CSE201: Advanced Programming

Lecture 10: Exception Handling
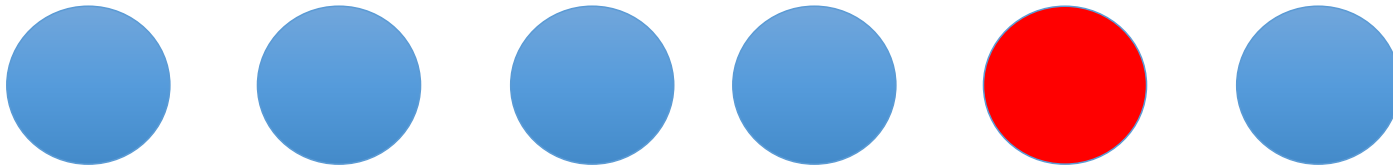
Vivek Kumar

Computer Science and Engineering
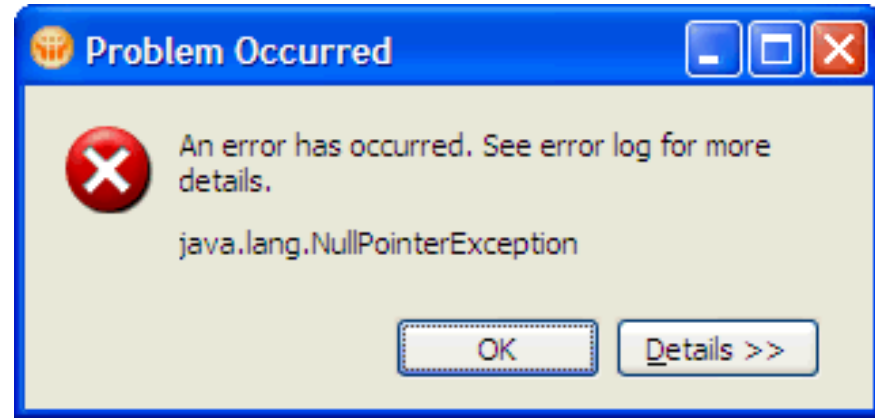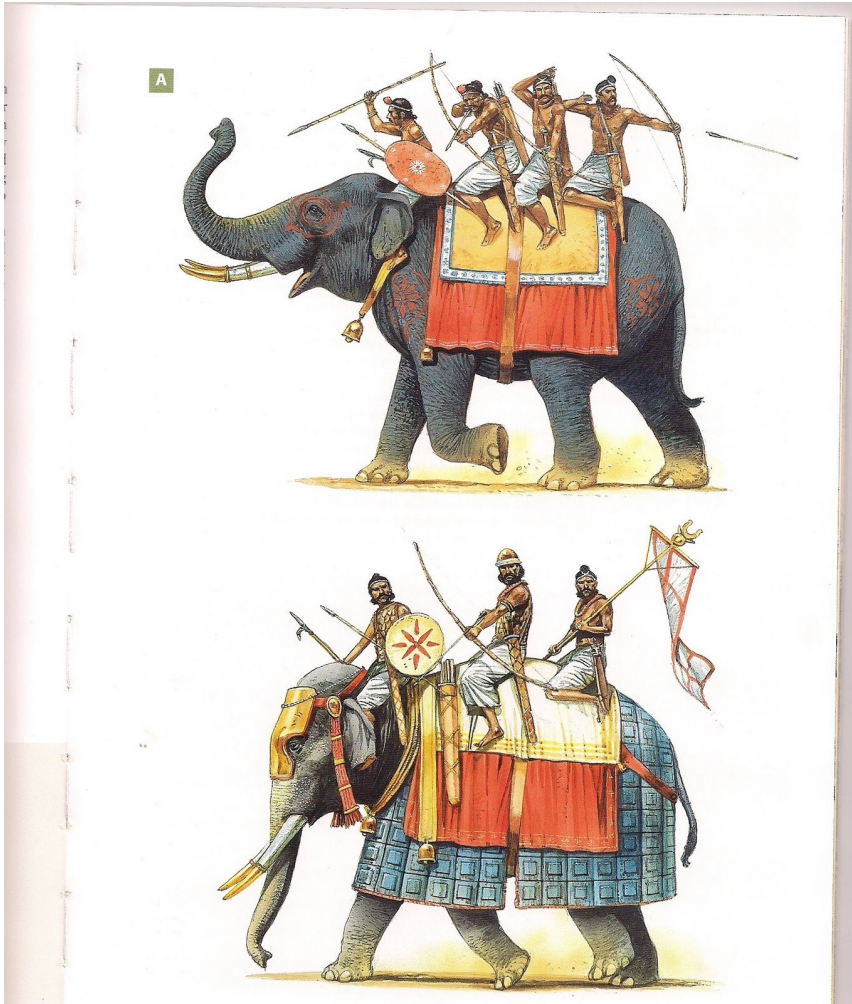
IIIT Delhi

vivekk@iiitd.ac.in

Today's Lecture: Exceptions

Being Defensive is Important

Problem Occurred

An error has occurred. See error log for more details.

java.lang.NullPointerException

OK   Details >>

© Vivek Kumar

www.shutterstock.com · 109665452

3

Defensive Programming

- Murphy's law
  - "Anything that can possibly go wrong, does."

- Finagle's law
  - "Anything that can go wrong, will – at the worst possible moment."

- Sod's law
  - "If something can go wrong, it will"

Defensive programming: Hope for the best, expect the worst!

Defensive Programming

- Collection of techniques to reduce the risk of failure at run time

  o An analogy is defensive driving by being never sure how other drivers would be driving

- The technique is in making the software behave in a predictable manner despite unexpected inputs or user actions and internal errors

  o After all debugging takes a lot of time!

Types of Programming Errors

- Syntax errors
  - Compile time errors
  - Easiest to fix

- Logical errors
  - Program runs without crashing but gives incorrect result
  - Most difficult to fix

- Runtime errors
  - Occur while the program is running if the environment detects an operation that is impossible to carry out
  - Could be fixed easily with defensive programming
    - **Exception handling!**

Exception Handling Syntax

- Process for handling exceptions
  - `try` some code, catch exception thrown by tried code, finally, "clean up" if necessary
  - `try`, `catch`, and `finally` are reserved words
- `try` denotes code that may throw an exception
  - place questionable code within a `try` block
  - a `try` block must be immediately followed by a `catch` block unlike an if w/o else
  - thus, `try-catch` blocks always occurs as pairs
- `catch` exception thrown in `try` block and write special code to handle it
  - catch blocks distinguished by type of exception
  - can have several **catch blocks**, each specifying a particular type of exception
  - Once an exception is handled, execution continues after the catch block
- `finally` (optional)
  - special block of code that is executed whether or not an exception is thrown
  - follows *catch block*

7

Trace a `try/catch` Program Execution (1/3)

```
try {
    statements;

}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

Trace a try/catch Program Execution (2/3)

```
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;

}

Next statement;
```

The final block is always executed

Trace a `try/catch` Program Execution (3/3)

```
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
```

Next statement in the method is executed

Trace a try/catch Program Execution (1/4)

```
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
```

> Suppose an exception of type Exception1 is thrown in statement2

Trace a `try/catch` Program Execution (2/4)

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}


Next statement;
```

The exception is handled.

Trace a `try/catch` Program Execution (3/4)

```
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a `try/catch` Program Execution (4/4)

```
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

Is this Defensive Programming ?

```java
import java.util.*;
public class Main {

    public static void main(String[] args) {


        System.out.println("Enter Integer Input");

            Scanner sc = new Scanner(System.in);
            int num = sc.nextInt();




    }
  }
}
```

- Is program correct?
  - Yes
    - But, only if the user is paying attention
      - Invalid input ?
      - String as input?

15

Exception Handling using try/catch

```java
import java.util.*;
public class Main {

    public static void main(String[] args) {
        boolean done = false;
        while(!done) {
            System.out.println("Enter Integer Input");
            try {
                Scanner sc = new Scanner(System.in);
                int num = sc.nextInt(); //exception
point    done = true;
            }
            catch(InputMismatchException inp) {
                System.out.println("Wrong input:");
                System.out.println("Try again");
            }
            finally {
                System.out.println("Always execute");
            }
        }
    }
}
```

- This is a foolproof program now!

- Exception handling using try/catch block of statements
  - Defensive programming

- InputMismatchException is a type of exception provided by the Scanner class in Java

© Vivek Kumar

16

Multiple catch Blocks

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] s = {"a", "23", null, "4", "P"};
        int sum = 0;
        for(int i=0; i<10; i++) {

            sum += (s[i].length() > 0) ?
                    Integer.parseInt(s[i]) : 0;




    }
}
```

17

Multiple catch Blocks

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] s = {"a", "23", null, "4", "P"};
        int sum = 0;
        for(int i=0; i<10; i++) {
            try {
                sum += (s[i].length() > 0) ?
                        Integer.parseInt(s[i]) : 0;
            }
            catch(NumberFormatException e) {
                System.out.println("Not an Integer");
            }
            catch(NullPointerException e) {
                System.out.println("NULL value
found");
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Index not in
range");
            }
        }
    }
}
```

© Vivek Kumar

- There could be multiple catch for a single try block

- They are designed to catch different types of exceptions that could be raised from a single try block

- **How the exceptions are generated here?**
  - i=0 will raise NumberFromatException
  - i=2 will raise NullPointerException
  - i=4 will raise NumberFormatException
  - i>4 will raise ArrayIndexOutOfBounds exception

18

Question

```java
public class Main {
    public static void main(String[] args) {
        String s = null;
        try {
            int length = s.length();
        }

        System.out.println("Just before catch block");

        catch(NullPointerException e) {
            System.out.println("String was null");
        }
    }
}
```

- What is the output of the following program?

- Answer
  - Compilation error!
  - **No statement is allowed between a pair of try and catch**
  - error: 'catch' without 'try'

19

Nested try/catch Blocks

```
public class Andy {
    .....

    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);
            System.out.println("Wendy is fired!");
            try {
                _water = johny.getADrink();
                int volume = _water.getVolume();
            }
            catch(NullPointerException e) {
                this.fire(johny);
                System.out.println("Johny is fired!");
            }
        }
    }
}
```

- **try/catch** block could be nested!
  - If Andy's call to getADrink from Wendy returns null, he can ask Johny to getADrink

© Vivek Kumar

20

Methods Can throw Exception

```
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            System.out.println("Wendy is fired!");
            throw new NullPointerException("NO Water");
        }
    }
}
```

- If you wish to throw an exception in your code you use the **throw** keyword

- Most common would be for an unmet precondition

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it:

  throw new TheException("Message");

- In the above constructor call for the exception, the message is optional but it's always good to pass some meaningful message

© Vivek Kumar

21

Re-throwing Exception

```java
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);
            System.out.println("Wendy is fired!");
            throw new NullPointerException("NO Water");
        }
    }
}
```

© Vivek Kumar

- The caught exceptions can be re-thrown using **throw** keyword

- Re-thrown exception must be handled some where in the program, otherwise program will terminate abruptly

Trace a `try/catch` Program Execution (1/4)

```
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a `try/catch` Program Execution (2/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a `try/catch` Program Execution (3/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

Trace a `try/catch` Program Execution (4/4)
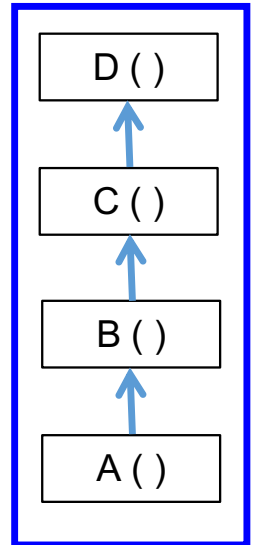
```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```
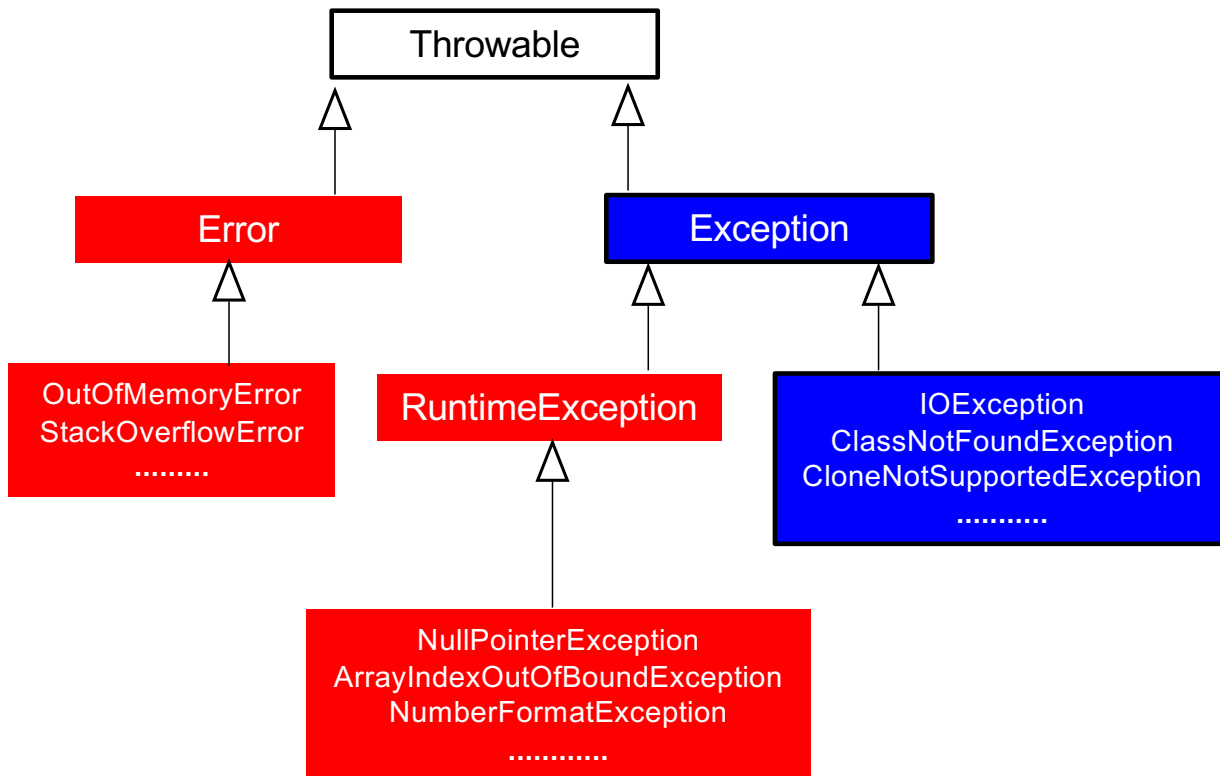
Rethrow the exception and control is transferred to the caller

How Exceptions are Handled by JVM

- Any method invocation is represented as a "**stack frame**" on the Java "**stack**"
  - **Callee-Caller** relationship
    - If method A calls method B then A is **caller** and B is **callee**
  - Each frame stores local variables, input parameters, return values and intermediate calculations
    - In addition, each frame also stores an "**exception table**"
    - This exception table stores information on each `try/catch/finally` block, i.e. the instruction offset where the `catch/finally` blocks are defined
  - When an exception is thrown, JVM does the following:
    1. Look for exception handler in current stack frame (method)
    2. If not found, then terminate the execution of current method and go to the callee method and repeat step 1 by looking into callee's exception table
    3. If no matching handler is found in any stack frame, then JVM finally terminates by throwing the stack trace (`printStackTrace` method)

```
D()
 ↑
C()
 ↑
B()
 ↑
A()
```

27

Exception Hierarchy

```
                    ┌─────────────┐
                    │  Throwable  │
                    └─────────────┘
                       △       △
              ┌────────┘       └────────┐
        ┌─────────┐              ┌─────────────┐
        │  Error  │              │  Exception  │
        └─────────┘              └─────────────┘
             △                    △            △
             │          ┌─────────┘            └─────────┐
  ┌──────────────────┐  ┌──────────────────┐  ┌──────────────────────────┐
  │ OutOfMemoryError │  │ RuntimeException  │  │      IOException          │
  │ StackOverflowError│ └──────────────────┘  │ ClassNotFoundException     │
  │ ..........        │          △            │ CloneNotSupportedException │
  └──────────────────┘          │            │ ...........                │
                                 │            └──────────────────────────┘
                     ┌───────────────────────────┐
                     │   NullPointerException     │
                     │ ArrayIndexOutOfBoundException│
                     │  NumberFormatException     │
                     │  ............              │
                     └───────────────────────────┘
```

- `Exceptions` are classes that extends `Throwable`
- Come in two types
  - **Checked exceptions**
    - Those that must be handled somehow (we will see soon)
      - E.g., `IOException` – file reading issue
  - **Unchecked exceptions**
    - Those that do not
      - E.g., `RuntimeExceptions` that is caused due to programming errors
      - You should not attempt to handle exceptions from subclass of `Error`
        - Rarely occurring exceptions that even if you try to handle, there is little you can do beyond notifying the user and trying to terminate the program gracefully

28

© Vivek Kumar

Handling Checked Exception (1/3)

```java
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- If we have code that tries to build a FileReader we must deal with the possibility of the exception
  - The code contains a syntax error. "unreported exception java.io.FileNotFoundException
    - must be caught or declared to be thrown

Handling Checked Exception (2/3)

```java
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- Here, there are 4 statements that can generate checked exceptions:
  o   The FileReader constructor
  o   the ready method
  o   the read method
  o   the close method

- To deal with the exceptions we can either state this method **"throws"** an Exception of the proper type or handle the exception within the method itself

© Vivek Kumar

Handling Checked Exception (3/3)

```java
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) throws
FileNotFoundException, IOException {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- It may be that we don't know how to deal with an error within the method that can generate it

- In this case we will pass the buck to the method that called us

- The keyword **throws** is used to indicate a method has the possibility of generating an exception of the stated type

- Now any method calling ours, must also throw an exception or handle it

31

© Vivek Kumar

Question

```
public class Main {
    public static void main(String[] args) {
        String s = null;
        try {
            int length = s.length();
        }

        catch (Exception e) {
            System.out.println("Catch block -1");
        }
        catch (NullPointerException e) {
            System.out.println("Catch block -2");
        }
    }
}
```

- What is the output of the following program?

- Answer
  ○ Compilation error!
  ○ **Unreachable catch block**
  ○ error: exception NullPointerException has already been caught

32

© Vivek Kumar

Some Important Methods in Throwable

String **toString()**   Returns a short description of the exception
String **getMessage()**   Returns the detail description of the exception
void **printStackTrace()** Prints the stacktrace information on the console

```
1. public class Andy {

2.     public void drinkWater() {
3.         getWater();
4.     }
5.  public void getWater() {
6.      try {
7.          _water = _wendy.getADrink();//null
8.          int volume = _water.getVolume();
9.      }
10.      catch(NullPointerException e) {
11.          e.printStackTrace();
12.      }
13.    }
14. }
```

● Output:

java.lang.NullPointerException
    at Andy.getWater(Andy.java:8)
    at
Andy.drinkWater(Andy.java:3)
    ......

© Vivek Kumar

33

Overriding Methods Having throws (1/3)

```java
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
                throws CloneNotSupportedException {

        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
    {

        System.out.println("Cloning not allowed");
    }
}
```

© Vivek Kumar

- If a method in parent class throws an exception (either checked or unchecked), then overridden implementation of that method in child class is not required to throw that exception
  - Although throwing that **same** exception in overridden method won't hurt

34

Overriding Methods Having throws (2/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {

        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
                throws CloneNotSupportedException {

        System.out.println("Cloning not allowed");
    }
}
```

- However, the reverse may/may not work

- Case-1: Overridden method throws **checked exception** but not the actual method in parent class
    - Compilation error

35

Overriding Methods Having throws (3/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {

        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
                throws RuntimeException {

        System.out.println("Cloning not allowed");
    }
}
```

© Vivek Kumar

- However, the reverse may/may not work

- Case-2: Overridden method throws **unchecked exception** but not the actual method in parent class
  - This works fine

36

Defining Your Own Exception (1/4)

```java
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NoWaterException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
}
```

© Vivek Kumar

- You can define and throw your own specialized exceptions
  - `throw new NoWaterException(…);`
- Useful for responding to special cases, not covered by pre-defined exceptions
- The class `Exception` has a method `getMessage()`.The String passed to `super` is printed to the output window for debugging when `getMessage()` is called by the user

37

Defining Your Own Exception (2/4)

```java
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NoWaterException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
}
```

- Every method that throws `Exceptions` that are not subclasses of `RuntimeException` must declare what exceptions it throws in method declaration

- `getWater()` is throwing the exception, hence it must declare that using the "`throws`" on method declaration

© Vivek Kumar

38

Defining Your Own Exception (3/4)

```java
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() throws NoWaterException {
        getWater();
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
    public static void main(String[] args) {
        Andy obj = new Andy();
        obj.drinkWater();
    }
}
```

- Any method that directly or indirectly calls getWater() must declare that it can generate NoWaterException using throws keyword
  - Not doing this generate compilation error
  - error: unreported exception NoWaterException; must be caught or declared to be thrown

39

© Vivek Kumar

Defining Your Own Exception (4/4)

```
1.public class NoWaterException extends Exception {
2.    public NoWaterException(String message) {
3.        super(message);
4.    }
5.}
6.public class Andy {
7.    public void drinkWater() throws NoWaterException {
8.        getWater();
9.    }
10.   public void getWater() throws NoWaterException {
11.       _water = _wendy.getADrink();
12.       if(_water == null) {
13.           this.fire(_wendy);
14.           throw new NoWaterException("NO Water");
15.       }
16.   }
17.   public static void main(String[] args)
18.                         throws NoWaterException {
19.       Andy obj = new Andy();
20.       obj.drinkWater();
21.   }
22.}
```

© Vivek Kumar

- This works fine, although we are not catching the NoWaterException anywhere that is again not a defensive programming!
  - Running this program with _water = null

Exception in thread "main" NoWaterException: NO Water

at Andy.getWater(Andy.java:14)

at Andy.drinkWater(Andy.java:8)

at Andy.main(Andy.java:20)

40

Pros and Cons of Exception

- Pros
  - o Cleaner code: rather than returning a boolean up chain of calls to check for exceptional cases, throw an exception!
  - o Use return value for meaningful data, not error checking
  - o Factor out error-checking code into one class, so it can be reused

- Cons
  - o Throwing exceptions requires extra computation
  - o Can become messy if not used economically
  - o Can accidentally cover up serious exceptions, such as `NullPointerException` by catching them

Next Lecture

- Assertions

- Java collection framework