



# D06

# PROGRAMMING with JAVA

## Ch15 – Exception Handling

# Chapter Goals

---

- To learn how to throw **exceptions**
- To be able to design your own exception classes
- To understand the difference between **checked** and **unchecked** exceptions
- To learn how to **catch** exceptions
- To know when and where to catch an exception

# Error Handling

- What should a method do when it detects an error condition?
- Traditional approach: The method returns a value that indicates whether it succeed or failed (error code)
  - **Problem 1:** The calling method may forget to check the return value
    - A failure notification may go completely undetected → the program keeps processing faulty information and mysteriously failing later
  - **Problem 2:** The calling method may not be able to do anything about the failure
    - The calling method must fail too and let its caller worry about it → many method calls would need to be checked for failure:

```
x.doSomething()
```

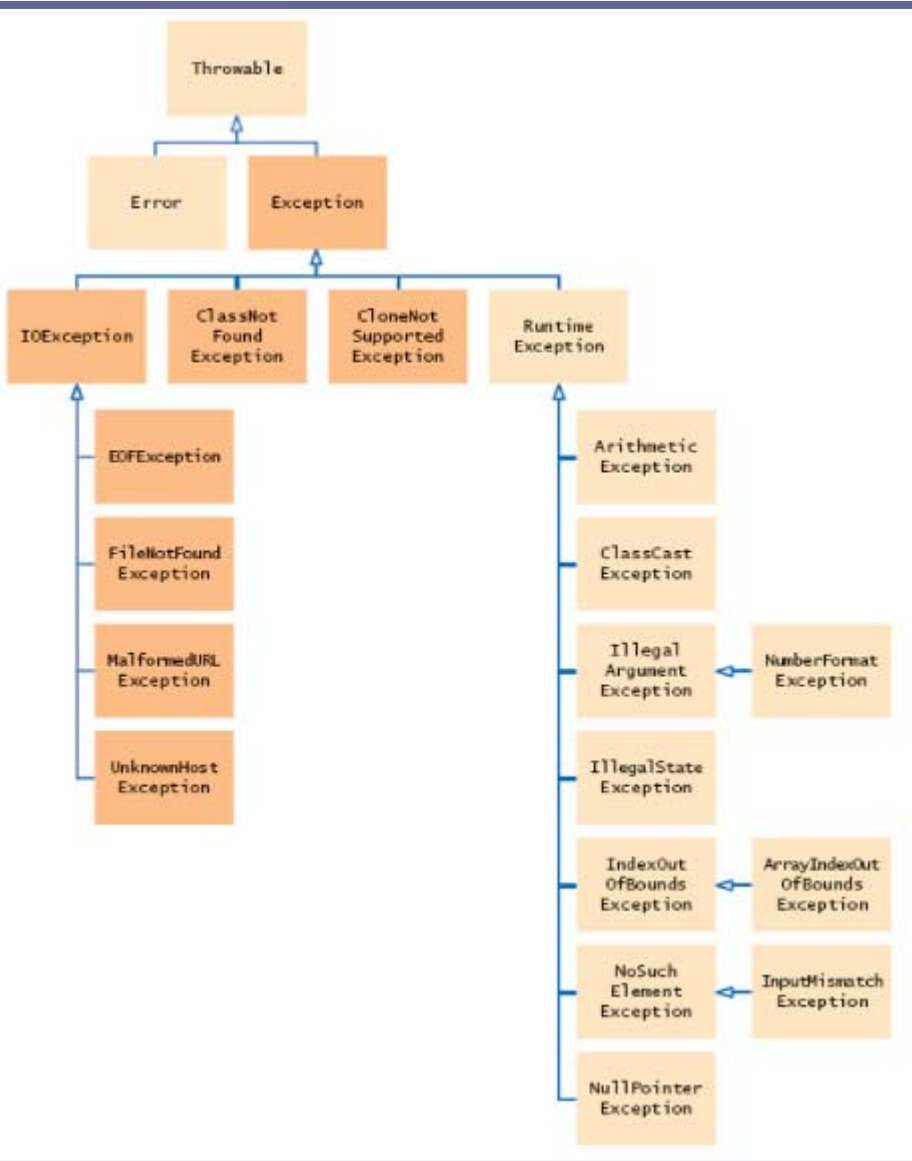
```
if (!x.doSomething()) return false;
```



The **exception handling** mechanism solves these two problems:

- **Exceptions can't be overlooked**
- **Exceptions are sent directly to an exception handler –not just the caller of the failed method**

# Throwing Exceptions



- When you detect an error condition, you just **throw** an appropriate exception object
- The Java library provides many classes to signal all sorts of exceptional conditions (Fig. 1)

Figure 1:  
The Hierarchy of Exception Classes

- Checked exceptions
- Unchecked exceptions

*Continued...*

# Throwing Exceptions

- Example:

```
public class BankAccount
{
    public void withdraw(double amount)
    {
        if (amount > balance)
        {
            IllegalArgumentException exception
                = new IllegalArgumentException("Amount
                    exceeds balance");
            throw exception;
        }
        balance = balance - amount;
    }
    . . .
}
```

- You don't have to store the exception object in a variable. You can just throw the object that the `new` operator returns:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

- When you throw an exception, execution does not continue with the next statement but with an **exception handler** (we'll see later)

# Syntax 15.1: Throwing an Exception

---

```
throw exceptionObject;
```

## Example:

```
throw new IllegalArgumentException();
```

## Purpose:

To throw an exception and transfer control to a handler for this exception type

# Self Check

---

1. How should you modify the `deposit` method to ensure that the balance is never negative?
2. Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of `balance` afterwards?

# Answers

---

1. Throw an exception if the amount being deposited is less than zero.
2. The balance is still zero because the last statement of the `withdraw` method was never executed.

# Checked and Unchecked Exceptions

- Java exceptions fall into two categories, called **checked** (due to external circumstances that the programmer cannot prevent) and **unchecked exceptions** (due to programming errors)
- When you call a method that throws a **checked** exception
  - The compiler checks that you don't ignore it
  - You must tell the compiler what you are going to do about the exception if it is ever thrown
  - E.g.: all subclasses of `IOException` are unchecked exceptions
- When you call a method that throws an **unchecked** exception
  - The compiler does not require you to keep track of unchecked exceptions
  - Exceptions, such as `NumberFormatException`, `IllegalArgumentException`, and `NullPointerException`, are unchecked exceptions. More generally, all exceptions that belong to subclasses of `RuntimeException` are unchecked, and all other subclasses of the class `Exception` are checked

*Continued...*

# Checked and Unchecked Exceptions

---

- There is a second category of internal errors that are reported by throwing objects of type **Error** (e.g.: `OutOfMemoryError`, which is thrown when all available memory has been used up). These are fatal errors that happen rarely and are beyond your control. They are too unchecked
- Why have two kinds of exceptions?
  - A checked exception describes a problem that is likely to occur at times, no matter how careful you are (e.g.: an unexpected end of file due to a disk error or a broken network connection). The compiler does insist that your program be able to handle error conditions that you cannot prevent. The majority of checked exceptions occur when you deal with input and output (i.e.: when programming with files and streams)
  - The unchecked exceptions are your fault (e.g.: a `NullPointerException` –your code was wrong when it tried to use a null reference; the compiler doesn't check whether you handle a `NullPointerException`, because you should test your references for null before using them rather than install a handler for that exception)

*Continued...*

# Checked and Unchecked Exceptions

- Example (checked): You can use the `Scanner` class to read data from a file, by passing a `FileReader` object to the `Scanner` constructor:

```
String filename = . . . ;  
FileReader reader = new FileReader(filename);  
Scanner in = new Scanner(reader);
```

However, the `FileReader` constructor can throw a `FileNotFoundException`, which is a checked exception, so you need to tell the compiler what you are going to do about it. You have two choices:

- You can handle the exception (we'll see how to do that), or
- You can tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. To do this, tag the method with a `throws` specifier. The `throws` clause signals the caller of your method that it may encounter a `FileNotFoundException` (then the caller needs to make the same decision –handle the exception, or tell its caller that the exception may be thrown):

```
public void read(String filename) throws FileNotFoundException  
{  
    FileReader reader = new FileReader(filename);  
    Scanner in = new Scanner(reader);  
    . . .  
}
```

*Continued...*

# Checked and Unchecked Exceptions

- If your method can throw checked exceptions of different types, you separate the class names by commas:

```
public void read(String filename)
    throws IOException, ClassNotFoundException
```



Always keep in mind that exception classes form an inheritance hierarchy. For example, `FileNotFoundException` is a subclass of `IOException`. Thus if a method can throw both of them, you only tag it as `throws IOException`



It sounds somehow irresponsible not to handle an exception when you know that it happened. Actually, though, it is usually best not to catch an exception if you don't know how to remedy the situation

# Syntax 15.2: Exception Specification

```
accessSpecifier returnType  
    methodName(parameterType parameterName, . . .)  
        throws ExceptionClass, ExceptionClass, . . .
```

## Example:

```
public void read(BufferedReader in) throws IOException
```

## Purpose:

To indicate the checked exceptions that this method can throw

# Self Check

---

3. Suppose a method calls the `FileReader` constructor and the `read` method of the `FileReader` class, which can throw an `IOException`. Which throws specification should you use?
4. Why is a `NullPointerException` not a checked exception?

# Answer

---

3. The specification throws `IOException` is sufficient because `FileNotFoundException` is a subclass of `IOException`.
4. Because programmers should simply check for `null` pointers instead of trying to handle a `NullPointerException`.

# Catching Exceptions

- Every exception should be handled somewhere in your program. If an exception has no handler, an error message is printed, and your program terminates → you should install exception handlers for all exceptions that your program might throw



You install an exception handler with the **try/catch** statement. Each **try** block contains one or more statements that may cause an exception. Each **catch** clause contains the handler for an exception type:

```
try
{
    String filename = . . . ;
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    String input = in.next();
    int value = Integer.parseInt(input);
    . . .
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    System.out.println("Input was not a number");
}
```

*Continued...*

# Catching Exceptions

- Three exceptions may be thrown in the former `try` block: The `FileReader` constructor can throw a `FileNotFoundException`, `Scanner.next` can throw a `NoSuchElementException`, and `Integer.parseInt` can throw a `NumberFormatException`. If any of these exceptions is actually thrown, then the rest of the instructions of the `try` block are skipped
- Here is what happens for the various exception types:
  - If a `FileNotFoundException` is thrown, then the `catch` clause for the `IOException` is executed (since `FileNotFoundException` is a subclass of `IOException`)
  - If a `NumberFormatException` occurs, then the second `catch` clause is executed
  - A `NoSuchElementException` is not caught by any of the `catch` clauses. The exception remains thrown until it is caught by another `try` block
- When the `catch (IOException exception)` block is executed, then some method in the `try` block has failed with an `IOException`. The variable `exception` contains a reference to the exception object that was thrown. The `catch` clause can analyze that object to find out more details about the failure (e.g.: you can get a printout of the chain of method calls that lead to the exception, by calling `exception.printStackTrace()`)

# Throw Early, Catch Late

---

- When a method notices a problem that it cannot solve, it is generally better to throw an exception rather than try to come up with an imperfect fix
- Conversely, a method should only catch an exception if it can really remedy the situation. Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler

 Remember: “Throw early, catch late”

# Syntax 15.3: General Try Block

```
try
{
    statement
    statement
    . . .
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    . . .
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    . . .
}
. . .
```

Continued...

# Syntax 15.3: General Try Block

## Example:

```
try
{
    System.out.println("How old are you?");
    int age = in.nextInt();
    System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
    exception.printStackTrace();
}
```

## Purpose:

To execute one or more statements that may generate exceptions. If an exception occurs and it matches one of the catch clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any catch clause, then skip the catch clauses.

# Self Check

---

5. Suppose the file with the given file name exists and has no contents. Trace the flow of execution in the `try` block in this section.
6. Is there a difference between catching checked and unchecked exceptions?

# Answers

---

5. The `FileReader` constructor succeeds, and `in` is constructed. Then the call `in.next()` throws a `NoSuchElementException`, and the `try` block is aborted. None of the `catch` clauses match, so none are executed. If none of the enclosing method calls catch the exception, the program terminates.
6. No—you catch both exception types in the same way, as you can see from the code example on page 558. Recall that `IOException` is a checked exception and `NumberFormatException` is an unchecked exception.

# The `finally` Clause

- Occasionally, you need to take some action whether or not an exception is thrown. The `finally` construct is used to handle this situation
- Example: a program can only open a finite number of files at one time → you need to close all file readers after you are done with them:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```

Suppose that one of the methods before the last line throws an exception → the call to `close` is never executed! Solve this problem by placing the call to `close` inside a `finally` clause:

# The `finally` Clause

```
FileReader reader = new FileReader(filename);
try
{
    Scanner in = new Scanner(reader);
    readData(in);
}
finally
{
    reader.close(); // if an exception occurs, finally clause
                   // is also executed before exception is
                   // passed to its handler
}
```

- In a normal case, there will no problem: when the `try` block is completed, the `finally` clause is executed, and the file is closed. However, if an exception occurs, the `finally` clause is also executed before the exception is passed to its handler



Use the `finally` clause whenever you need to do some clean up, such as closing a file, to ensure that the clean up happens no matter how the method exits

*Continued...*

# The finally Clause



It is tempting to combine `catch` and `finally` clauses, but the resulting code can be hard to understand. Instead, use a `try/finally` statement to close resources and a separate `try/catch` to handle errors:

```
try
{
    FileReader reader = new FileReader(filename);
    try
    {
        // Read input
    }
    finally
    {
        reader.close();
    }
}
catch (IOException exception)
{
    // Handle exception
}
```

# Syntax 15.4: The `finally` clause

```
try
{
    statement
    statement
    . . .
}
finally
{
    statement
    statement
    . . .
}
```

*Continued...*

# Syntax 15.4: The `finally` clause

## Example:

```
FileReader reader = new FileReader(filename);
try
{
    readData(reader);
}
finally
{
    reader.close();
}
```

## Purpose:

To ensure that the statements in the `finally` clause are executed whether or not the statements in the `try` block throw an exception.

# Self Check

---

7. Why was the `reader` variable declared outside the `try` block?
8. Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.

# Answers

---

7. If it had been declared inside the `try` block, its scope would only have extended to the end of the `try` block, and the catch clause could not have closed it.
8. The `FileReader` constructor throws an exception. The `finally` clause is executed. Since `reader` is `null`, the call to `close` is not executed. Next, a catch clause that matches the `FileNotFoundException` is located. If none exists, the program terminates.

# Designing Your Own Execution Types

- You can design your own exception types—subclasses of `Exception` or `RuntimeException`:

```
if (amount > balance)
{
    throw new InsufficientFundsException(
        "withdrawal of " + amount + " exceeds balance of "
        + balance);
}
```

```
public class InsufficientFundsException
    extends RuntimeException
{
    public InsufficientFundsException() {}

    public InsufficientFundsException(String message)
    {
        super(message);
    }
}
```

## Supply two constructors

- Default constructor
  - A constructor that accepts a message string describing reason for exception
- Make it an unchecked exception—programmer could have avoided it by calling `getBalance` first

# Self Check

---

9. What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?
10. Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

# Answers

---

9. To pass the exception message string to the `RuntimeException` superclass.
10. `Exception` or `IOException` are both good choices. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException`.

# A Complete Program

---

- Program
  - Asks user for name of file
  - File expected to contain data values
  - First line of file contains total number of values
  - Remaining lines contain the data
  - Typical input file:

```
3
1.45
-2.1
0.05
```
- What can go wrong?
  - File might not exist
  - File might have data in wrong format
- Who can detect the faults?
  - `FileReader` constructor will throw an exception when file does not exist
  - Methods that process input need to throw exception if they find error in data format

# A Complete Program

---

- What exceptions can be thrown?
  - `FileNotFoundException` can be thrown by `FileReader` constructor
  - `IOException` can be thrown by `close` method of `FileReader`
  - `BadDataException`, a custom checked exception class
- Who can remedy the faults that the exceptions report?
  - Only the `main` method of `DataSetTester` program interacts with user
    - Catches exceptions
    - Prints appropriate error messages
    - Gives user another chance to enter a correct file

# File DataSetTester.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: public class DataSetTester
06: {
07:     public static void main(String[] args)
08:     {
09:         Scanner in = new Scanner(System.in);
10:         DataSetReader reader = new DataSetReader();
11:
12:         boolean done = false;
13:         while (!done)
14:         {
15:             try
16:             {
```

*Continued...*

# File DataSetTester.java

```
17:         System.out.println("Please enter the file name: ");
18:         String filename = in.next();
19:
20:         double[] data = reader.readFile(filename);
21:         double sum = 0;
22:         for (double d : data) sum = sum + d;
23:         System.out.println("The sum is " + sum);
24:         done = true;
25:     }
26:     catch (FileNotFoundException exception)
27:     {
28:         System.out.println("File not found.");
29:     }
30:     catch (BadDataException exception)
31:     {
32:         System.out.println
            ("Bad data: " + exception.getMessage());
```

*Continued...*

# File DataSetTester.java

```
33:         }
34:         catch (IOException exception)
35:         {
36:             exception.printStackTrace();
37:         }
38:     }
39: }
40: }
```

# The `readFile` method of the `DataSetReader` class

- Constructs `Scanner` object
- Calls `readData` method
- Completely unconcerned with any exceptions (if there is a problem with the input file, it simply passes the exception to the caller)

```
public double[] readFile(String filename)
    throws IOException, BadDataException
    // FileNotFoundException is an IOException
{
    FileReader reader = new FileReader(filename);
    try
    {
        Scanner in = new Scanner(reader);
        readData(in);
    }
    finally
    {
        reader.close();
    }
    return data;
}
```

# The readData method of the DataSetReader class

- Reads the number of values
- Constructs an array
- Calls readValue for each data value

```
private void readValue(Scanner in, int i)
    throws BadDataException
{
    if (!in.hasNextDouble())
        throw new BadDataException("Data value expected");
    data[i] = in.nextDouble();
}
```

```
private void readData(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
        throw new BadDataException("Length expected");
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];

    for (int i = 0; i < numberOfValues; i++)
        readValue(in, i);

    if (in.hasNext())
        throw new BadDataException("End of file expected");
}
```

- Checks for two potential errors:
  - File might not start with an integer
  - File might have additional data after reading all values
- Makes no attempt to catch any exceptions

# Scenario

---

1. `DataSetTester.main` calls `DataSetReader.readFile`
2. `readFile` calls `readData`
3. `readData` calls `readValue`
4. `readValue` doesn't find expected value and throws `BadDataException`
5. `readValue` has no handler for exception and terminates
6. `readData` has no handler for exception and terminates
7. `readFile` has no handler for exception and terminates after executing finally clause
8. `DataSetTester.main` has handler for `BadDataException`; handler prints a message, and user is given another chance to enter file name

# File DataSetReader.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:     Reads a data set from a file. The file must have
        // the format
07:     numberOfValues
08:     value1
09:     value2
10:     . . .
11: */
12: public class DataSetReader
13: {
```

*Continued...*

# File DataSetReader.java

```
14:    /**
15:        Reads a data set.
16:        @param filename the name of the file holding the data
17:        @return the data in the file
18:    */
19:    public double[] readFile(String filename)
20:        throws IOException, BadDataException
21:    {
22:        FileReader reader = new FileReader(filename);
23:        try
24:        {
25:            Scanner in = new Scanner(reader);
26:            readData(in);
27:        }
28:        finally
29:        {
30:            reader.close();
31:        }
```

*Continued...*

# File DataSetReader.java

```
32:         return data;
33:     }
34:
35:     /**
36:      Reads all data.
37:      @param in the scanner that scans the data
38:     */
39:     private void readData(Scanner in) throws BadDataException
40:     {
41:         if (!in.hasNextInt())
42:             throw new BadDataException("Length expected");
43:         int numberOfValues = in.nextInt();
44:         data = new double[numberOfValues];
45:
46:         for (int i = 0; i < numberOfValues; i++)
47:             readValue(in, i);
```

*Continued...*

# File DataSetReader.java

```
48:
49:     if (in.hasNext())
50:         throw new BadDataException("End of file expected");
51:     }
52:
53:     /**
54:      Reads one data value.
55:      @param in the scanner that scans the data
56:      @param i the position of the value to read
57:     */
58:     private void readValue(Scanner in, int i)
59:         throws BadDataException
60:     {
61:         if (!in.hasNextDouble())
62:             throw new BadDataException("Data value expected");
63:         data[i] = in.nextDouble();
64:     }
65:     private double[] data;
66: }
```

# Self Check

---

11. Why doesn't the `DataSetReader.readFile` method catch any exceptions?
12. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

# Answers

---

11. It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user
12. `DataSetTester.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught

# Chapter Summary

---

- To signal an exceptional condition, use the **throw** statement to throw an **exception** object
- When you throw an exception, the current method terminates immediately
- There are two kinds of exceptions: **checked** and **unchecked**. Unchecked exceptions extend the class `RuntimeException` or `Error`
- **Checked** exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions
- In a method that is ready to handle a particular exception type, place the statements that can cause the exception inside a **try** block, and the handler inside a **catch** clause

*Continued*

# Chapter Summary

---

- It is better to declare that a method throws a checked exception than to handle the exception poorly
- Once a **try** block is entered, the statements in a **finally** clause are guaranteed to be executed, whether or not an exception is thrown
- You can design your own exception types –subclasses of `Exception` or `RuntimeException`