CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Exception Handling

- Basic constructs of Java's exception handling: `try`, `catch`, `finally`, `throw` and `throws`.
- Put the statement that may cause an exception inside a `try` block
- Put a `catch` block just after the `try` block.
  - Inside the `catch` block write the code that you want to be executed when an exception occurs.
- An exception is an object which is created during runtime and "thrown" by the Java runtime system to the method that caused the exception.
  - Execution is immediately transferred to the `catch` block.
  - A `catch` block "catches" the thrown exception and executes its code.
- `finally` is an optional block (if at least one catch block is provided) that follows the `catch` block(s).
  - This block is executed irrespective of the exception's occurrence, and furthermore, irrespective of the handling of the exception.
  - Contains code that must be executed anyway, such as file closing tasks.
- Let's see an example that involves `try`, `catch` and `finally` blocks… [EXAMPLE 2]

Multiple `catch` Clauses

- A single piece of code may raise more than one type of exception.
- So we may want to put a distinct handler for each of the possible exceptions.
- These multiple catch blocks are examined in order of their appearance.
  - Whenever a catch block is executed, the others are bypassed to the finally block (if exists).
- Take the next example of division by zero and array index out of bound exceptions…[EXAMPLE 3]
- When using multiple catch blocks, it is important to put the subclass (of `Exception`) before superclass.
  - Because recall that superclass variable can refer to a subclass object, so a subclass that is put later than its superclass in catch block will never be executed
    - Also, unreachable code is a compile-time error in Java

```
// Demonstrate multiple catch statements.
class MultipleCatches {
  public static void main(String[] args) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int[] c = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Example from Textbook

```
/* This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String[] args) {
    try {
      int a = 0;
      int b = 42 / a;
     } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

```
// An example of nested try statements.
class NestTry {
  public static void main(String[] args) {
    try {
      int a = args.length;

      /* If no command-line args are present,
          the following statement will generate
          a divide-by-zero exception. */
      int b = 42 / a;

       System.out.println("a = " + a);

       try { // nested try block
         /* If one command-line arg is used,
            then a divide-by-zero exception
            will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
           int[] c = { 1 };
           c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

Nested try statement: Example from Textbook

- try within try [EXAMPLE 4]

Output with 0, 1 and 2 command line arguments?

Nested try statement: Example from Textbook

```
// An example of nested try statements.
class NestTry {
  public static void main(String[] args) {
    try {
      int a = args.length;

      /* If no command-line args are present,
          the following statement will generate
          a divide-by-zero exception. */
      int b = 42 / a;

       System.out.println("a = " + a);

       try { // nested try block
         /* If one command-line arg is used,
            then a divide-by-zero exception
            will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
           int[] c = { 1 };
           c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:
  Index 42 out of bounds for length 1
```

Exception Handling: “throw” and “throws” clauses

- Using `throw` clause, a programmer can explicitly “throw” an exception object.
  - This is written when we want to signal an exceptional situation which is not listed in Java’s exceptional cases, but our own choice.
- The clause `throws` is used when a method doesn’t handle an exception (using `catch` block), but rather “throws” the exception out of itself to its calling statement (where it should be caught and dealt with).
- Let’s see an example… [EXAMPLE 5]
- A `try-catch` is a complete unit.
  - Whenever you put a `try` block, that must have at least one `catch` block (or at least `finally` block if we are dealing with “unchecked exceptions” - this will be discussed after a couple of slides).

Creating Your Own Exception

- So far we have discussed automatically produced exceptions
- Although Java provides with a good number of built-in exceptions, we oftentimes want to define our own exceptions to handle some specific situation of our program.
  - The created exception can then be thrown using throw clause.
- What we need to do is just to extend the `Exception` class.
  - Recall that `Exception` is also a subclass of `Throwable`
  - Thus our own exceptions have all the methods provided by `Throwable`
  - Usually we override the `toString()` method of `Throwable` to offer a customized description of our exception.
- Let's see an example ... [EXAMPLE 6]

Example from Textbook

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String[] args) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String[] args) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Example from Textbook

Output:

```
Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]
```

Are Exceptions Mandatory to Handle?

- Some exceptions are mandatory to handle (using catch block) while some others are not.
  - They are called checked and unchecked exceptions respectively.
- List of checked and unchecked exceptions: find yourselves (e.g.: Java Checked and Unchecked Exceptions).

```
    public void has_unchecked(){ //requires no throws even
//though exception is thrown
        throw new ArithmeticException();
    }

    public void has_checked() throws Exception{//requires throws
        throw new FileNotFoundException();
    }
```

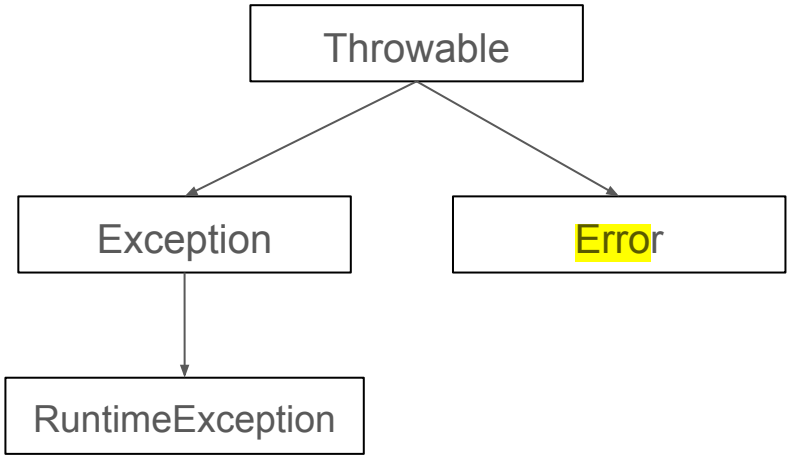
What Happens if an Exception is Not Caught?

- When Java Runtime Environment (JRE) experiences an error during execution of a Java program, it creates an `Throwable` object (or one of its subclasses) and "throw"s it to the method where the exception was originated from.
- This exception must be dealt with immediately (either in the very method where the statement in question resides in, or by a method in the method-calling stack)
  - If properly dealt with, the code of the corresponding `catch` block (and `finally`, if any) is executed, and then execution of the program resumes just after the `finally` block.
- If this exception is neither caught nor thrown away from the method, and if it is an unchecked exception, a default exception handler is provided by the JRE.
  - The default handler prints a short description of the exception and a stack trace of the error, and terminates the program immediately.
  - The stack trace shows all the methods in the method-calling hierarchy that are involved in the error.
  - For checked exceptions, you must provide a `catch` block or `throws` clause.

About Exception Class Hierarchy

- `Throwable` is a built-in class that is inherited by all exception classes.
  - `Throwable` class is a member of `java.lang` package, which is by default imported in every Java program.
  - Recall that every class in Java implicitly extends `Object` class, and `Throwable` is no exception to that.
- The two immediate classes of `Throwable` are: `Exception` and `Error`.
- An object of type `Exception` is created when "normal" errors occur like division by zero or going out of bound of array index.
  - An important subclass of `Exception` class is `RuntimeException`
- An object of type `Error` is created when relatively uncommon errors occur such as stack overflow problem.
  - We will not study this subclass as it is not very much used in Java programs.
- Displaying a description of exception is done by printing the `Exception` object.
  - `Throwable` overrides the `toString()` method of `Object` class.

Exception Types



Important: Although "catch"ing an exception may seem just like parameter passing to methods, technically it is not the same:

```
Object e = new Throwable(); //allowed as usual
catch (Object e) // not allowed
catch (Throwable e) // allowed
```

Similarly:

```
throw new any-class();
```

//not allowed unless `any-class` extends `Throwable` or any of it's subclasses

List of Java exceptions along with class hierarchy: https://programming.guide/java/list-of-java-exceptions.html
Throwable class: https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

Why Using Your Own Handler?

- You might wonder: “if I’m getting a clear stack trace of the error, why bother with my own handling code?”
  - Again, because it provides you a more controlled way of handling the error.
  - You can fix the error and resume the program instead of getting abruptly terminated.

`throw` Clause

- Syntax: `throw` *`ThrowableObject`*`;`
  - `ThrowableObject` must be an object of `Throwable` class or one of its subclasses.
  - Primitive types cannot be thrown (`throw new int;` // not allowed)
    - This feature is a contrast to that of C++.
- The flow of execution stops immediately after this statement, and jumps to `catch` clauses (if exist), or returns from the method (if `throws` is provided), or the default handler provided by JRE is executed.

Rethrowing an Exception and `throws` Clause

- After catching an exception, it might be rethrown to either the calling statement, or to the default handler.
- Let's see an example…[EXAMPLE 7]
- If an exception is explicitly thrown from a method, that method must provide some appropriate catch clause, or must use a throws clause.
  - Otherwise, the code will not compile.
  - However, always recall the unchecked and checked exceptions: unchecked exceptions need not be included in throws clause.

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String[] args) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String[] args) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

Output:

```
Caught inside demoproc.
Recaught:
java.lang.NullPointerExcep
tion: demo
```

Example from Textbook

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String[] args) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Example from Textbook

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String[] args) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

```
inside throwOne

caught
java.lang.IllegalAccessException:
demo
```

More on `finally` Clause

- If a program needs to execute some code irrespective of handling an exception, that code is put in the finally block.
  - E.g.: an opened file should be closed before the program ends.
- For checked exceptions, every try block must have a catch block.
- For unchecked exceptions, every try block must have a catch or finally block.

Example from Textbook

```
// Demonstrate finally.
class FinallyDemo {
  // Throw an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String[] args) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }

    procB();
    procC();
  }
}
```

Example from Textbook

```
// Demonstrate finally.
class FinallyDemo {
  // Throw an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String[] args) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }

    procB();
    procC();
  }
}
```

Output:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Why is `finally` Block Used?

```
try {
        int c = a/b;
        System.out.println("After divide by zero.");
        //FileReader fr = new FileReader("");
        //ob.f();
        System.out.println("something");
        //throw new NullPointerException();
        //System.exit(0);
        return;
    }
    catch (ArithmeticException e){
        System.out.println(e);
        System.out.println("Inside my catch block.");
    }
    finally{
        System.out.println("Inside finally block.");
    }
    System.out.println("Near the end.");
```

Output:
After divide by zero.
something
Inside finally block.

```
try {
        int c = a/b;
        System.out.println("After divide by zero.");
        //FileReader fr = new FileReader("");
        //ob.f();
        System.out.println("something");
        //throw new NullPointerException();
        System.exit(0);
        //return;
    }
    catch (ArithmeticException e){
        System.out.println(e);
        System.out.println("Inside my catch block.");
    }
    finally{
        System.out.println("Inside finally block.");
    }
    System.out.println("Near the end.");
```

Why is `finally` Block Used?

Output:
After divide by zero.
something

```
try {
    int c = a/b;
    System.out.println("After divide by zero.");
    //FileReader fr = new FileReader("");
    //ob.f();
    System.out.println("something");
    throw new NullPointerException();//unchecked
exception
    //System.exit(0);
    //return;
    }
    catch (ArithmeticException e){
        System.out.println(e);
        System.out.println("Inside my catch block.");
    }
    finally{
        System.out.println("Inside finally block.");
    }
    System.out.println("Near the end.");
```

Why is `finally` Block Used?

Output:

```
After divide by zero.
something
Inside finally block.

Exception in thread "main"
java.lang.NullPointerException
        at
ExceptionHandling.Basics.main(B
asics.java:35)
C:\Users\Muhammad
Ibrahim\AppData\Local\NetBeans\
Cache\12.6\executor-snippets\run.
xml:111: The following error
occurred while executing this line:
C:\Users\Muhammad
Ibrahim\AppData\Local\NetBeans\
Cache\12.6\executor-snippets\run.
xml:94: Java returned: 1
BUILD FAILED (total time: 0
seconds)
```

Optional Topics:
Three New Features of Exception Handling (JDK7+)

1. Automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the try statement called try-with-resources.
2. Multi-catch clause
3. Final rethrow or more precise rethrow. Not much used.

Optional Topic: Multi-Catch

Allows two or more exceptions to be caught by the same `catch` clause.

Because each multi-catch parameter is implicitly `final`, it can't be assigned a new value.

```
// Demonstrate the multi-catch feature.
class MultiCatch {
  public static void main(String[] args) {
    int a=10, b=0;
    int[] vals = { 1, 2, 3 };

    try {
      int result = a / b; // generate an ArithmeticException

//      vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

      // This catch clause catches both exceptions.
    } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception caught: " + e);
    }

    System.out.println("After multi-catch.");
  }
}
```

Optional Topics: Chained Exception

Self-study from textbook

Final Comment on Exception Handling

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time situations.

It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic.

Instead of using error return codes to indicate failure, you should use Java's exception handling capabilities.

Thus, in the situation when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

End of Lecture 13.

Reading materials: Up to Chapter 10 of the textbook.

Note: Some text of this slide are copied verbatim from your textbook.