Objective

Exception Handling

- ➢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

Study chapter 11.4 of BiG Java: Early Object 6th Edition.

Error Handling

- Traditional approach: Method returns error code

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

- Problem: Forget to check for error code
  - Failure notification may go undetected
- Problem: Calling method may not be able to do anything about failure
  - Program must fail too and let its caller worry about it
  - Many method calls would need to be checked

Error Handling

Instead of programming for success

```
x.doSomething();
```

you would always be programming for failure:

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

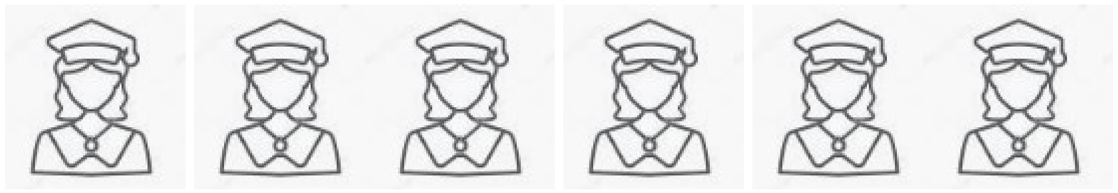
Exceptions

Security

Lost & Found

IT

Which one do you contact if there is a problem with your account



Throwing Exceptions

- ➢Exceptions:

–Can't be overlooked

–Sent <u>directly to an exception handler</u>–not just caller of failed method

- ➢Throw an exception object to signal an exceptional condition

- ➢Example: `IllegalArgumentException:`

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

Throwing Exceptions

No need to store exception object in a variable:

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

When an exception is thrown, method terminates immediately

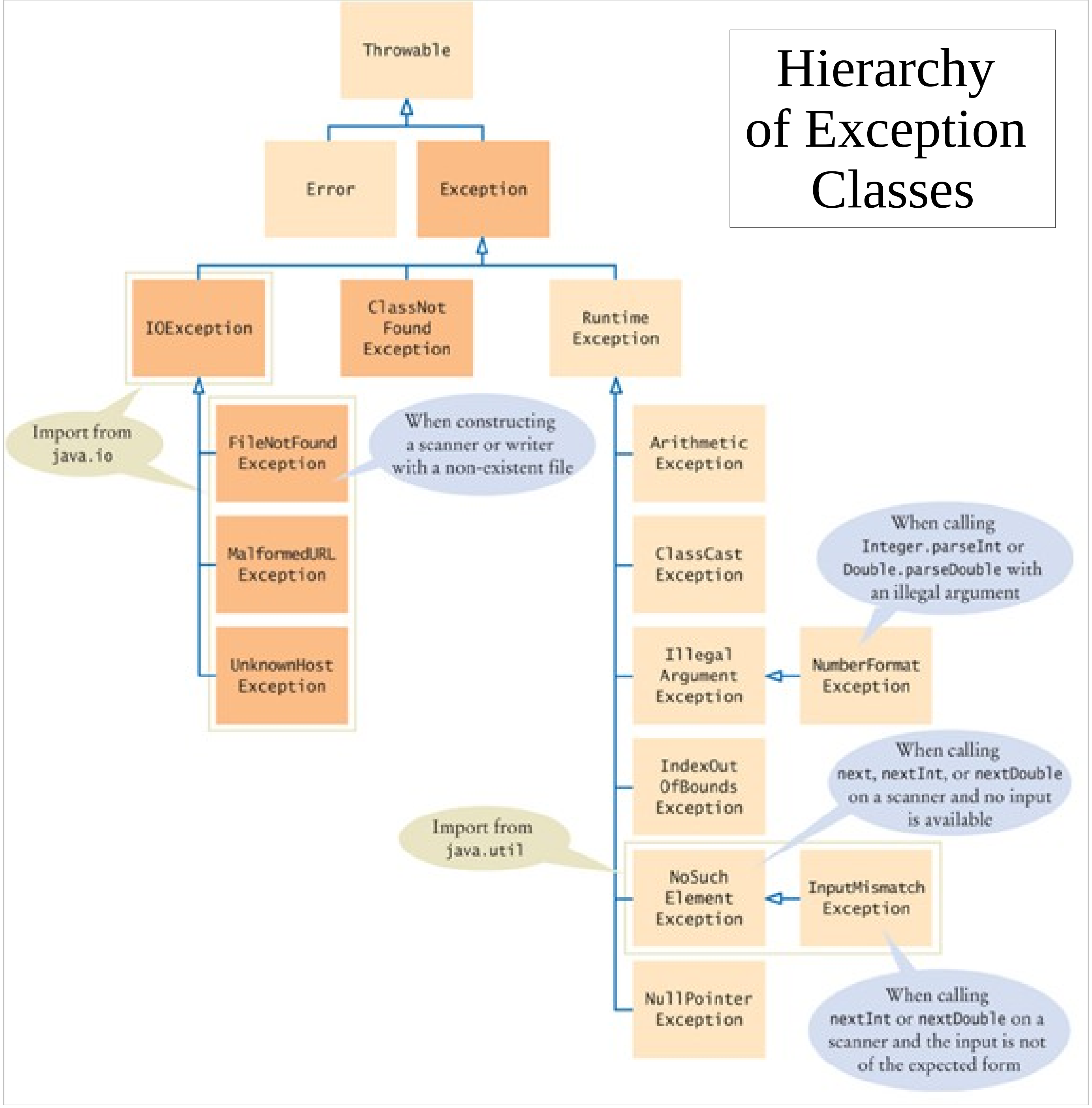
–Execution continues with an exception handler

Example

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

    }
    balance = balance - amount;
  }
  . . .
}
```

Hierarchy of Exception Classes


Throwable
Error
Exception
IOException
ClassNot Found Exception
Runtime Exception
Import from java.io
FileNotFound Exception
When constructing a scanner or writer with a non-existent file
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
When calling Integer.parseInt or Double.parseDouble with an illegal argument
Illegal Argument Exception
NumberFormat Exception
IndexOut OfBounds Exception
When calling next, nextInt, or nextDouble on a scanner and no input is available
Import from java.util
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception
When calling nextInt or nextDouble on a scanner and the input is not of the expected form

Throwing an Exception

```
throw exceptionObject;
```

Example:

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

Purpose:

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

Checked and Unchecked Exceptions

- Two types of exceptions:

–Checked

- The compiler checks that you don't ignore them
- Due to external circumstances that the programmer cannot prevent
- Majority occur when dealing with input and output
- For example, `IOException`

Checked and Unchecked Exceptions (continue)

Two types of exceptions:

–Unchecked:

Extend the class `RuntimeException` or `Error`

They are the programmer's fault

Examples of run time exceptions:

```
NumberFormatException
IllegalArgumentException
NullPointerException
```

Example of Error: `OutOfMemoryError`

Checked and Unchecked Exceptions(continue)

- Categories aren't perfect:

–`Scanner.nextInt` throws unchecked `InputMismatchException`

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

–`FileReader` constructor can throw a `FileNotFoundException`

–Programmer cannot prevent users from entering incorrect input

Deal with checked exceptions principally when programming with files and streams

Throw Exception

Two choices:

1. Report it to the caller method

   Use `throws`

2. Handle the exception

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

Throw Multiple Exceptions

- For multiple exceptions:

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

- Keep in mind inheritance hierarchy:

  If method can throw an `IOException` and `FileNotFoundException`, catch second exception first, or just catch `IOException`

- Better to throw exception than to handle it incompetently

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

example_A

Example

Check example _A

- 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
```

exmaple_A

Now change 3 to 3.1 in the first line of text.txt file.
Run the program, and the program will crash with message InputMismatchException.
Since the nextInt() expects an integer number, but it is a double number.
test.txt file:

```
3.1
1.45
-2.1
0.05
```

exmaple_A

Now remove the last element in the file (remove 0.05). Run the program, and the program will crash with message NoSuchElementException.
Since the nextInt() expects an integer number, but it is a double number.
test.txt file:

```
3
1.45
-2.1
```

Catching Exceptions

- Install an exception handler with **`try/catch`** statement
- **`try`** block contains statements that may cause an exception
- **`catch`** clause contains handler for an exception type

Example

```
try
{
   String filename = "test.txt";
   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");
}
```

Catching Exceptions

- Statements in try block are executed
- If no exceptions occur, `catch` clauses are skipped
- If exception of matching type occurs, execution jumps to `catch` clause
- `catch (IOException exception)`

-`exception` contains reference to the exception object that was thrown

-`catch` clause can analyze object to find out more details

-`exception.printStackTrace()`: printout of chain of method calls that lead to exception (default if not try and catch created)

General Try Block

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

example_B

Now run the program and modify text.txt file as we did in example_A.
The program will not crash, and it will provide useful comments.
It is up to the programmer to terminate the program, or do something else.

Check example_B

Problem with example_B

What is the problem with example_B?

1. We opened a file to read in fileRead(...) method.
2. If an exception occurs in readData(...) method, then the statement in.close() will skip.
3. However, we must close the file before terminating the process.

Handling resources

When a resource that must be closed no matter if it ends normally, or an exception occurs.

Use try with resources.

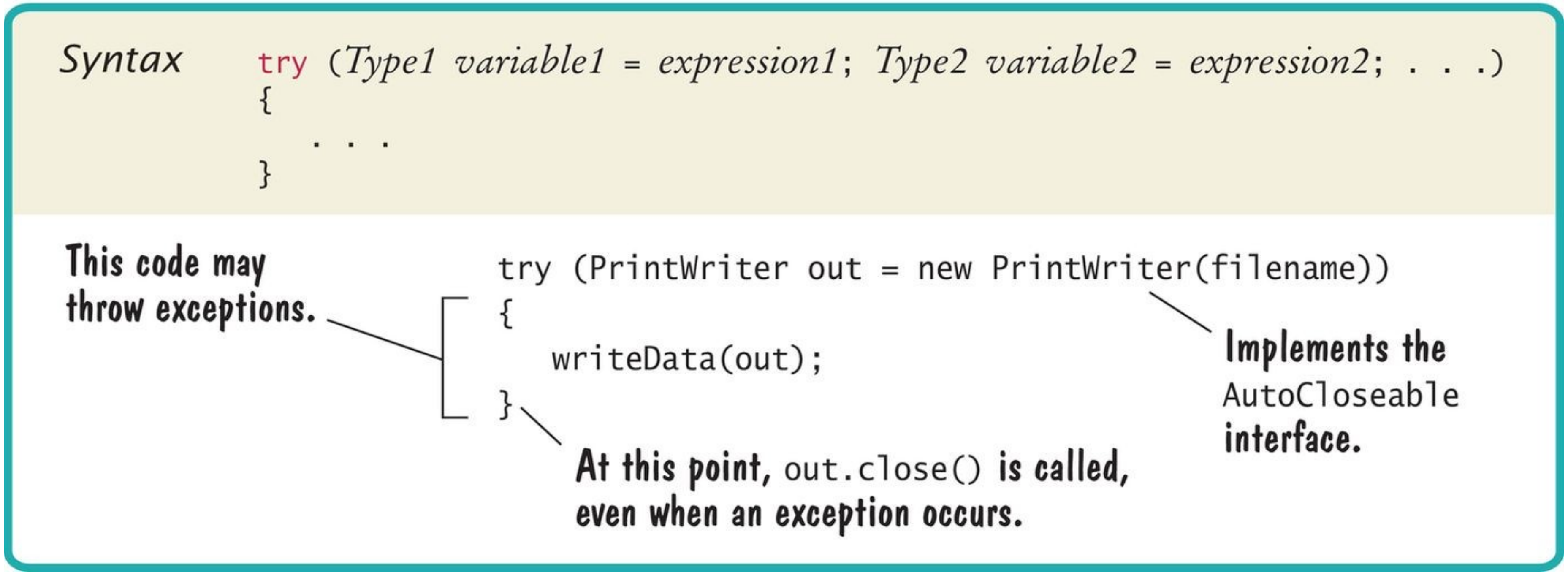
Example:

```
try(PrintWriter out = new PrintWriter(fileName))
{
    writeData(out);
    // out.close() is always called
}
```

Try with resource

There may be multiple resources. They all should be separated by semicolon.

We use try with resources normally when dealing with files.



Syntax

```
try (Type1 variable1 = expression1; Type2 variable2 = expression2; . . .)
{
   . . .
}
```

```
try (PrintWriter out = new PrintWriter(filename))
{
   writeData(out);
}
```

This code may throw exceptions.
Implements the AutoCloseable interface.
At this point, out.close() is called, even when an exception occurs.

Multiple resources

Multiple resources

```
try(Scanner in = new Scanner(inFile);
     PrintWriter out = new PrintWriter(outFile))
{
    String input = in.nextLine();
    String result = process(input);
    out.println(result);
}
```

Check example_C

We have another alternative which is more general. Use finally clause for more general type of process handling before terminating the method.

The `finally` general resource cleaner

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

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

- Must execute `reader.close()` even if exception happens
- Use **`finally`** clause for code that must be executed "no matter what"

The `finally` clause

`finally` Executed when `try` block is exited in any of three ways:

–After last statement of `try` block

–When an exception was thrown in `try` block and not caught

–After last statement of `catch` clause, if this `try` block caught an exception

example_D

```
Scanner in=null;
try{
   FileReader reader = new FileReader(filename);
   in = new Scanner(reader);
   readData(in);
}finally{
   in.close(); // will run no matter what
}
return data;
```

Check example _D

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.

Designing Your Own Exception

- 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);
}
```

Designing Your Own Execution Types

- ➢Make it an unchecked exception–programmer could have avoided it by calling `getBalance` first
- ➢Extend `RuntimeException` or one of its subclasses
- ➢Supply two constructors

1. Default constructor
2. A constructor that accepts a message string describing reason for exception

Designing Your Own Execution Types

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

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

A Complete Program

Example

–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
```

Check example_E

A Complete Program

- 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

A Complete Program

- ➢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

Summary

When you throw an exception, processing continues in an exception handler.

Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.

The try-with-resource statement ensures that a resource is closed when statement ends normally or due to an exception.

The final statement is an alternative to try-with-resource to ensure that part of the program will run in any case. When statement ends normally or due to an exception.

Summary

Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.

To describe an error condition, provide a subclass of an existing exception class.

When designing a program, ask yourself what kinds of exception can occur.

For each exceptions, you need to decide which part of your program can competently handle it.