COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

Exception Handling

- Framework provided by Java to handle "problematic situations" gracefully
- Program continues to execute if possible
  - Doesn't just stop/crash if something goes wrong
- Separates the concepts:
  - Errors that are unrecoverable, versus
  - Exceptions that should be handled properly by program

Phases in Exception Handling

- Typical scenario:
  - Problem occurs
  - Catch and Handle exception (associated class hierarchy)
- An exception is an object that describes the problem

Dealing with Exceptions

- Java has a predefined set of exceptions that can occur during execution
- Ways to deal with an exception:
  1. Ignore it
  2. Handle it where it occurs
  3. Handle it an another place in the program
- How you handle it is an important design consideration

Ignoring Exceptions

- This means exception is not caught
- Program ends with appropriate message
- Message includes a call to the stack trace that:
  - Indicates the line on which the exception occurred
  - Shows the trail of method calls leading to the attempted execution of the erroneous line

Example of Stack Trace

- Short example:

```
Exception in thread "main" java.lang.IllegalStateException: A book has a null property
        at com.example.myproject.Author.getBookIds(Author.java:38)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
Caused by: java.lang.NullPointerException
        at com.example.myproject.Book.getId(Book.java:22)
        at com.example.myproject.Author.getBookIds(Author.java:35)
        ... 1 more
```

- Note:
  - Trail of method calls
  - Originating line where problem occurred

```
Caused by: java.lang.NullPointerException <-- root cause
        at com.example.myproject.Book.getId(Book.java:22) <-- important line
```

Zero.java

```
public class Zero
{
   public static void main( String[] args )
   {
      int numerator = 10;
      int denominator = 0;

      System.out.println( numerator / denominator );

      System.out.println( "This text will not be printed." );
   }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Zero.main(Zero.java:8)
```

Anticipating Problem in Zero.java

```
public class Zero
{
  public static void main (String[] args)
  {
    int numerator = 10;
    int denominator = 0;

    try
    {
      System.out.println( numerator / denominator );
    }
    catch( ArithmeticException exception )
    {
      System.out.println( "denominator cannot be 0 but is "  + denominator );
    }
    System.out.println( "This text will not be printed." );
  }
}
```

new

Anticipating Problem in Zero.java

```
public class Zero
{
  public static void main (String[] args)
  {
    int numerator = 10;
    int denominator = 0;

    try
    {
      System.out.println( numerator / denominator );
    }
    catch( ArithmeticException exception )
    {
      System.out.println( "denominator cannot be 0 but is "  + denominator );
    }
    System.out.println( "This text will not be printed." );
  }
}
```

new

Output

```
Denominator cannot be 0 but is 0
This text will not be printed.
```

The `try` Statement

- To handle exception where it occurs, use a `try` statement
- Syntax:
  - A `try` block
  - One or more `catch` blocks
    - One `catch` per anticipated exception
  - Optionally, a `finally` block
- Program goes from normal execution flow to exception execution flow

Earlier Example

- One `catch`, no `finally`:

```
try
{
    System.out.println( numerator / denominator );
}
catch( ArithmeticException exception )
{
    System.out.println( "denom cannot be 0 but is "
    + denom );
}
```

Earlier Example

- Thrown exception changes control flow

```
try
{
    System.out.println( numerator / denominator );
    System.out.println( "another line" );
}
catch( ArithmeticException exception )
{
    System.out.println( "denom cannot be 0 but is "
    + denom );
}
```

- "another line" won't be printed

ProductCodes.java

```
public class ProductCodes
{
   public static void main (String[] args)
   {
      String code;
      char zone;
      int district, valid = 0, banned = 0;
      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter product code (XXX to quit): ");
      code = scan.nextLine();
      while (!code.equals ("XXX"))
      {
          // parses code and handles exceptions for "bad" codes
         System.out.print ("Enter product code (XXX to quit): ");
         code = scan.nextLine();
      }
      System.out.println ("# of valid codes entered: " + valid);
      System.out.println ("# of banned codes entered: " + banned);
   }
}
```

details:

Possible code:
TRV2475A5R-14

parse code + handle exceptions

```
while (!code.equals ("XXX"))
{
   try
   {
      zone = code.charAt(9);   ?
      district = Integer.parseInt(code.substring(3, 7));
      valid++;
      if (zone == 'R' && district > 2000)
         banned++;
   }
   catch (StringIndexOutOfBoundsException exception)
   {
      System.out.println ("Improper code length: " + code);
   }
   catch (NumberFormatException exception)
   {
      System.out.println ("District is not numeric: " + code);
   }
   System.out.print ("Enter product code (XXX to quit): ");
   code = scan.nextLine();
}
System.out.println ("# of valid codes entered: " + valid);
System.out.println ("# of banned codes entered: " + banned);
```

details:

```
while (!code.equals ("XXX"))
{
   try
   {
      zone = code.charAt(9);   ?
      district = Integer.parseInt(code.substring(3, 7));
      valid++;
      if (zone == 'R' && district > 2000)
         banned++;
   }
   catch (StringIndexOutOfBoundsException exception)
   {
      System.out.println ("Improper code length: " + code);
   }
   catch (NumberFormatException exception)
   {
      System.out.println ("District is not numeric: " + code);
   }
   System.out.print ("Enter product code (XXX to quit): ");
   code = scan.nextLine();
}
System.out.println ("# of valid codes entered: " + valid);
System.out.println ("# of banned codes entered: " + banned);
```

Possible code:
TRV2475A5R-14

At most one block is executed

details:

```
while (!code.equals ("XXX"))
{
   try
   {
      zone = code.charAt(9);
      district = Integer.parseInt(code.substring(3, 7));
      valid++;
      if (zone == 'R' && district > 2000)
         banned++;
   }
   catch (StringIndexOutOfBoundsException exception)
   {
      System.o
   }
   catch (Numb
   {
      System.o
   }
   System.out.
   code = scan
}
System.out.pri
System.out.pri
```

?

parse code + handle exceptions

Possible code:
TRV2475A5R-14

Sample Run

```
Enter product code (XXX to quit): TRV2475A5R-14
Enter product code (XXX to quit): TRD1704A7R-12
Enter product code (XXX to quit): TRL2k74A5R-11
District is not numeric: TRL2k74A5R-11
Enter product code (XXX to quit): TRQ2949A6M-04
Enter product code (XXX to quit): TRV2105A2
Improper code length: TRV2105A2
Enter product code (XXX to quit): TRQ2778A7R-19
Enter product code (XXX to quit): XXX
# of valid codes entered: 4
# of banned codes entered: 2
```

General Structure

```
try
{
     // code statements
     // throw exception here
     // these lines not reached if exception thrown above
}
catch( Exception e1 )
{
     // execute these lines only
     // if e1 was caught
}
catch( Exception e2 )
{
     // execute these lines only
     // if e2 was caught
} // may have more catch blocks for more exceptions
finally
{
     // always executed
     // regardless of exception thrown or not
}
```

Handling Exceptions Elsewhere

- Sometimes you want to catch an exception at a higher level method call instead
- In this case, if an exception occurs:
  - Exception follows the call stack
  - And propagate up through the method calls until
    - Exception is caught, or
    - Exception reaches `main()`

Example: Propagation.java

```
public class Propagation
{
   public static void main (String[] args)
   {
      ExceptionScope demo = new ExceptionScope();

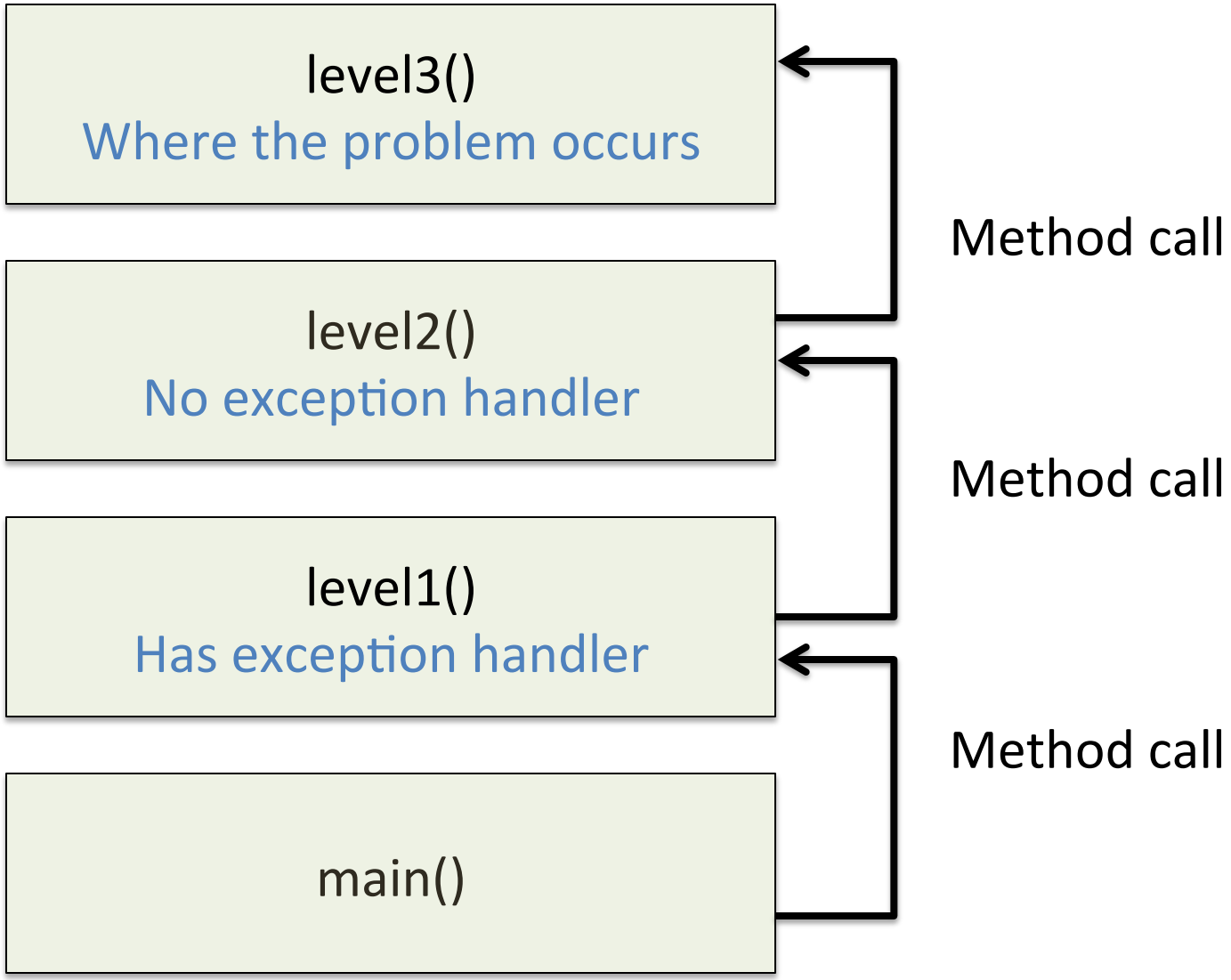
      System.out.println("Program beginning.");
      demo.level1();
      System.out.println("Program ending.");
   }
}
```

```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (ArithmeticException e)
    {
      System.out.println (" Exception message is: " + e.getMessage());
      System.out.println ("Call stack trace:");
      e.printStackTrace();
    }
    System.out.println("Level 1 ending.");
  }
  public void level2()
  {
    System.out.println("Level 2 beginning.");
    level3 ();
    System.out.println("Level 2 ending.");
  }
  public void level3 ()
  {
    int numerator = 10;
    int denominator = 0;
    System.out.println("Level 3 beginning.");
    int result = numerator / denominator;
    System.out.println("Level 3 ending.");
  }
}
```

Output

?

Method Call Stack



```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (
    {
      Syste
      Syste
      e.pri
    }
    System.
  }
  public vo
  {
    System.
    level3
    System.
  }
  public vo
  {
    int num
    int den
    System.
    int res
    System.
  }
}
```

Output

```
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.

The exception message is: / by zero

The call stack trace:
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level3(ExceptionScope.java:54)
    at ExceptionScope.level2(ExceptionScope.java:41)
    at ExceptionScope.level1(ExceptionScope.java:18)
    at Propagation.main(Propagation.java:17)

Level 1 ending.
Program ending.
```

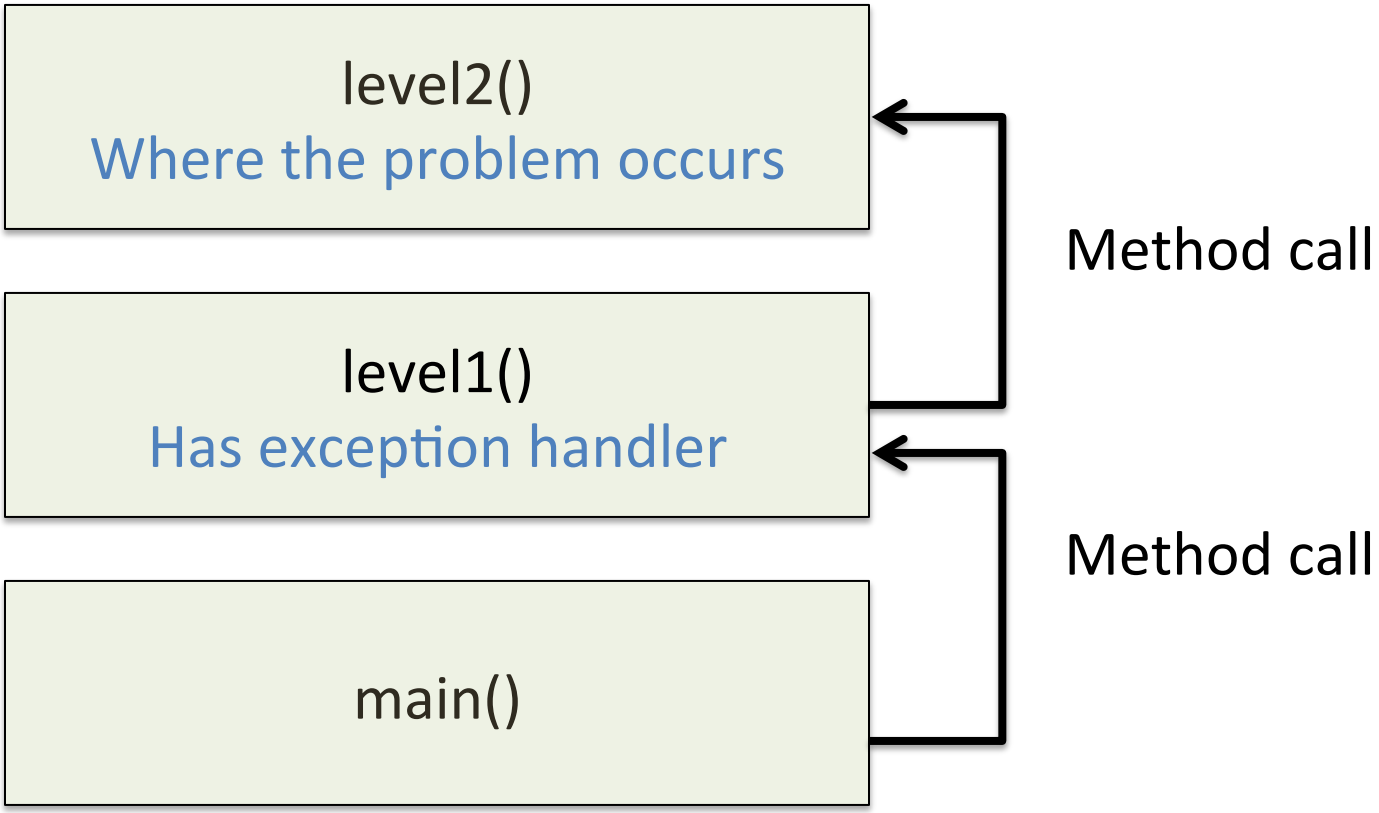
```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (ArithmeticException e)
    {
      System.out.println (" Exception message is: " + e.getMessage());
      System.out.println ("Call stack trace:");
      e.printStackTrace();
    }
    System.out.println("Level 1 ending.");
  }
  public void level2()
  {
    System.out.println("Level 2 beginning.");
    // ADD NEW LINES HERE
    int num = 10;
    int den = 0;
    int rez = num / den;
    level3 ();
    System.out.println("Level 2 ending.");
  }
  public void level3 () // same definition
  { … }
}
```

new

Output

?

Method Call Stack



```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (
    {
      Syste
      Syste
      e.pri
    }
    System.
  }
  public vo
  {
    System.
    // ADD
    int num
    int den
    int rez
    level3
    System.
  }
  public vo
  { … }
}
```

Output

```
Program beginning.
Level 1 beginning.
Level 2 beginning.

The exception message is: / by zero

The call stack trace:
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level2(ExceptionScope.java:41)
    at ExceptionScope.level1(ExceptionScope.java:18)
    at Propagation.main(Propagation.java:17)

Level 1 ending.
Program ending.
```

```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (ArithmeticException e)
    {
      System.out.println (" Exception message is: " + e.getMessage());
      System.out.println ("Call stack trace:");
      e.printStackTrace();
    }
    System.out.println("Level 1 ending.");
  }
  public void level2()
  {
    System.out.println("Level 2 beginning.");
    level3 ();
    // ADD NEW LINES HERE
    int num = 10;
    int den = 0;
    int rez = num / den;
    System.out.println("Level 2 ending.");
  }
  public void level3 () // same definition
  { … }
}
```

switch order

Output

?

```
public class ExceptionScope
{
  public void level1()
  {
    System.out.println("Level 1 beginning.");
    try
    {
      level2();
    }
    catch (
    {
      Syste
      Syste
      e.pri
    }
    System.
  }
  public vo
  {
    System.
    level3
    // ADD
    int num
    int den
    int rez
    System.
  }
  public vo
  { … }
}
```

switch order

Output

```
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.

The exception message is: / by zero

The call stack trace:
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level3(ExceptionScope.java:54)
    at ExceptionScope.level2(ExceptionScope.java:41)
    at ExceptionScope.level1(ExceptionScope.java:18)
    at Propagation.main(Propagation.java:17)

Level 1 ending.
Program ending.
```

Where to Handle Exception

```
public void calculate() {
   try{
      // some code
   }
   catch( Exception e ){
      // handle the exception
   }
}
```

- vs.

```
public void calculate() {// no try-catch inside }
public void calcMore()
{
   try
   {
      calculate();
   }
   catch( Exception e )
   {
      // handle exception
   }
}
```

Where to Handle Exception

```
public void calculate() {
   try{
      // some code
   }
   catch( Exception e ){
      // handle the exception
   }
}
```

Handles exception
in place:
Method where problem appears
decides how best to handle it

- vs.

```
public void calculate() {// no try-catch inside }
public void calcMore()
{
   try
   {
      calculate();
   }
   catch( Exception e )
   {
      // handle exception
   }
}
```

Exception propagation:
Caller decides how best
to handle exception

When to Handle Exceptions?

- Depends on situation

- Example for when you want your method to handle the exception:
  - You want to open a file with a certain name and write some information to it
  - If file not found, you want to create a new file
  - Rest of code continues to work

- Example for when you want the caller to handle the exception:
  - You want to open a file with a certain file name and read some information from it
  - If file not found, you want the caller to give you another file name
  - Rest of code continues to work

When to Use Exception Handling

- When you foresee a problem may occur at run-time and know possible solution to it while programming
  - Problem should be recoverable so program execution can continue
  - Solution can't be too different from original purpose
- Example:
  - Open file with read and write permissions
  - Open file with read permissions, write to another file

Scenario

- Suppose you have code:

```
// open the file
// determine the size of the file
// allocate that much memory
// read everything from file into memory
// close the file
// do some calculations with info from file
```

- What could go wrong?

Scenario

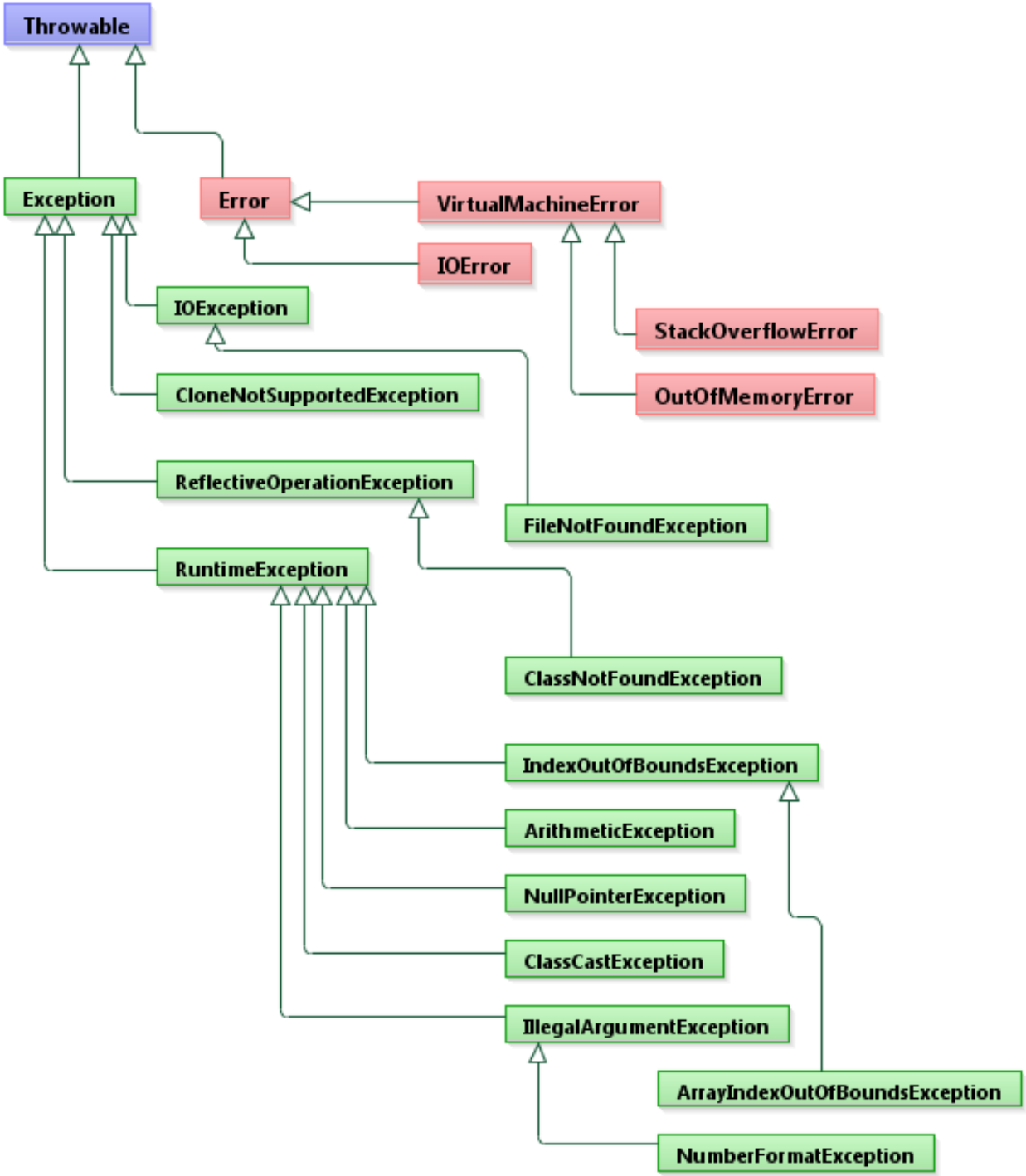
- Suppose you have code:

```
// open the file
// determine the size of the file
// allocate that much memory
// read everything from file into memory
// close the file
// do some calculations with info from file
```

- What could go wrong?
  - Can't open file
  - Can't determine length of file
  - Computer doesn't have enough memory
  - Fails in reading from file
  - Can't close file

Types of Exceptions

- Exception classes in Java are related by inheritance
  - Forms an exception class hierarchy
- All error and exception classes are descendents of `Throwable` class
- Programmers can define new types of exceptions
  - Extend the `Exception` class or its descendant
  - Choose parent class based on how the new exception will be used


Throwable
Exception
Error
VirtualMachineError
IOError
IOException
StackOverflowError
CloneNotSupportedException
OutOfMemoryError
ReflectiveOperationException
FileNotFoundException
RuntimeException
ClassNotFoundException
IndexOutOfBoundsException
ArithmeticException
NullPointerException
ClassCastException
IllegalArgumentException
ArrayIndexOutOfBoundsException
NumberFormatException

Creating Your Own Exceptions

- Advanced technique – simple example here only
- Steps:
  - Use `throws` to declare a type of exception will be thrown
  - Create an exception object
  - Anticipate problem
  - Use `throw` to intentionally throw exception
  - Where problem occurs: may want to handle exception

Example

- Creating a new exception class:

```
public class OutOfRangeException extends Exception
{
  public OutOfRangeException( String message )
  {
    super( message );
  }
}
```

- Using it (next slide)

```
public class CreatingExceptions
{
  public static void main( String[] args ) throws OutOfRangeException
  {
    final int MIN = 25, MAX = 40;

    Scanner scan = new Scanner( System.in );

    OutOfRangeException problem =
      new OutOfRangeException( "Input value is out of range.” );

    System.out.print( "Enter an integer value between " + MIN +
                      " and " + MAX + ", inclusive: " );
    int value = scan.nextInt();

    //  Determine if the exception should be thrown
    if( value < MIN || value > MAX )
      throw problem;

    // may never reach
    System.out.println( "End of main method.” );
  }
}
```

Output

?

```
public class CreatingExceptions
{
  public static void main( String[] args ) throws OutOfRangeException
  {
    final int MIN = 25, MAX = 40;

    Scanner scan = new Scanner( System.in );

    OutOfRangeException problem =
      new OutOfRangeException( "Input value is out of range.” );

    System.out.print( "Enter an integer value between " + MIN +
                      " and " + MAX + ", inclusive: " );
    int value = scan.nextInt();

    //  Determine if the exception should be thrown
    if( value < MIN || value > MAX )
      throw problem;

    // ma
    Syste
  }
}
```

Sample Run

```
Enter an integer value between 25 and 40, inclusive: 69
Exception in thread "main" OutOfRangeException:
    Input value is out of range.
    at CreatingExceptions.main(CreatingExceptions.java:20)
```

```
public class CreatingExceptions
{
  public static void main( String[] args ) throws OutOfRangeException
  {
    final int MIN = 25, MAX = 40;

    Scanner scan = new Scanner( System.in );

    OutOfRangeException problem =
      new OutOfRangeException( "Input value is out of range." );

    System.out.print( "Enter an integer value between " + MIN +
                      " and " + MAX + ", inclusive: " );
    int value = scan.nextInt();

    //  Determine if the exception should be thrown
    if( value < MIN || value > MAX )
      throw problem;

    // may never reach
    System.out.println( "End of main method." );
  }
}
```

create exception

anticipate it

throw it

Phases in Exception Handling v2

- Typical scenario:
  - Problem occurs
  - Handle exception (associated class hierarchy)

- Customized Scenario – creating your own:
  - Problem may occur
  - Create and throw exception
  - May still handle exception

General Structure

- Define your own exception class
- ```
  public returnType methodName() throws Exception
  {
    // create new exception
    Exception problem = new …

    // some code

    // anticipate scenario and throw exception
    if( … )
      throw problem;

    // lines not reached if exception thrown above
  }
  ```
  ```

Summary of Exceptions

- Anticipating problems while programming and handling them during execution
- Exception is an object that describes the problem
- Ways to deal with an exception:
  1. Ignore it
  2. Handle it where it occurs
  3. Handle it an another place in the program (via propagation)
- Exception class hierarchy relates various exception classes
- Custom exceptions can be created
- New reserved words:
  - `try`, `catch`, `finally`
  - `throws`, `throw`