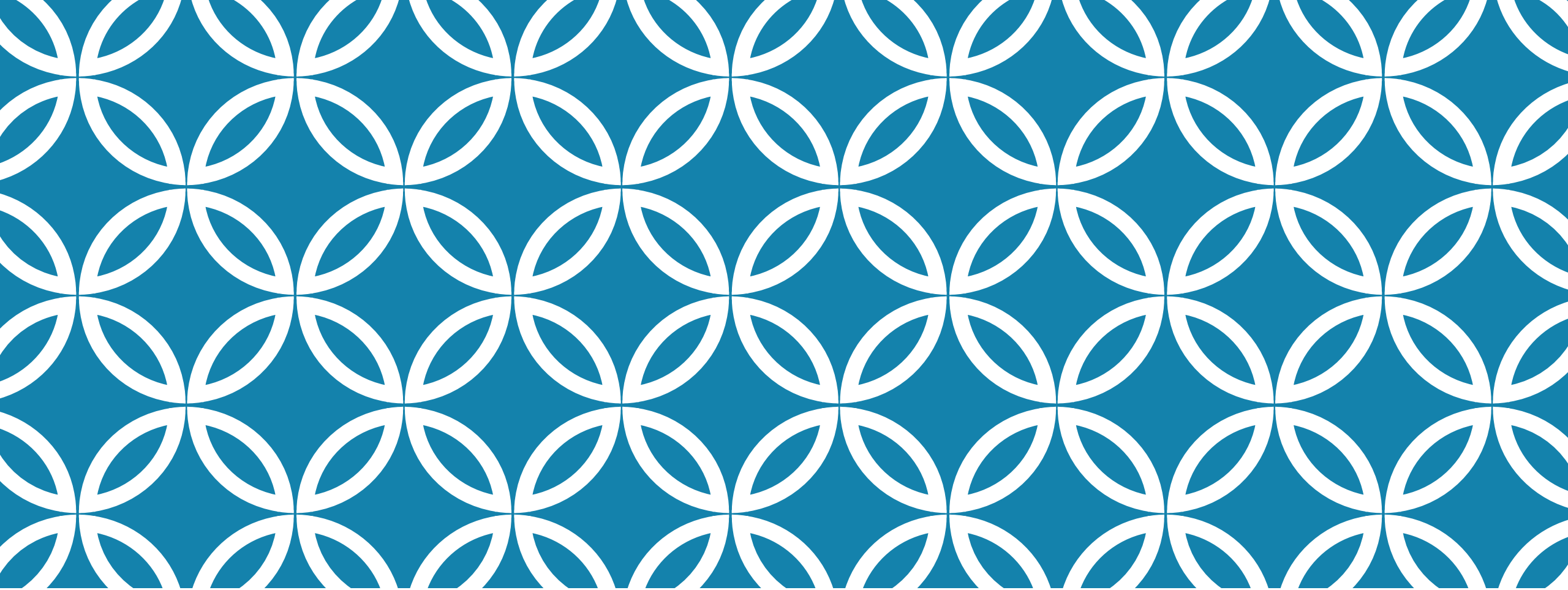


# CHAPTER 10 EXCEPTION HANDLING

# EXCEPTION-HANDLING FUNDAMENTALS

- An *exception* is an abnormal condition that arises in a code sequence at run time.
- A Java exception is *an object* that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Java exception handling is managed via five keywords:
  - try
  - catch
  - throw
  - throws
  - finally

# EXCEPTION-HANDLING FUNDAMENTALS

- This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

# EXCEPTION TYPES

- All exception types are subclasses of the built-in class **Throwable.**
- Two subclasses of Throwable:
  - Exception - used for exceptional conditions that user programs should catch
  - Error - exceptions that are not expected to be caught under normal circumstances by your program, ex: Stack overflow
- important subclass of Exception, called RuntimeException
  - Ex: division by zero, invalid array indexing

# UNCAUGHT EXCEPTIONS

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them.

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.
- we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

# UNCAUGHT EXCEPTIONS

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
        at Exc1.subroutine(Exc1.java:4)
        at Exc1.main(Exc1.java:7)
```

# USING TRY AND CATCH

- To handle an exception yourself gives two benefits:
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.

- You cannot use **try** on a single statement.

# USING TRY AND CATCH

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e)
        {
          System.out.println("Division by zero.");
          a = 0; // set a to zero and continue
        }
        System.out.println("a: " + a);
      }
    }
  }
```

# DISPLAYING A DESCRIPTION OF AN EXCEPTION

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

# MULTIPLE CATCH CLAUSES

- In some cases, more than one exception could be raised by a single piece of code.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.

```
// Demonstrate multiple catch statements.
class MultiCatch {
  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.");
  }
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

```
/*  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.");
    }
  }
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught.

# NESTED TRY STATEMENTS

- That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

```
// An example 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 an 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:42
```

# NESTED TRY STATEMENTS

- For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement.
- In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method.

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
  static void nesttry(int a) {
    try { // nested try block
      /* If one command line arg is used,
         then an 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);
}
  }
  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);

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

# throw

- it is possible for your program to throw an exception explicitly, using the **throw** statement.

- The general form of **throw** is shown here:

  throw *ThrowableInstance*;

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.

- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

- There are two ways you can obtain a **Throwable** object:
  - using a parameter in a **catch** clause, or
  - creating one with the **new** operator.

# throw

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

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

throw new NullPointerException("demo");

- Many of Java's builtin run-time exceptions have at least two constructors:
  - one with no parameter and
  - one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**.
- It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

# throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

```
// This is now correct.
class ThrowsDemo1 {
  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);
    }  }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

# finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.
- **finally** creates a block of code that will be executed after a **try/catch block** has completed and before the code following the **try/catch** block.
- The finally block will execute whether or not an exception is thrown.

# finally

- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- The **finally** clause is optional.
- However, each **try** statement requires at least one **catch** or a **finally** clause.

- In next slide, an example program that shows three methods that exit in various ways, none without executing their finally clauses

```
// Demonstrate finally.
class FinallyDemo {
  // Through 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();
  }
}
```

```
// Demonstrate finally.
class FinallyDemo {
  // Through 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();
  }
}
```

Here is the output generated by the preceding program:

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

# finally

**REMEMBER** *If a* `finally` *block is associated with a* `try`*, the* `finally` *block will be executed upon conclusion of the* `try`*.*

# 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**.
- As previously explained, these exceptions need not be included in any method's throws list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- those exceptions defined by **java.lang** that must be included in a **method's throws list** if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.

| Exception | Meaning |
|---|---|
| ArithmeticException | Arithmetic error, such as 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. |
| 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. |
| 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. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| TypeNotPresentException | Type not found. |
| UnsupportedOperationException | An unsupported operation was encountered. |

**TABLE 10-1** Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

| 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. |

**TABLE 10-2** Java's Checked Exceptions Defined in **java.lang**

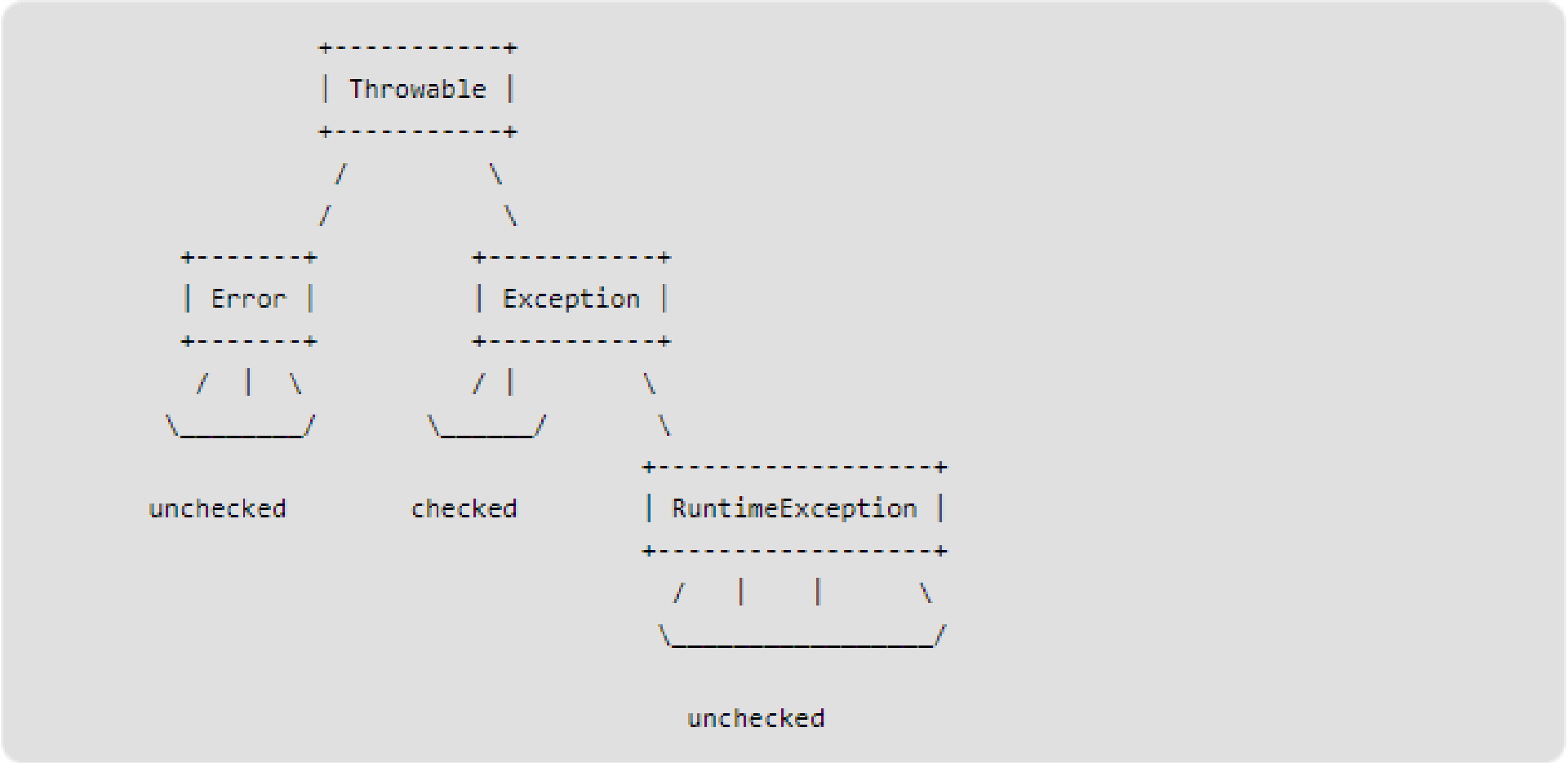
# CHECKED EXCEPTION

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Here: FileReader() throws a checked exception *FileNotFoundException.*

Reference: http://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

# UNCHECKED EXCEPTION




Reference: http://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

# UNCHECKED EXCEPTION

```
class Main {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

Reference: http://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

# Creating Your Own Exception Subclasses

- To create a own exception, just define a subclass of **Exception** (which is, of course, 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.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

| Method | Description |
|---|---|
| Throwable fillInStackTrace( ) | Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown. |
| Throwable getCause( ) | Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. |
| String getLocalizedMessage( ) | Returns a localized description of the exception. |
| String getMessage( ) | Returns a description of the exception. |
| StackTraceElement[ ] getStackTrace( ) | Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. |
| Throwable initCause(Throwable *causeExc*) | Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to 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. |
| void setStackTrace(StackTraceElement *elements*[ ]) | Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use. |
| String toString( ) | Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object. |

**TABLE 10-3** The Methods Defined by **Throwable**

# Creating Your Own Exception Subclasses

- You may also wish to override one or more of these methods in exception classes that you create.

- **Exception** defines four constructors.
  - Exception( ) → has no description
  - Exception(String msg) → has description of the exception

  **Note:** two constructors are explained in next topic (chained exceptions)

- override **toString( ):**
  - The version of **toString**( ) defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By **overriding toString(** ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

```
// 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);
    }
  }
}
```

the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

# Chained Exceptions

- This feature allows you to associate another exception with an exception.
- This second exception describes the cause of the first exception.
- For example,
  - imagine a situation in which a method throws an **ArithmeticException** because of an attempt to **divide by zero**. However, the ***actual cause of the problem was that an I/O error occurred***, which caused the divisor to be set improperly.
- To allow chained exceptions, two constructors and two methods were added to **Throwable**.
- The constructors are shown here:
  - Throwable(Throwable causeExc) → that causes the current exception
  - Throwable(String msg, Throwable causeExc) → the underlying reason that an exception occurred
- These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

# Chained Exceptions

- The chained exception methods added to **Throwable** are **getCause( )** and **initCause( ).**

- Throwable getCause( )
  - It returns the exception that underlies the current exception
  - If there is no underlying exception, **null** is returned.

- Throwable initCause(Throwable causeExc)
  - It associates *causeExc* with the invoking exception and returns a reference to the exception
  - you can call **initCause( )** only once for each exception object.
  - if the cause exception was set by a constructor, then you can't set it again using **initCause( ).**
  - In general, **initCause( )** is used to set a cause for legacy exception classes that *don't support the two additional constructors described earlier*.

```
// Demonstrate exception chaining.
class ChainExcDemo {
  static void demoproc() {
    // create an exception
    NullPointerException e =
      new NullPointerException("top layer");

    // add a cause
    e.initCause(new ArithmeticException("cause"));

    throw e;
  }
```

```
  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      // display top level exception
      System.out.println("Caught: " + e);

      // display cause exception
      System.out.println("Original cause: " +
                         e.getCause());
    }
  }
}
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

# Using Exceptions

- Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- 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.
- One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

# DISCLAIMER

- These slides are not original and have been prepared from various sources for teaching purpose.

Sources:

- Herbert Schildt, Java™: The Complete Reference

***Thank You***