Exception Handling

CSCI 201

Principles of Software Development

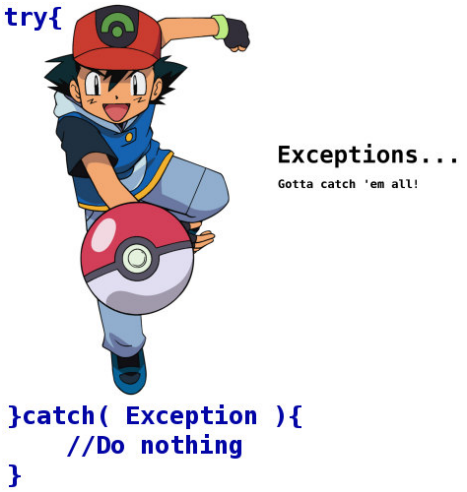
Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

USC Viterbi
School of Engineering

Outline

- Exception Handling
- Program

Exception Handling



- An exception is an indication of a problem that occurs during a program's execution
  - Exceptions should not be part of the normal execution of a program
  - Exception handling is on the order of 10 times slower compared to writing good code to avoid an exception
- In Java, we can handle most errors that would cause our programs to terminate prematurely using exception handling

`try-catch` Blocks

- Any code that has the potential to throw an exception should be enclosed in a `try` block
- Immediately following a `try` block are zero or more `catch` blocks
  - In parentheses after the word `catch` will be the exception parameter that represents the type of exception the catch handler can process
  - The exception parameter will also include the variable name representing the type of exception that is being handled
- An optional `finally` clause can be included after all of the `catch` blocks (or after the `try` block if there are no catch blocks)
  - The `finally` clause will be executed regardless of whether an exception was thrown in the `try` block or not, regardless whether it was handled by a `catch` block
  - Even if an thrown exception was not handled by a `catch` clause, the `finally` clause will still execute

Exception Handling Example

```
1  public class Test {
2    public void foo(int [] arr) {
3      try {
4        for (int i=0; i < 5; i++) {
5          arr[i] = i;
6        }
7      } catch (ArrayIndexOutOfBoundsException aioobe) {
8        System.out.println(“aioobe: " + aioobe.getMessage());
9      } finally {
10       System.out.println(“in finally block”);
11     }
12   }
13 }
14
15 public class Main {
16   public void bar() {
17     int myArr[] = new int[5];
18     foo(myArr);
19   }
20 }
```

Exception Handling Example

```
1  public class Test {
2    public void foo(int [] arr) {
3      try {
4        for (int i=0; i < 5; i++) {
5          arr[i] = i;
6        }
7      } catch (ArrayIndexOutOfBoundsException aioobe) {
8        System.out.println("aioobe: " + aioobe.getMessage());
9      } finally {
10       System.out.println("in finally block");
11     }
12   }
13 }
14
15 public class Main {
16   public void bar() {
17     int myArr[] = new int[3];
18     foo(myArr);
19   }
20 }
```

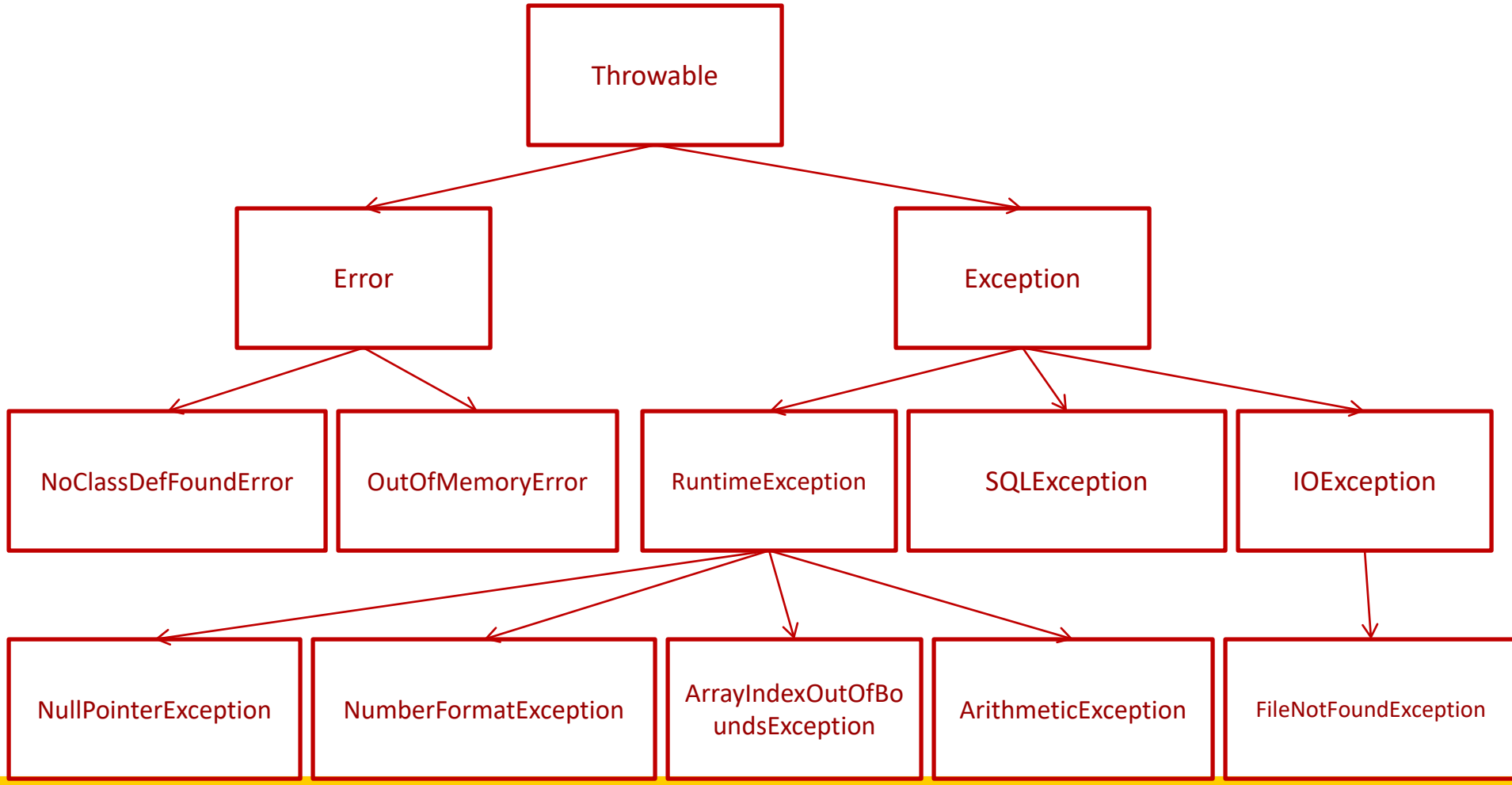
What happens if the size of the array is 3?

Exception Handling Example

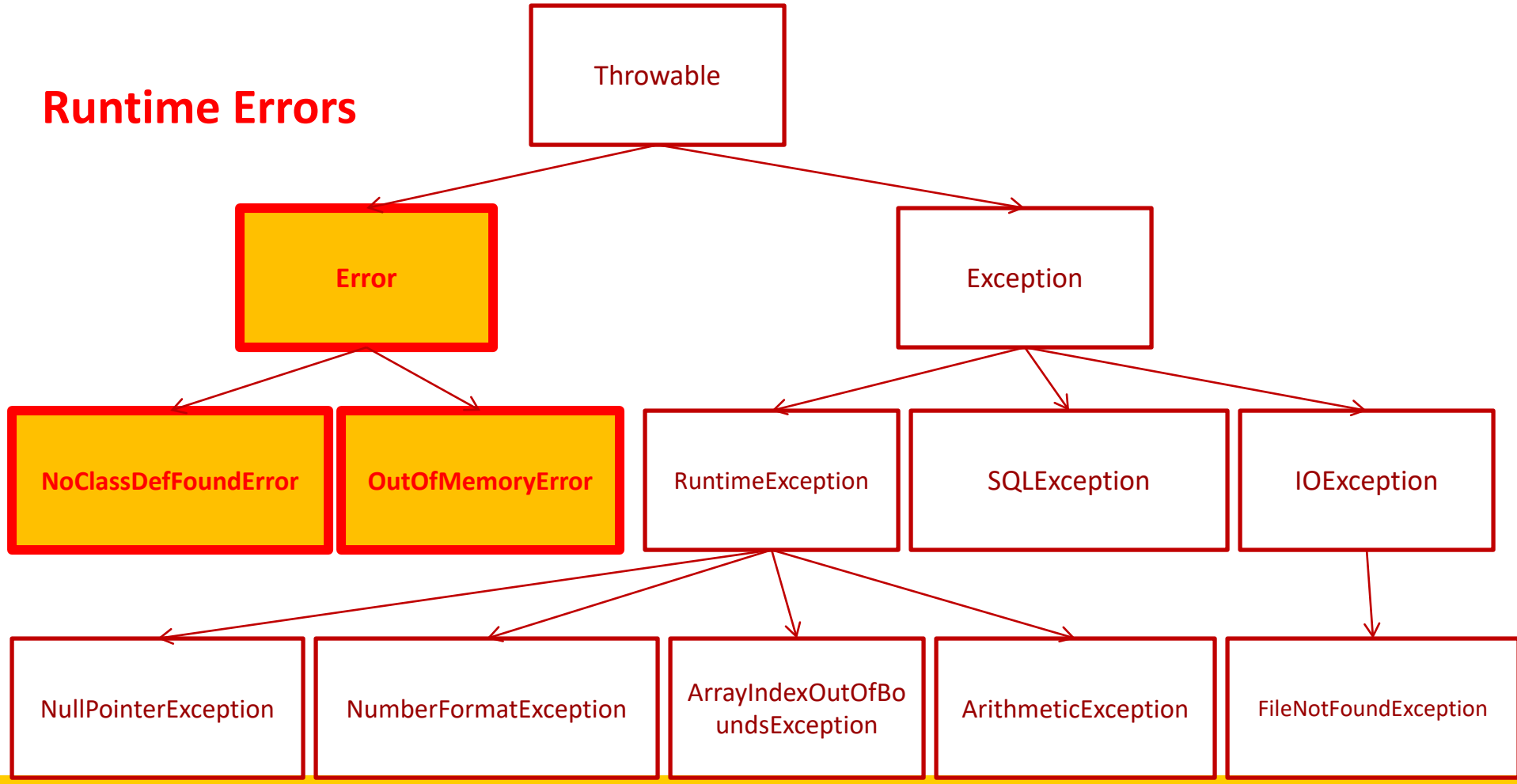
```
1  public class Test {
2    public void foo(int [] arr) {
3      try {
4        for (int i=0; i < 5; i++) {
5          arr[i] = i;
6        }
7      } catch (ArrayIndexOutOfBoundsException aioobe) {
8        System.out.println("aioobe: " + aioobe.getMessage());
9      } finally {
10       System.out.println("in finally block");
11     }
12   }
13 }
14
15 public class Main {
16   public void bar() {
17     foo(null);
18   }
19 }
```

What happens if the array passed in is null?

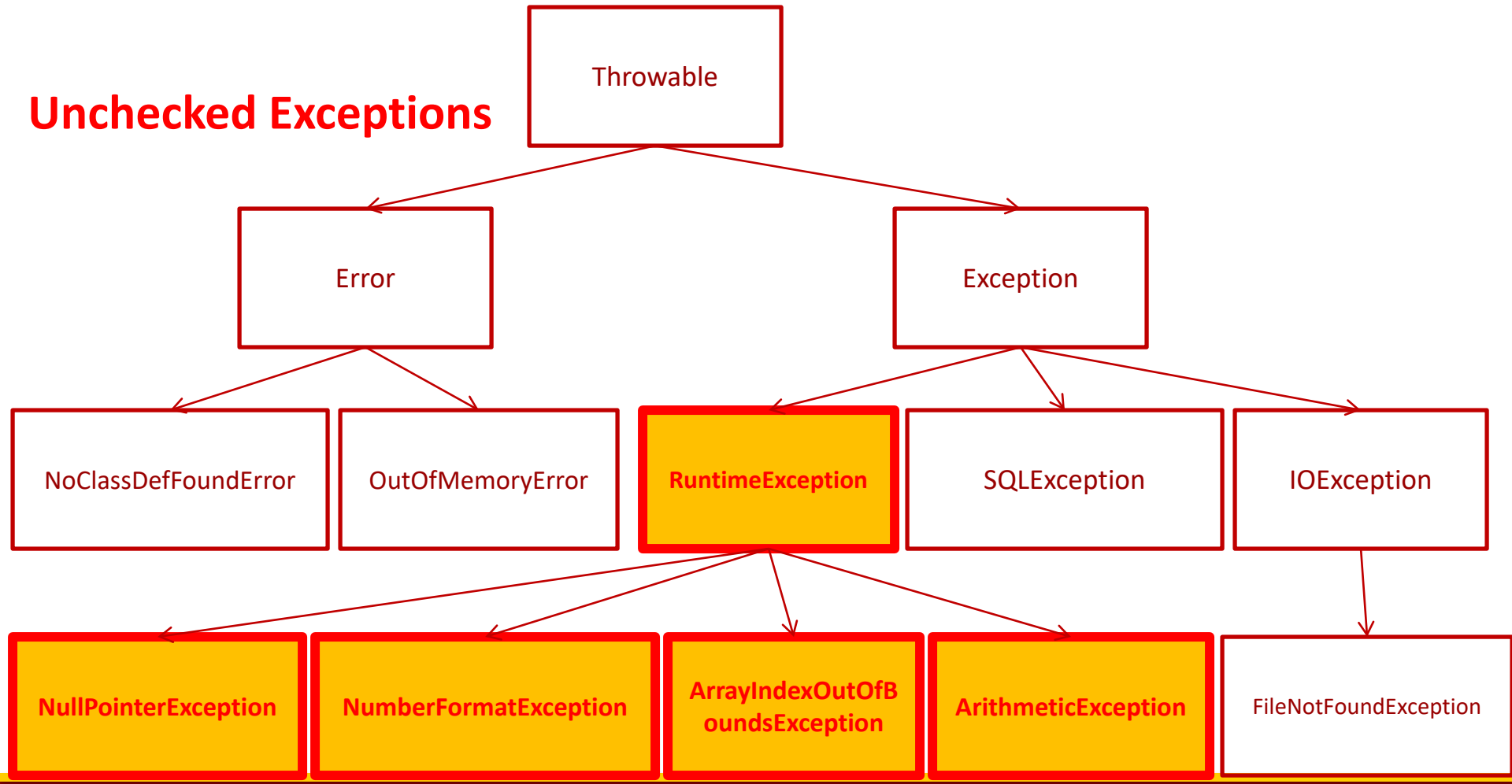
Java Exception Inheritance Hierarchy



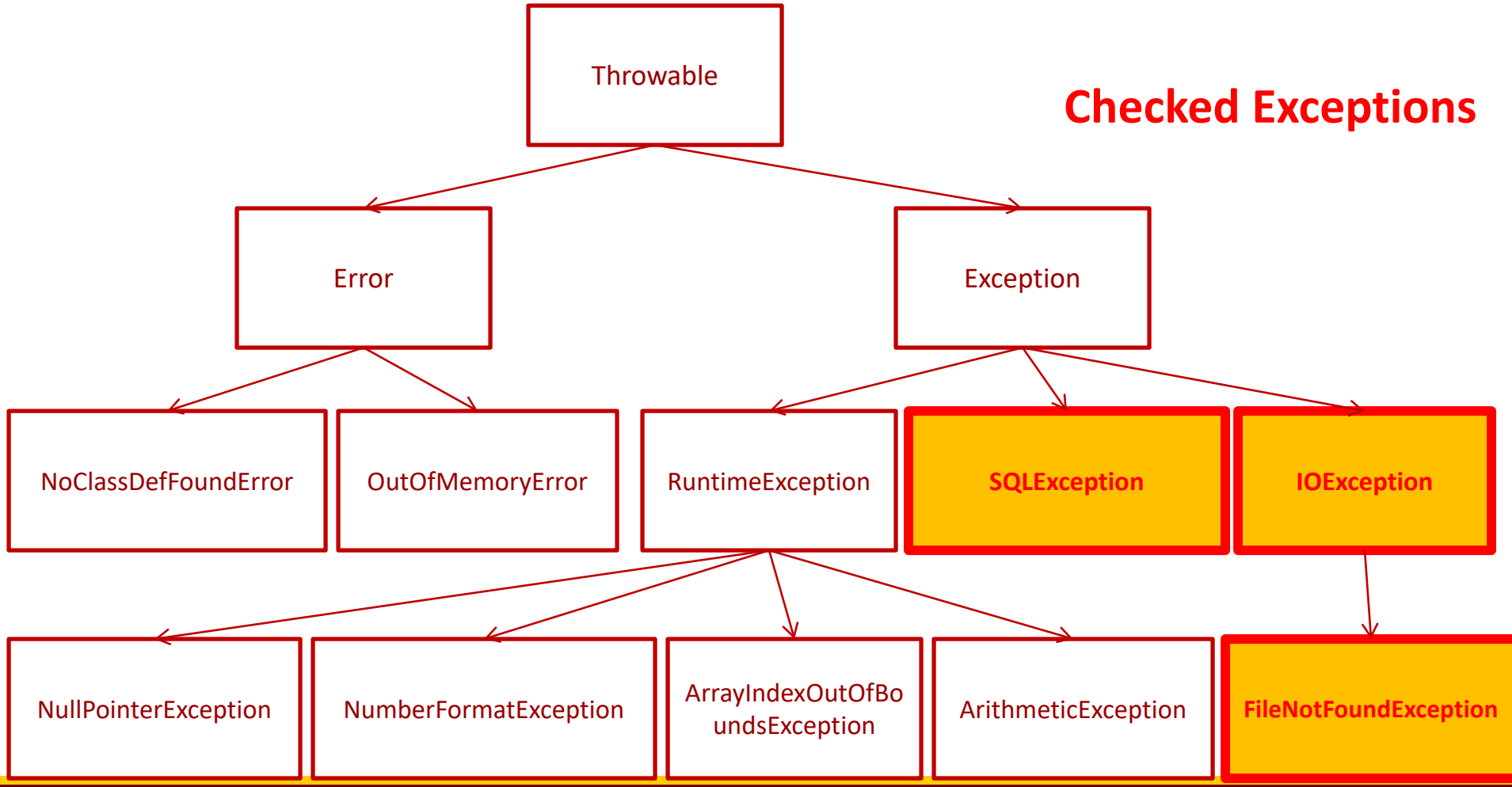
Java Exception Inheritance Hierarchy



Java Exception Inheritance Hierarchy



Java Exception Inheritance Hierarchy



Checked vs Unchecked Exceptions

- **Unchecked** exceptions do not have to be handled in a `try-catch` block
  - Exceptions that inherit from `RuntimeException` are **unchecked** exceptions
  - These exceptions can usually be avoided by good coding practices

- **Checked** exceptions must be handled in a `try-catch` block
  - Exceptions that do not inherit from `RuntimeException` are **checked** exceptions

- Note: Classes that inherit from `Error` are not exceptions and cannot be handled through exception handling

FileNotFoundException and IOException

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  public class Test {
6   public static void main(String [] args) {
7    FileReader fr;
8    BufferedReader br;
9    try {
10    fr = new FileReader("Test.java");
11    br = new BufferedReader(fr);
12    String line = br.readLine();
13    System.out.println("#1:" + line);
14    if (line != null) {
15     line = br.readLine();
16     System.out.println("#2:" + line);
17    }
18   } catch (FileNotFoundException fnfe) {
19     System.out.println(fnfe.getMessage());
20   } catch (IOException ioe) {
21     System.out.println(ioe.getMessage());
22   } finally {
23    if (br != null) {
24      try {
25        br.close();
26      } catch (IOException ioe) {
27        System.out.print(ioe.getMessage());
28      }
29    }
30    if (fr != null) {
31      try {
32        fr.close();
33      } catch (IOException ioe) {
34        System.out.print(ioe.getMessage());
35      }
36     }
37   } // ends finally
38  } // ends main
39 } // ends Test
```

FileNotFoundException and IOException

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  public class Test {
6   public static void main(String [] args) {
7    FileReader fr;
8    BufferedReader br;
9    try {
10    fr = new FileReader("Test.java");
11    br = new BufferedReader(fr);
12    String line = br.readLine();
13    System.out.println("#1:" + line);
14    if (line != null) {
15     line = br.readLine();
16     System.out.println("#2:" + line);
17    }
18   } catch (IOException ioe) {
19     System.out.println(ioe.getMessage());
20   } catch (FileNotFoundException fnfe) {
21     System.out.println(fnfe.getMessage());
22   } finally {
23    if (br != null) {
24      try {
25        br.close();
26      } catch (IOException ioe) {
27        System.out.print(ioe.getMessage());
28      }
29    }
30    if (fr != null) {
31      try {
32        fr.close();
33      } catch (IOException ioe) {
34        System.out.print(ioe.getMessage());
35      }
36     }
37   } // ends finally
38  } // ends main
39 } // ends Test
```

This won't compile because `IOException` handles the `FileNotFoundException`, so the `FileNotFoundException` block is an unreachable block of code

FileNotFoundException and IOException

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  public class Test {
6   public static void main(String [] args) {
7    FileReader fr;
8    BufferedReader br;
9    try {
10    fr = new FileReader("Test.java");
11    br = new BufferedReader(fr);
12    String line = br.readLine();
13    System.out.println("#1:" + line);
14    if (line != null) {
15     line = br.readLine();
16     System.out.println("#2:" + line);
17    }
18   } catch (IOException ioe) {
19     System.out.println(ioe.getMessage());
20   // } catch (FileNotFoundException fnfe) {
21   //   System.out.println(fnfe.getMessage());
22   } finally {
23    if (br != null) {
24      try {
25        br.close();
26      } catch (IOException ioe) {
27        System.out.print(ioe.getMessage());
28      }
29    }
30    if (fr != null) {
31      try {
32        fr.close();
33      } catch (IOException ioe) {
34        System.out.print(ioe.getMessage());
35      }
36     }
37    } // ends finally
38  } // ends main
39 } // ends Test
```

This compiles but is not good programming practice because the `IOException` won't give us as much information about the problem as the more specific `FileNotFoundException`

FileNotFoundException and IOException

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  public class Test {
6   public static void main(String [] args) {
7    FileReader fr;
8    BufferedReader br;
9    try {
10    fr = new FileReader("Test.java");
11    br = new BufferedReader(fr);
12    String line = br.readLine();
13    System.out.println("#1:" + line);
14    if (line != null) {
15     line = br.readLine();
16     System.out.println("#2:" + line);
17    }
18   } catch (Exception e) {
19     System.out.println(e.getMessage());
20   // } catch (FileNotFoundException fnfe) {
21   //   System.out.println(fnfe.getMessage());
22   } finally {
23    if (br != null) {
24      try {
25        br.close();
26      } catch (IOException ioe) {
27        System.out.print(ioe.getMessage());
28      }
29    }
30    if (fr != null) {
31      try {
32        fr.close();
33      } catch (IOException ioe) {
34        System.out.print(ioe.getMessage());
35      }
36     }
37   } // ends finally
38  } // ends main
39 } // ends Test
```

This compiles but also is not good programming practice because the Exception will just give a message that says “Exception occurred”

Throwing Exceptions

- Exceptions can be thrown inside of `try` blocks if you desire
- If a `catch` block handles that exception, it will
  - If it doesn't, the exception will be thrown up to the calling method
  - If no `catch` block handles that exception, the program will crash

Java `Exception` Class

- The `Exception` class is defined in the `java.lang` package, so you do not need to import that class
  - Other exceptions are in other packages, so they will need to be imported
- `Exception` defines a constructor that takes a `String` representing the error message
- There is also an inherited method `getMessage()` that returns an exception object's error message (which is what was passed into the constructor of the `Exception`)
  - This method can be overridden to return a specialized error message if the `String` entered into the constructor is not sufficient

Creating a Custom Exception

- You are able to create a custom exception class by inheriting from an existing exception
  - You can inherit from `Exception` if you want the exception to be required to be checked
- The `String` passed into the constructor of the `Exception` class will be the `String` returned from the `getMessage()` method
  - The `getMessage()` method can be overridden as well if a custom message is to be returned
- Use the keyword `throw` to specify when an exception will be thrown
- Use the keyword `throws` to specify that a method has the ability to throw an exception

Creating a Custom Exception Example

```
1  class DivideByZeroException extends Exception  {
2    public DivideByZeroException(String message) {
3      super(message);
4    }
5  }
6  public class Test {
7    public double divide(double num, double den) throws DivideByZeroException {
8      if (den == 0) {
9        throw new DivideByZeroException("divide by zero");
10     }
11     return num / den;
12   }
13   public static void main(String [] args) {
14     try {
15       Test t = new Test();
16       double quotient = t.divide(100, 0);
17     } catch (DivideByZeroException dbze) {
18       System.out.println("DivideByZeroException: " + dbze.getMessage());
19     }
20   }
21 }
```

Overriding `getMessage()` Example

```
1  class DivideByZeroException extends Exception  {
2    private double numer;
3    private double denom;
4    public DivideByZeroException(String message, double numer, double denom) {
5      super(message);
6      this.numer = numer;
7      this.denom = denom;
8    }
9    public String getMessage() {
10     return "Dividing " + numer + "/" + denom + " " + super.getMessage();
11   }
12 }
13 public class Test {
14   public double divide(double numer, double denom) throws DivideByZeroException {
15     if (denom == 0) {
16       throw new DivideByZeroException("divide by zero", numer, denom);
17     }
18     return numer / denom;
19   }
20   public static void main(String [] args) {
21     try {
22       Test t = new Test();
23       double quotient = t.divide(100, 0);
24     } catch (DivideByZeroException dbze) {
25       System.out.println("DivideByZeroException: " + dbze.getMessage());
26     }
27   }
28 }
```

Outline

- Exception Handling
- Program

Program

- Write a program that prompts the user to enter two numbers. If the first number is not less than the second number, throw a custom exception called `NumberGreaterException`. Handle that exception in the main method and display the value of `getMessage()` to the user to match the output below.

```
c:\>java csci201.NumberExceptions
Enter the first number: 100
Enter the second number: 50
NumberGreaterException: 50 is not greater than 100.
c:\>
```