

HandleExceptionDemo

Example – Using Exception in Circle Class

- This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class. The new setRadius method throws an exception if radius is negative.

TestCircleWithException

Forwarding Exceptions

- Java forces you to deal with checked exceptions. Usually you will use a pair of try-catch block to enclose the error code as shown in (a).
- You may choose to forward the exception to the next handler in the caller. In this case, you can simply declare the exception type being thrown as shown in (b). If you call p1 in somewhere, you therefore need to use a pair of try-catch block to enclose it.

Handle the exception locally, i.e. resolved by p1()

```
void p1() {
  try {
    p2();   //may throw IOException
  }
  catch (IOException ex) {
    ...    //handler for IOException
  }
}

void p2() throws IOException {
  throw new IOException();
}
```

(a)

Forward the exception to the caller of p1()

```
void p1() throws IOException {

  p2();   //may throw IOException

}



void p2() throws IOException {
  throw new IOException();
}
```

(b)

Rethrowing Exceptions

- Sometimes you may want to re-throw the exception that you just caught, particularly when you want the exception to be processed by multiple handlers. In this case, you can simply throw the exception instance again.
- Re-throwing an exception immediately causes a jump to the exception handlers in the caller. Any further catch clauses for the same try block are ignored.

```
try {
  statements;
}
catch(Exception1 ex) {
  perform operations before exits;
  throw ex;
}
catch(Exception2 ex) {
  perform operations before exits;
}
```

Re-throw the instance *ex*

Performing Cleanup – Finally Block

- There is often some piece of code that you want to execute whether or not an exception is thrown within a try block. To achieve this effect, you put a **finally block** at the end of all the exception handlers. This ensures that the finally block is executed (not being bypassed accidentally) even if an unexpected exception occurs.
  - It is common to put cleanup code in the finally block, such as closing an opened file.
- If any catch block re-throws the exception, the finally block will be executed before the exception goes to the next handler in the caller.

```
try {
    // The guarded region: Dangerous activities
    open a file
    read the file                // may throw IOException
    close the file               // this line does not execute if read fail
}   catch(IOException e) {
    // Handler for IOException
}   catch(B b1) {
    // Handler for situation B
} finally {
    // Activities that happen every time
    close the file               // should be closed here instead
}
```

Trace a Program Execution 1

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

Trace a Program Execution 1

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

Trace a Program Execution 1

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}


Next statement;
```

Next statement in the method is executed

Trace a Program Execution 2

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a Program Execution 2

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

Trace a Program Execution 2

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a Program Execution 2

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

Trace a Program Execution 3

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a Program Execution 3

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a Program Execution 3

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

Trace a Program Execution 3

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Re-throw the exception to the caller

*this statement will not be executed.

Try-with-resource Statement

- A resource is an object that must be closed after the program is finished with it. For example, a file or buffered reader (to be discussed in the next topic) is a resource that should be closed after use.
- Prior to Java SE 7, programmer uses a finally block to ensure that a resource is closed. Starting from Java SE 8, you can use the try-with-resource statement that allows us to declare resources to be used in a *try* block with the assurance that the resources will be closed automatically when after execution of that block.
- The resources declared must implement the *AutoCloseable* interface.

```
try (PrintWriter writer = new PrintWriter(new File("test.txt"))) {
    writer.println("Hello World");
}
```

- For multiple resources (separated with a semi-colon):

```
try (Scanner scanner = new Scanner(new File("testRead.txt"));
    PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
    while (scanner.hasNext()) {
    writer.print(scanner.nextLine());
    }
}
```

Replacing try–catch-finally with try-with-resources

- Try-catch-finally:

```java
Scanner scanner = null;
try {
    scanner = new Scanner(new File("test.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

- Try-with-resource (more clean):

```java
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

Advantages of Using Exceptions

1. Separating Error-Handling Code from "Regular" Code
2. Propagating Errors Up the Call Stack
   - A second advantage is the ability to propagate error reporting up the call stack of methods.
   - Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

```
method1 {                       method2 {                      method3 {
   call method2;                   call method3;                  call readFile;
}                               }                              }
```

Advantages of Using Exceptions

- Suppose also that method1 is the only method interested in the errors that might occur within readFile.
- Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1— the only method that is interested in them.

Traditional

```
method1 {
   errorCodeType error;
   error = call method2;
   if (error)
      doErrorProcessing;
   else
      proceed;
}
```

```
errorCodeType method2 {
   errorCodeType error;
   error = call method3;
   if (error)
      return error;
   else
      proceed;
}
```

```
errorCodeType method3 {
   errorCodeType error;
   error = call readFile;
   if (error)
      return error;
   else
      proceed;
}
```

With Exception

```
method1 {
   try {
      call method2;
   } catch (exception e) {
      doErrorProcessing;
   }
}
```

```
method2 throws exception {
   call method3;
}
```

```
method3 throws exception {
   call readFile;
}
```

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program. Assertions is used to assure program correctness and avoid logic errors.
- An assertion contains a Boolean expression that should be true during program execution.
- Assertions are checked at runtime and can be turned on or off at startup time.
- An assertion is declared using the Java keyword assert. There are two forms:

  - assert assertionExpression;   // form 1
  - assert assertionExpression : detailMessage;   // form 2

- where *assertionExpression* is a boolean expression and *detailMessage* is a primitive value or an instance of Object. For example:

  - assert (count == 10);   // form 1
  - assert (count == 10) : -1;   // form 2, send an errorcode -1
  - assert (count == 10) :"Incorrect count value.";   // form 2, send an error message

Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion. If it is false, an AssertionError will be thrown.
- The AssertionError class has a no-arg constructor and seven overloaded single-argument constructors of type int, long, float, double, boolean, char, and Object.
  - For the first assert statement (form 1) with no detail message, the no-arg constructor of AssertionError is used.
  - For the second assert statement (form 2) with a detail message, an appropriate AssertionError constructor is used to match the data type of the message.
- Since AssertionError is a subclass of Error (unchecked), when an assertion is evaluated as false, the program displays a message on the console and exits.

Example - Executing Assertions

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    // assure the loop finished properly
    assert i == 10;
    // assure the sum value is correct
    assert (sum > 10 && sum < 5 * 10) : "sum is " + sum;
  }
}
```

Running Programs with Assertions

- By default, the assertions are **disabled** at runtime. To enable it, use the switch –enableassertions, or –ea for short, as follows:
- java –ea AssertionDemo
- Assertions can be **selectively enabled or disabled** at class level or package level. The disable switch is –disableassertions or –da for short.
- For example, the following command enables assertions in package *packageA* and disables assertions in class *ClassA*.
- java –ea:packageA –da:ClassA AssertionDemo

<u>Latter parameter</u> overrides <u>former parameter</u>

Proper Use of Assertions

- Using assertions as a general-purpose error handling mechanism is unwise because assertions do not allow for recovery from errors. An assertion failure will halt the program's execution abruptly.
- Use assertions to reaffirm assumptions. An assumption is a condition that is supposed to be true in all situations.
  - A common use of assertions is to replace assumptions in comments with assertions in the code.

```
int total = countNumberOfUsers();
if (total % 2 == 0) {
    // total is even
} else {
    // total is odd
}
```

```
int total = countNumberOfUsers();
if (total % 2 == 0) {
    // total is even
} else {
    assert (total % 2 == 1);
}
```

```
assert (month > 0 && month < 13);                    // pre-condition
switch (month) {
  case 1: ... ; break;
  case 2: ... ; break;
  ...
  case 12: ... ; break;
  default:
    assert false : "Invalid month: " + month;        // post-condition
}
```

Exceptions Vs. Assertions

- Exception and Assertion are used for different purposes:
  - Exception handling deals with unusual and unexpected circumstances during program execution, and provides a mechanism to recover from error. Assertions are used to assure a (correct) condition throughout the program execution.
  - Exception addresses robustness (recover from error) and assertion addresses correctness (affirm no error).