Exception Handling

By Warren Mansur, edited by Eric Braude

BOSTON UNIVERSITY METROPOLITAN COLLEGE

Exceptions Agenda

1. **Introduction**
2. Handled vs. unhandled exceptions
3. *Checked vs. unchecked* exceptions
4. *finally* and chained exceptions
5. User-defined exceptions

Exceptions vs. Errors

- Related, but different implications
- *Both* describe an event that disrupts normal flow
- *Exception*: language has built-in syntax for wrapping errors and separate code paths for handling them
- *Error* only suggests problem has occurred

Handling Errors Without Exceptions

- Traditional error handling requires programmer to remember to check return codes and out variables for errors, use conditionals to handle errors
- Result: uncaught errors

```
int main () {

   FILE * pf; // password-protected file
   pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {
     // handle error here…
   }
}
```

```
int main () {

   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      //handle error here…
   }
   else {
      quotient = dividend / divisor;
   }
}
```

Exception Advantages

- Error *detection* separated by syntax from error *handling*
  - Detection of the error usually occurs in a called method
    - (same as with traditional)
  - Handling of the error occurs in a special block just for exceptions
    - (not available with traditional)
- Methods can be labeled with the exceptions they throw
- Callers of methods that throw exceptions can be forced to handle the exception (which includes “passing the buck”)

Built-in Java Exception Classes

```
AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException,
BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException,
BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException,
DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException,
GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException,
InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException,
IOException, JAXBException, JMException, KeySelectorException, LambdaConversionException, LastOwnerException,
LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException,
NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException,
ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException,
ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException,
ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException,
TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException,
UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException,
XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Built-in ArithmeticException

Class ArithmeticException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.ArithmeticException

All Implemented Interfaces:

Serializable

https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html

BOSTON UNIVERSITY METROPOLITAN COLLEGE

Exceptions Agenda

1. Introduction
2. **Handled vs. unhandled exceptions**
3. *Checked vs. unchecked* exceptions
4. *finally* and chained exceptions
5. User-defined exceptions

Example of *Unhandled* Exception

```
public class UnhandledException {

  public static void main(String[] args) {
    int dividend = 23;
    int divisor = 0;
    int result = dividend / divisor;

    System.out.printf("Result = %d%n", result);
  }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exception_handling.UnhandledException.main(UnhandledException.java:7)
```

Example of *Handled* Exception

```
public class HandledException {

  public static void main(String[] args) {
    int dividend = 23;
    int divisor = 0;
    try {
      int result = dividend / divisor;
      System.out.printf("Result = %d%n", result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Attempt to divide by zero");
    }

    System.out.println("This line is reached exception or no exception");
  }
}
```

Output

```
Attempt to divide by zero
This line is reached exception or no exception
```

throws Declaration

```
public class ThrowsDeclaration {

  public static void main(String[] args) {
    int dividend = 23;
    int divisor = 0;
    try {
      int result = quotient(dividend,divisor);
      System.out.printf("Result = %d%n", result);
    }
    catch (ArithmeticException ex) {
    System.out.println("Attempt to divide by zero");
    }

    System.out.println("This line is reached exception or no exception");
}
  public static int quotient(int dividend, int divisor)
       throws ArithmeticException {
    return dividend / divisor;
  }
}
```

Output

```
Attempt to divide by zero
This line is reached exception or no exception
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Exceptions Agenda

1. Introduction
2. Handled vs. unhandled exceptions
3. ***Checked vs. unchecked* exceptions**
4. *finally* and chained exceptions
5. User-defined exceptions

Checked Exceptions in a Method

- A question for handled exceptions
- Checked = force callers to either handle exception or to propagate the exception to the next
- Rule of thumb: if caller can possibly take some action to correct the issue, use a checked exception
- “When a method sometimes cannot do what its name claims, use a checked exception.”

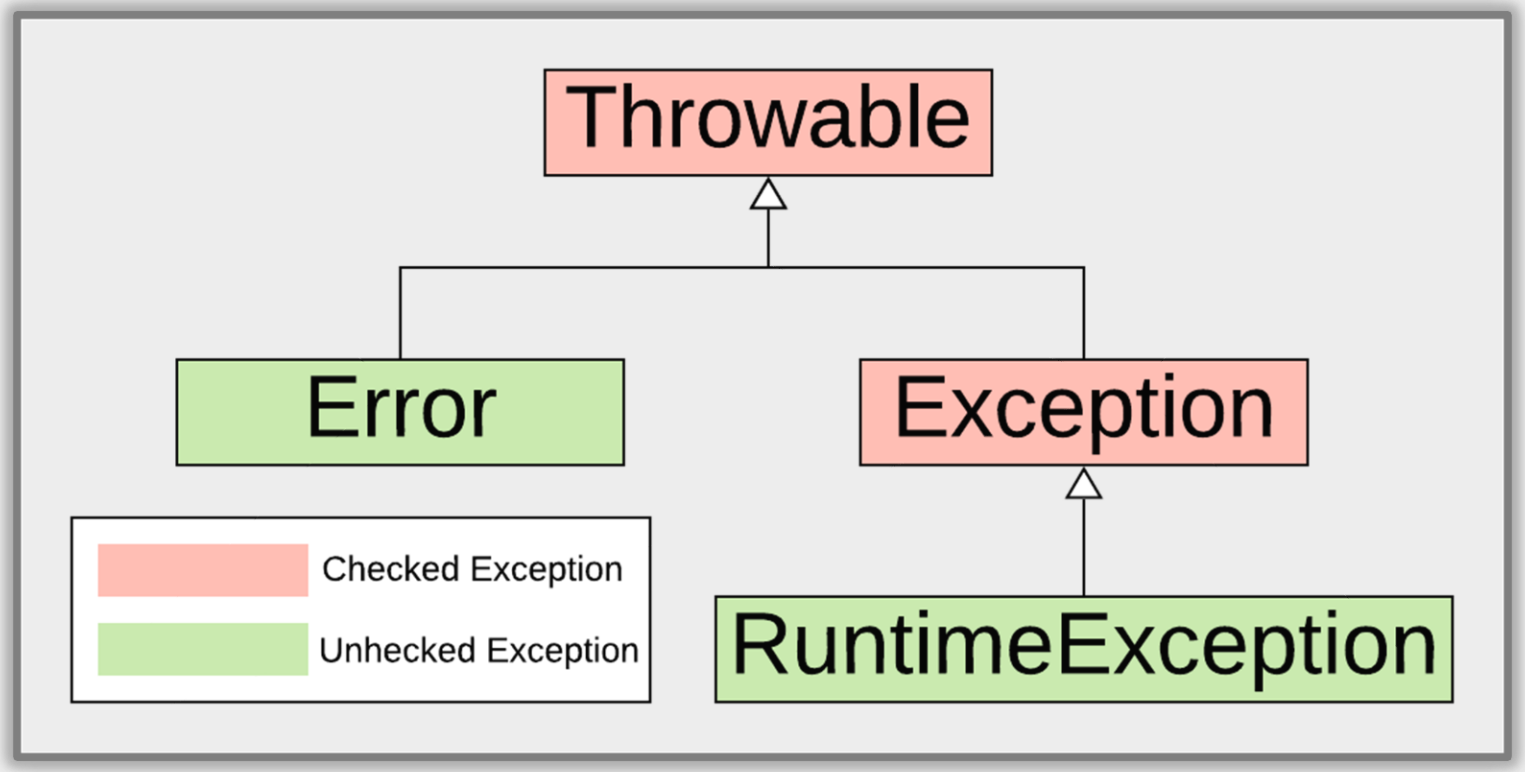
Unhandled Checked Exception Compiler Error

```
public class CheckedException {

  public static void main(String[] args) {
    throwCheckedException();
  }

  public static void throwCheckedException() throws DataFormatException {
    throw new DataFormatException("Uh Oh!");
  }
}
```

Compiler Error

```
Unhandled exception type DataFormatException
```

Checked/Unchecked Hierarchy



Any exception extending *Error* or *RuntimeException* is unchecked

Checked Exception Controversy

- Supporters: forcing programmers to handle exceptions makes code more robust
- Opponents: overkill, cause problems such as:
  - every method in call stack forced to handle an exception instead of the one designated method
  - code is written to catch and swallow exceptions instead of letting error crash the app as it should
  - abuse of checked exceptions forces programmers to over-catch
- Alternative: return values treated for errors.

BOSTON UNIVERSITY METROPOLITAN COLLEGE

Exceptions Agenda

1. Introduction
2. Handled vs. unhandled exceptions
3. *Checked vs. unchecked* exceptions
4. ***finally* and chained exceptions**
5. User-defined exceptions

finally Clause

```
public class FinallyDemo {

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int dividend = input.nextInt();
      int divisor = input.nextInt();
      try {
        int result = quotient(dividend, divisor);
        System.out.printf("Result = %d%n", result);
      }
      catch (ArithmeticException ex) {
         System.out.println("Attempt to divide by zero.");
      }
      finally {
          System.out.println("Finally clause always executed");
          input.close();
      }
      System.out.println("This line reached even when divisor = zero");
   }
   public static int quotient(int dividend, int divisor) throws ArithmeticException
   {
      return dividend / divisor;
   }
}
```

Output

```
Attempt to divide by zero.
Finally clause always executed
This line reached even when divisor = zero
```

Chained Exceptions

```
public class ChainedException {
   public static void main(String[] args) {
       try {
           methodCallingMethodThatThrows();
       }
       catch(Exception ex) {
           System.out.println("Exception caught and processed in main().");
           ex.printStackTrace();
       }
   }
   public static void methodCallingMethodThatThrows() throws Exception {
      try {
          methodThatThrows();
      }
      catch (Exception ex) {
          throw new Exception("Exception from methodCallingMethodThatThrows", ex);
      }
   }
   public static void methodThatThrows() throws Exception {
      throw new Exception("Exception thrown in methodThatThrows");
   }
}
```

Chained Exceptions

```java
public class ChainedException {
   public static void main(String[] args) {
       try {
           methodCallingMethodThatThrows();
       }
       catch(Exception ex) {
           System.out.println("Exception caught and processed in main().");
           ex.printStackTrace();
       }
   }
   public static void methodCallingMethodThatThrows() throws Exception {
      try {
          methodThatThrows();
      }
      catch (Exception ex) {
          throw new Exception("Exception from methodCallingMethodThatThrows", ex);
      }
   }
   public static void methodThatThrows() throws Exception {
      throw new Exception("Exception thrown in methodThatThrows");
   }
}
```

Output

```
Exception caught and processed in main().
java.lang.Exception: Exception from methodCallingMethodThatThrows
        at exception_handling.ChainedException.methodCallingMethodThatThrows(ChainedException.java:19)
        at exception_handling.ChainedException.main(ChainedException.java:7)
Caused by: java.lang.Exception: Exception thrown in methodThatThrows
        at exception_handling.ChainedException.methodThatThrows(ChainedException.java:23)
        at exception_handling.ChainedException.methodCallingMethodThatThrows(ChainedException.java:16)
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Exceptions Agenda

1. Introduction
2. Handled vs. unhandled exceptions
3. *Checked vs. unchecked* exceptions
4. *finally* and chained exceptions
5. **User-defined exceptions**

User-defined Exceptions

- Must inherit from *Exception* class
- Provide variables, constructor(s), toString()
- Use like any other Exception

Custom Exception *1*

```
public class UnderAgeException extends Exception {

private int age;

public UnderAgeException(int age) {
	this.age = age;
}


@Override
public String toString() {
	return "UnderAgeException: Age " + age +
			" is under the required limit.";
}
}
```

Custom Exceptions 2

```
public class AgeValidator {
private static final int MINIMUM_AGE = 18;


public void checkAge(int age) throws UnderAgeException {
        if (age < MINIMUM_AGE) {
                throw new UnderAgeException(age);
        } else {
                System.out.println("Access granted.");
        }
}

public static void main(String[] args) {
        AgeValidator validator = new AgeValidator();
        try {
                validator.checkAge(16);
        } catch (UnderAgeException e) {
                e.printStackTrace();
        }
}
}
```

Output

```
UnderAgeException: Age 16 is under the required limit.
at custom_exception_example.AgeValidator.checkAge(AgeValidator.java:9)
at custom_exception_example.AgeValidator.main(AgeValidator.java:18)
```

Summary *1*

- Exceptions separate error handling from mainline code
- Methods can be labeled with the exceptions they throw
- Unhandled exceptions rise up call stack and cause program exit
- Checked exceptions force caller to either handle exception or propagate to the next caller

Summary *2*

- Code in *finally* clause always executed
  - whether or not an exception is handled or propagated
- Exceptions can be chained