JAVA PROGRAMMING

Week 4: Exception Handling

Lecturer:

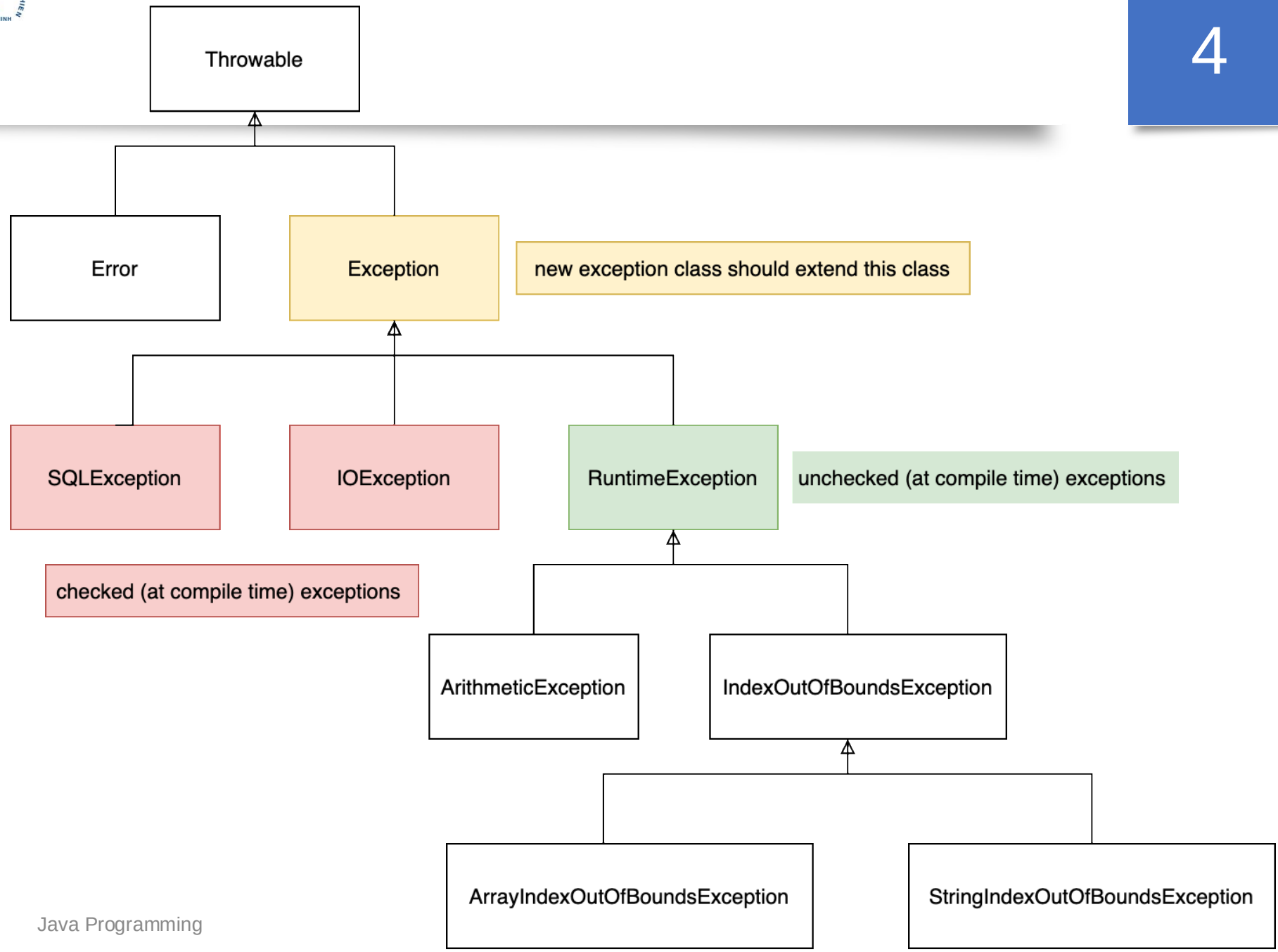
- HO Tuan Thanh, M.Sc.

Plan

1. Exception hierarchy
2. Consequences of an uncaught exception
3. Using multiple catch statements
4. Catching subclass exceptions
5. Try blocks can be nested
6. Throwing an exception
7. A closer look at throwable
8. Java’s built-in exceptions
9. Creating exception subclasses

The exception hierarchy

- In Java, all exceptions are represented by classes derived from the class Throwable.
- Two direct subclasses of Throwable: **Exception** and **Error**
- Exceptions of type Error are related to errors that occur in the Java Virtual Machine itself, and not in your program.
  - Are beyond your control, and your program will not usually deal with them.
- Errors that result from program activity are represented by sub-classes of Exception
  - Example: divide-by-zero, array boundary, and file errors.
  - An important subclass of Exception is RuntimeException, which is used to represent various common types of runtime errors.


Throwable
Error
Exception
new exception class should extend this class
SQLException
IOException
RuntimeException
unchecked (at compile time) exceptions
checked (at compile time) exceptions
ArithmeticException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException

Exception handling fundamentals

- Keywords: try, catch, throw, throws, and finally.
- Program statements that you want to monitor for exceptions are contained within a try block.
- If an exception occurs within the try block, it is thrown.
- Your code can catch this exception using catch and handle it in some rational manner.
- System generated exceptions are automatically thrown by the Java runtime system.
- To manually throw an exception: use the keyword throw.
- Any code that absolutely must be executed upon exiting from a try block is put in a finally block.

Using try and catch


```
1.  try {
2.  // block of code to monitor for errors
3.  }catch (ExcepType1 exOb) {
4.          // handler for ExcepType1
5.  }catch (ExcepType2 exOb) {
6.          // handler for ExcepType2
7.  }
8.    .
9.    .
10.   .
```

- ExcepType is the type of exception that has occurred

Example


```
1.  class ExcDemo1 {
2.      public static void main(String args[]) {
3.          int nums[] = new int[4];
4.          try {
5.              System.out.println("Before exception is
6.               generated.");
7.              // Generate an index out-of-bounds exception.
8.              nums[7] = 10;
9.              System.out.println("this won't be displayed");
10.         }catch (ArrayIndexOutOfBoundsException exc) {
11.             // catch the exception
12.             System.out.println("Index out-of-bounds!");
13.         }
14.         System.out.println("After catch statement.");
15.     }
16. }
```

```
1.  class ExcTest {
2.      // Generate an exception.
3.      static void genException() {
4.          int nums[] = new int[4];
5.          System.out.println("Before exception is generated.");
6.          nums[7] = 10;
7.          System.out.println("this won't be displayed");
8.      }
9.  }
10. class ExcDemo2 {
11.     public static void main(String args[]) {
12.         try {  ExcTest.genException();
13.         } catch (ArrayIndexOutOfBoundsException exc) {
14.             System.out.println("Index out-of-bounds!");
15.         }
16.         System.out.println("After catch statement.");
17.     }
18. }
```

Consequences of an uncaught exception

- Catching one of Java's standard exceptions has a side benefit: It prevents abnormal program termination.
- When an exception is thrown: it must be caught by some piece of code, somewhere.
- In general: if your program does not catch an exception, then it will be caught by the JVM.
  - The trouble is that the **JVM's default exception handler terminates execution** and **displays a stack trace and error message**.

→ It is important for your program to handle exceptions itself, rather than rely upon the JVM.

Example: Consequences of an uncaught exception


```
1.  class NotHandled {
2.      public static void main(String args[]) {
3.          int nums[] = new int[4];
4.
5.          System.out.println("Before exception is generated.");
6.
7.          // generate an index out-of-bounds exception
8.          nums[7] = 10;
9.      }
10. }
```

```
1.  class ExcTypeMismatch {
2.      public static void main(String args[]) {
3.          int nums[] = new int[4];
4.          try {
5.              System.out.println("Before exception is
6.               generated.");
7.              nums[7] = 10;
8.              System.out.println("this won't be displayed");
9.          }
10.         /* Can't catch an array boundary error with an
11.         ArithmeticException. */
12.         catch (ArithmeticException exc) {
13.             // catch the exception
14.             System.out.println("Index out-of-bounds!");
15.         }
16.         System.out.println("After catch statement.");
17.     }
18. }
```

Benefits of exception handling

- It enables your program to respond to an error and then continue running.

```
1.  class ExcDemo3 {
2.      public static void main(String args[]) {
3.          int numer[] = { 4, 8, 16, 32, 64, 128 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.
6.          for (int i = 0; i < numer.length; i++) {
7.              try {
8.                  System.out.println(numer[i] + " / " +
9.                      denom[i] + " is " + numer[i] / denom[i]);
10.             } catch (ArithmeticException exc) {
11.                 // catch the exception
12.                 System.out.println("Can't divide by Zero!");
13.             }
14.         }
15.     }
16. }
```

Once an exception has been handled, it is removed from the system

Using multiple catch statements

- It is possible to associate more than one catch statement with a try.
- Each catch must catch a different type of exception.
  - Responds only to its own type of exception.
- In general:
  - catch expressions are checked in the order in which they occur in a program.
  - Only a matching statement is executed.
  - All other catch blocks are ignored.

Example


```
1.  class ExcDemo4 {
2.      public static void main(String args[]) {
3.          int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.          for (int i = 0; i < numer.length; i++) {
6.              try {
7.                  System.out.println(numer[i] + " / " +
8.                                     denom[i]+ " is " + numer[i] / denom[i]);
9.              } catch (ArithmeticException exc) {
10.               System.out.println("Can't divide by Zero!");
11.             } catch (ArrayIndexOutOfBoundsException exc) {
12.               System.out.println("No matching element found.");
13.             }
14.         }
15.     }
16. }
```

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

Catching subclass exceptions

- A catch clause for a superclass will also match any of its subclasses.
  - Example: to catch all possible exceptions, catch Throwable.
- If you want to catch exceptions of both a superclass and a subclass type: put the subclass first in the catch sequence.
  - If you don't: the superclass catch will also catch all derived classes.
  - This rule is self-enforcing because putting the superclass first causes unreachable code to be created, since the subclass catch clause can never execute.
  - In Java, unreachable code is an error.

Example


```
1.  class ExcDemo5 {
2.      public static void main(String args[]) {
3.          int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.          for (int i = 0; i < numer.length; i++) {
6.              try {
7.                  System.out.println(numer[i] + " / " + denom[i]+
                                                " is " + numer[i] / denom[i]);
8.              } catch (ArrayIndexOutOfBoundsException exc) {
9.                System.out.println("No matching element found.");
10.             } catch (Throwable exc) {
11.               System.out.println("Some exception occurred.");
12.             }
13.         }
14.     }
15.
```

```
4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
Some exception occurred.
128 / 8 is 16
No matching element found.
No matching element found.
```

try blocks can be nested

- One try block can be nested within another.
- An exception generated within the inner try block that is not caught by a catch associated with that try is propagated to the outer try block.

```
1.  class NestTrys {
2.      public static void main(String args[]) {
3.          int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.          try { // outer try
6.              for (int i = 0; i < numer.length; i++) {
7.                  try { // nested try
8.                    System.out.println(numer[i] + " / " +
9.                                  denom[i]+" is "+numer[i]/denom[i]);
10.                 } catch (ArithmeticException exc) {
11.                  System.out.println("Can't divide by Zero!");
12.                 }
13.             }
14.         }catch (ArrayIndexOutOfBoundsException exc) {
15.          System.out.println("No matching element found.");
16.          System.out.println("Fatal error--program terminated.");
17.         }
18.     }
19. }
```

Throwing an exception

- It is possible to manually throw an exception by using the throw statement. General form:

```
throw exceptOb;
```

exceptOb must be an object of an exception class derived from Throwable.

Example


```
1.  class ThrowDemo {
2.      public static void main(String args[]) {
3.          try {
4.              System.out.println("Before throw.");
5.              throw new ArithmeticException();
6.          } catch (ArithmeticException exc) {
7.              System.out.println("Exception caught.");
8.          }
9.          System.out.println("After try/catch block.");
10.     }
11. }
```

```
Before throw.
Exception caught.
After try/catch block.
```

Remember: throw throws an object → you must create an object for it to throw.

Rethrowing an Exception

- An exception caught by one `catch` statement can be rethrown so that it can be caught by an outer `catch`.
- This allows multiple handlers access to the exception.
  - Example: perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect.
- **Remember**: when you rethrow an exception:
  - It will not be re-caught by the same `catch` statement.
  - It will propagate to the next `catch` statement.

```
1.  class Rethrow {
2.      public static void genException() {
3.          int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.          for (int i = 0; i < numer.length; i++) {
6.              try {
7.                  System.out.println(numer[i] + " / " + denom[i]+
8.                                                  " is " + numer[i] / denom[i]);
9.              } catch (ArithmeticException exc) {
10.               System.out.println("Can't divide by Zero!");
11.             } catch (ArrayIndexOutOfBoundsException exc) {
12.               System.out.println("No matching element found.");
13.                 throw exc; // rethrow the exception
14.             }
15.         }
16.     }
17. }
```

```
1.  class RethrowDemo {
2.      public static void main(String args[]) {
3.          try {
4.              Rethrow.genException();
5.          } catch (ArrayIndexOutOfBoundsException exc) {
6.              // recatch exception
7.              System.out.println("Fatal error -- " +
8.                                 "program terminated.");
9.          }
10.     }
11. }
```

Methods defined by Throwable

- A catch clause specifies an exception type and a parameter which receives the exception object.
- All exceptions are subclasses of Throwable → all exceptions support the methods defined by Throwable.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString( )	Returns a **String** object containing a complete description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

Example


```
1.  class ExcTest1 {
2.      static void genException() {
3.          int nums[] = new int[4];
4.          System.out.println("Before exception is generated.");
5.          // generate an index out-of-bounds exception
6.          nums[7] = 10;
7.          System.out.println("this won't be displayed");
8.      }
9.  }
```

```
1.  class UseThrowableMethods {
2.      public static void main(String args[]) {
3.          try {
4.                  ExcTest1.genException();
5.          } catch (ArrayIndexOutOfBoundsException exc) {
6.              System.out.println("Standard message is: ");
7.              System.out.println(exc);
8.              System.out.println("\nStack trace: ");
9.              exc.printStackTrace();
10.         }
11.         System.out.println("After catch statement.");
12.     }
13. }
```

```
Before exception is generated.
Standard message is:
java.lang.ArrayIndexOutOfBoundsException: 7

Stack trace:
After catch statement.
java.lang.ArrayIndexOutOfBoundsException: 7
        at Week9.ExcTest1.genException(UseThrowableMethods.java:12)
        at Week9.UseThrowableMethods.main(UseThrowableMethods.java:21)
```

Using finally

- To specify a block of code to execute when a try/catch block is exited, include a finally block at the end of a try/catch sequence.
- The finally block will be executed whenever execution leaves a try/catch block, no matter what conditions cause it.
  - That is: whether the try block ends normally, or because of an exception, the last code executed is that defined by finally.
  - The finally block is also executed if any code within the try block or any of its catch statements return from the method.

finally statement


```
1.  try {
2.          // block of code to monitor for errors
3.  }
4.  catch (ExcepType1 exOb) {
5.          // handler for ExcepType1
6.  }
7.  catch (ExcepType2 exOb) {
8.          // handler for ExcepType2
9.  }
10. //...
11. finally {
12.         // finally code
13. }
```

```
1.  class UseFinally {
2.      public static void genException(int what) {
3.          int t, nums[] = new int[2];
4.          System.out.println("Receiving " + what);
5.          try {
6.              switch (what) {
7.                  case 0: //generate div-by-zero error
8.                          t = 10 / what; break;
9.                  case 1: // generate array index error.
10.                         nums[4] = 4; break;
11.                 case 2:
12.                         return; // return from try block
13.             }
14.         } catch (ArithmeticException exc) {
15.                 System.out.println("Can't divide by Zero!");
16.                 return; // return from catch
17.         } catch (ArrayIndexOutOfBoundsException exc) {
18.                 System.out.println("No matching element found.");
19.         } finally { System.out.println("Leaving try."); }
20.     }
21. }
```

```
1.  class FinallyDemo {
2.      public static void main(String args[]) {
3.          for (int i = 0; i < 3; i++) {
4.              UseFinally.genException(i);
5.              System.out.println();
6.          }
7.      }
8.  }
```

```
Receiving 0
Can't divide by Zero!
Leaving try.

Receiving 1
No matching element found.
Leaving try.

Receiving 2
Leaving try.
```

Using throws

- In some cases, if a method generates an exception that it does not handle, it must declare that exception in a throws clause.

```
rettype methName(paramlist) throws exceptlist {
        // body
}
```

- exceptlist is a comma-separated list of exceptions that the method might throw outside of itself.

```
1.  class ThrowsDemo {
2.      public static char prompt(String str)
3.                                  throws java.io.IOException {
4.          System.out.print(str + ": ");
5.          return (char) System.in.read();
6.      }
7.      public static void main(String args[]) {
8.          char ch;
9.          try {
10.                 ch = prompt("Enter a letter");
11.         } catch (java.io.IOException exc) {
12.                 System.out.println("I/O exception occurred.");
13.                 ch = 'X';
14.         }
15.         System.out.println("You pressed " + ch);
16.     }
17. }
```

Multi-catch

- Allows two or more exceptions to be caught by the same `catch` clause.
- Can use a single `catch` clause to handle the exceptions without code duplication.
- To create a multi-catch:
  - Specify a list of exceptions within a single `catch` clause.
  - Separate each exception type in the list with the `OR` operator.
  - Each multi-catch parameter is implicitly final → it can't be assigned a new value.

```
1.  class MultiCatch {
2.      public static void main(String args[]) {
3.          int a = 88, b = 0, result;
4.          char chrs[] = { 'A', 'B', 'C' };
5.
6.          for (int i = 0; i < 2; i++) {
7.              try {
8.                  if (i == 0) result = a / b;
9.                  // generate an ArithmeticException
10.                 else chrs[5] = 'X';
11.             } catch (ArithmeticException |
12.                 ArrayIndexOutOfBoundsException e){
13.                 System.out.println("Exception caught: " + e);
14.             }
15.         }
16.         System.out.println("After multi-catch.");
17.     }
18. }
```

Java's built-in exceptions

- Inside the standard package java.lang, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException.
  - Since java.lang is implicitly imported into all Java programs, many exceptions derived from RuntimeException are automatically available.
  - They need not be included in any method's throws list.

  → These are called **unchecked exceptions** because the compiler does not check to see if a method handles or throws these exceptions.

Unchecked exceptions defined in java.lang [1]

Exception	Meaning
ArithmeticException	Arithmetic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Unchecked exceptions defined in java.lang [2]

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Creating exception subclasses [1]

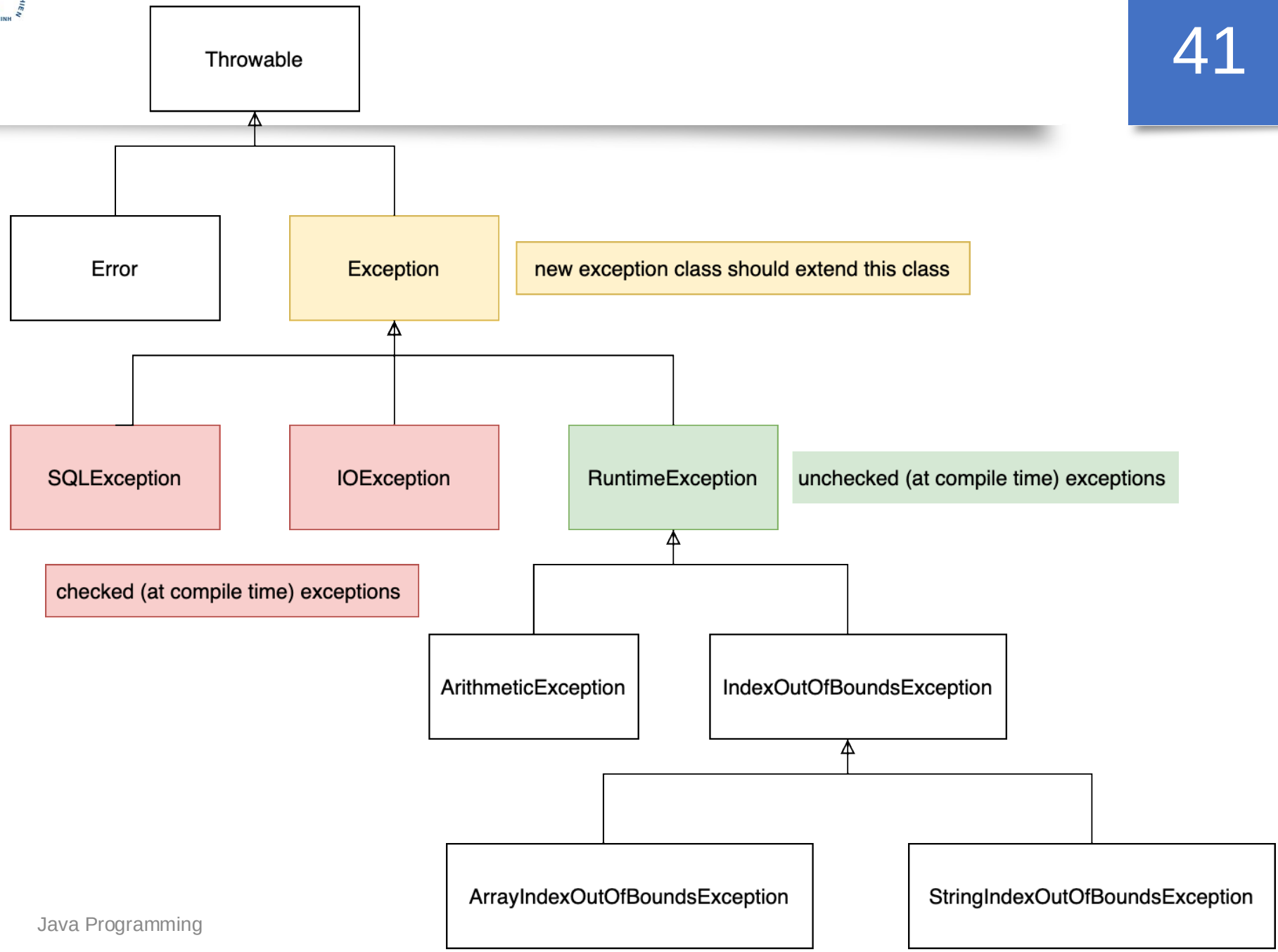
- Although Java's built-in exceptions handle most common errors, Java's exception handling mechanism is not limited to these errors.
- Part of the power of Java's approach to exceptions is its ability to handle exception types that you create.
- Through the use of custom exceptions, you can manage errors that relate specifically to your application.
- Creating an exception class: define a subclass of Exception (which is, a subclass of Throwable).
  - Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

Creating exception subclasses [2]

- The Exception class does not define any methods of its own.
  - It inherits those methods provided by Throwable.

→All exceptions, including those that you create, have the methods defined by Throwable available to them.

  - You can override one or more of these methods in exception subclasses that you create.


Throwable
Error
Exception
new exception class should extend this class
SQLException
IOException
RuntimeException
unchecked (at compile time) exceptions
checked (at compile time) exceptions
ArithmeticException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException

Example


```
1.  //Create an exception.
2.  class NonIntResultException extends Exception {
3.      int n;
4.      int d;
5.      NonIntResultException(int i, int j) {
6.          n = i;
7.          d = j;
8.      }
9.      public String toString() {
10.         return "Result of " + n + " / " + d +
11.                                          " is non-integer.";
12.     }
13. }
```

```
1.  class CustomExceptDemo {
2.      public static void main(String args[]) {
3.          int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
4.          int denom[] = { 2, 0, 4, 4, 0, 8 };
5.          for (int i = 0; i < numer.length; i++) {
6.              try {
7.                  if ((numer[i] % 2) != 0)
8.                  throw new NonIntResultException(numer[i], denom[i]);
9.                  System.out.println(numer[i] + " / " + denom[i]+
10.                                     " is " + numer[i] /denom[i]);
11.             } catch (ArithmeticException exc) {
12.                 System.out.println("Can't divide by Zero!");
13.             } catch (ArrayIndexOutOfBoundsException exc) {
14.                 System.out.println("No matching element found.");
15.             } catch (NonIntResultException exc) {
16.                 System.out.println(exc);
17.             }
18.         }
19.     }
20. }
```

QUESTION ?