320341 Programming in Java

Fall Semester 2014

Lecture 6: Exceptions, Logging, Assertions

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Objectives

JACOBS UNIVERSITY

This lecture focuses on the following

- Dealing with errors in Java
- Catching exceptions in Java
- Event Logging
- Using assertions
- Debugging techniques

Introduction

JACOBS UNIVERSITY

Basic Philosophy of Java

- “Badly formed code will not be run”
- Ideal time to catch error is at compile time

- However, not all errors can be caught at compile time

- Errors that are not caught at compile time are caught during run time using **exception handling mechanism**

Introduction

When an error occurs, make sure you do the following at least:

Notify the user of the error

i. **Save** all work

ii. Allow the user to **gracefully exit** the program

OR

Notify the user of the error

i. **Return to a safe state**

ii. Enable the user to execute other commands

Dealing With Errors

JACOBS UNIVERSITY

Error can happen for any number of reasons, e.g.,

- Flaky network connection
- Use of invalid array index
- Use of object reference that has not been initialized

AND if an operation cannot be completed due to the error, the program must *either*:

- Return to a **safe state**, and enable the user to execute other commands

OR

- Allow the user to **save all work and terminate** the program

Examples of Errors

JACOBS UNIVERSITY

Error	Example
User input errors	Typing typos, wrong URL syntax
Device errors	Web page temporarily unavailable, printer out of paper
Physical limitations	Disk fill up, running out of memory
Code errors	Method not performing correctly, computing wrong array index, computing nonexistent entry in a hash table
…	

Exceptions in Java

JACOBS UNIVERSITY

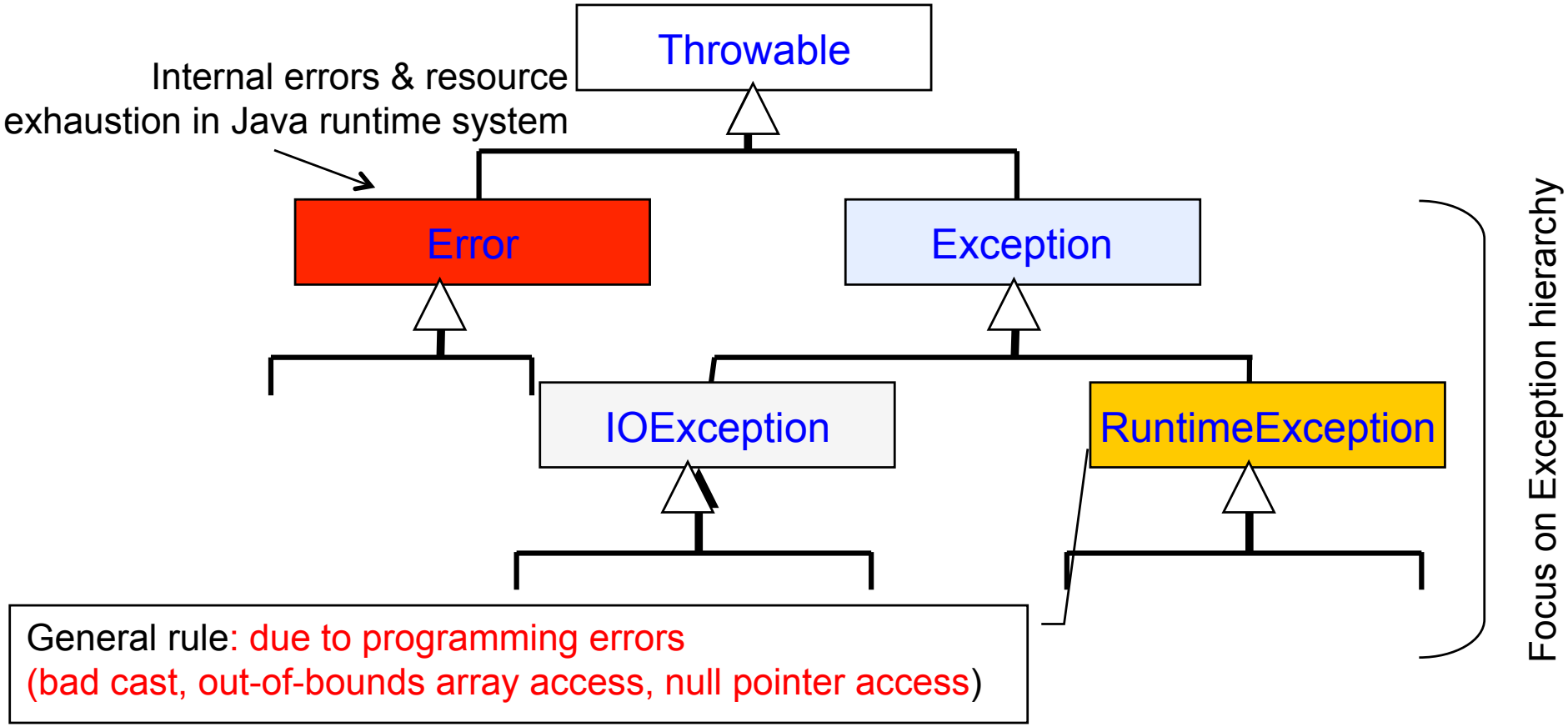
Java allows every method an **alternative exit path** if an error occurs

- The method does not return a value
- It **throws** an **object that encapsulates error information**
- The method exits immediately; does not return its normal value
- *The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation*
- The **Exception handling mechanism** begins search for an **exception handler** that handles that particular error condition

The Classification of Exceptions

An exception object is always an instance of a class derived from **Throwable**

Exception Hierarchy



General rule: due to programming errors
(bad cast, out-of-bounds array access, null pointer access)

The Classification of Exceptions

JACOBS UNIVERSITY

Unchecked Exceptions

- Any exception that derives from the class **Error** or **RuntimeException**
- Ex. **ArrayIndexOutOfBoundsException**, **NullPointerException**
- Rule: *"If it was a RuntimeException it was your fault"*

Checked Exceptions

- All other exceptions NOT deriving from the class **Error** or **RuntimeException**
- Ex. **IOException**, **MalformedURLException**

- The compiler checks that you provide exception handlers for all checked exceptions

Declaring Checked Exceptions

JACOBS UNIVERSITY

Checked exceptions are specified as part of a method header

```
public FileInputStream(String name) throws
                                        FileNotFoundException
```

The declaration says

- The constructor produces **FileInputStream** object from a **String** parameter, but that it can also throw a **FileNotFoundException**

- The throws keyword is used to specify exceptions thrown by a method

- If an error occurs, the runtime system will search for an exception handler that knows how to deal with **FileNotFoundException** objects

Declaring Checked Exceptions

JACOBS UNIVERSITY

If there is more than one exception, list all in the header, separated by commas

Example:

```
public Image loadImage(String s) throws EOFException,
                                        MalformedURLException {
    ...
}
```

When to Throw an Exception?

JACOBS UNIVERSITY

An Exception is thrown when:

1. A method that calls a checked exception is called
2. An error is detected & a **checked exception** is called with throw statement
3. A programming error is made resulting in an **unchecked exception**
4. An **internal error** occurs in the virtual machine or runtime library

If (1) or (2) occurs, inform the client programmer

- If no handler catches the exception, the current thread terminates

*Unchecked exceptions (inheriting from **RuntimeException**) are not advertised*

When to throw an Exception?

JACOBS UNIVERSITY

Declaring checked Exceptions

- A method must declare all the **checked exceptions** that it might throw

Unchecked Exceptions

- Unchecked exceptions are beyond your control (**Error**) or

- Result from conditions you should not have allowed to occur in the first place (**RuntimeException**)

How to Throw an Exception

To throw an exception do the following:

- Find appropriate exception class
- Make an object of the class
- Throw it

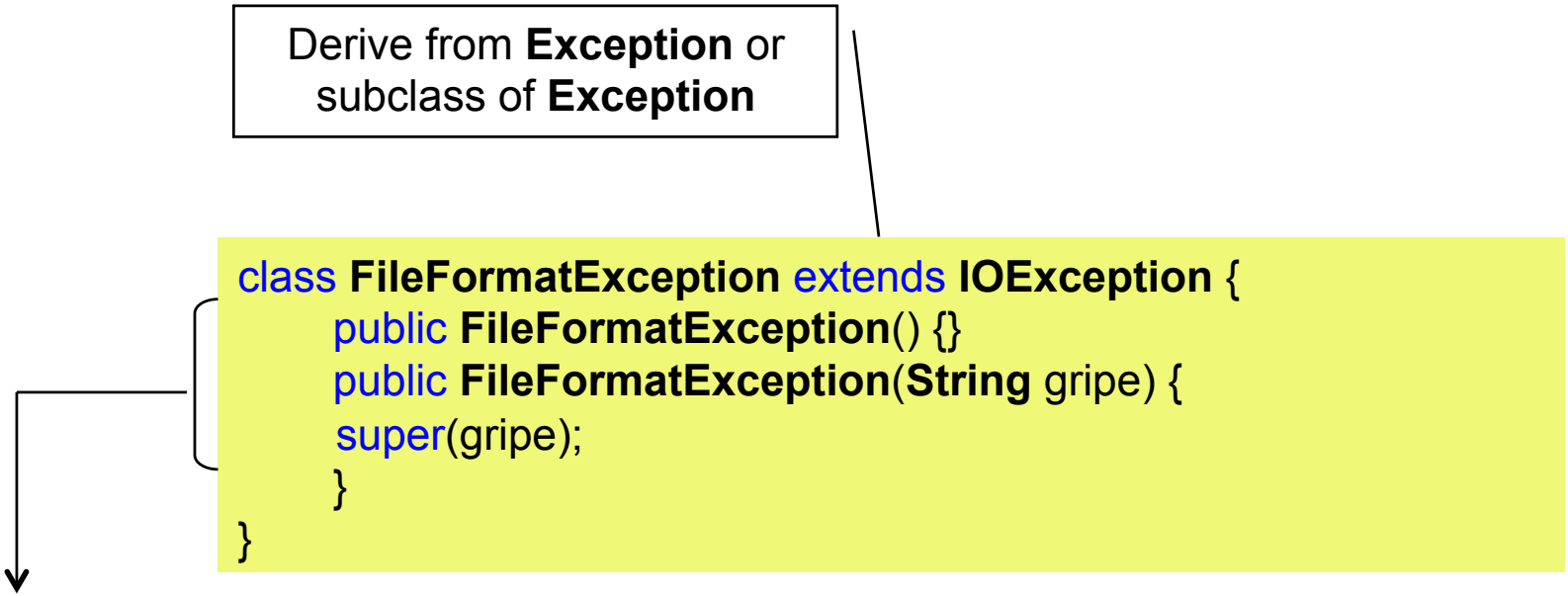
Exception declaration

Exception throwing (use keyword throw)

```
String readData(Scanner in) throws EOFException {
        . . .
        while (. . .) {
            if (!in.hasNext()) { // EOF encountered
                if (n < len)
                    throw new EOFException();
            }
            . . .
        }
        return s;
}
```

Creating Exception Classes

Derive from **Exception** or from a child class of **Exception** e.g. **IOException**



It is customary to give both default constructor and constructor with detailed message

Creating Exception Classes

JACOBS UNIVERSITY

How to throw your own **Exception**

```
String readData(BufferedReader in) throws FileFormatException {
        . . .
        while (. . .)  {
            if (ch == -1) { // EOF encountered
                if (n < len)
                    throw new FileFormatException();
            }
            . . .
        }
       return s;
}
```

- Use `toString()` method of **Throwable** superclass to print detailed message
- Can also use `getMessage()` method of **Throwable** to get detailed message

Catching Exceptions

If an **Exception** is not caught, the program terminates and prints a stack trace

- Use `try/ catch` block to catch exceptions

```
try {
        statement1;
        …
         statementn;
} catch (ExceptionType e) {
        handler for this type
}
```

Catching Exceptions

JACOBS UNIVERSITY

If an **Exception** of the type in the catch clause is thrown:

- The program skips the remainder of the code in the try block
- The program executes the handler code inside the catch clause
- If the **Exception** is not of the type specified in the catch, the method exits immediately

The catch clause is skipped if no **Exception** is thrown

Catching Exceptions

JACOBS UNIVERSITY

Example

Guarded region

Exception handler

```
public void read(String filename) {
    try {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) !=-1) {
              process input
        }
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

- The compiler enforces the throws specifier
- If you call a method that throws a checked exception, you must either handle it or pass it

Catching Exceptions

JACOBS UNIVERSITY

When writing a method that *overrides* a superclass method that throws no exceptions, then

- You **MUST** catch each checked exception in the method's code

- You are not allowed to add more throws specifiers to a subclass method than are present in the superclass method

Catching Multiple Exceptions

JACOBS UNIVERSITY

Guarded region

Handler for situation A

Handler for situation B

Handler for situation C

```
try {
    code that might throw exceptions
} catch (MalformedURLException e1) {
    emergency action for malformed URLs
} catch (UnknownHostException e2) {
    emergency action for unknown hosts
} catch (IOException e3) {
    emergency action for all other I/O problems
}
```

Use

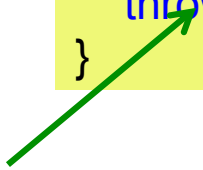
- `e3.getMessage()` to get detailed error message if there is one from `e3`
- `e3.getClass().getName()` to get the actual type of the exception object

Rethrowing and Chaining Exceptions

JACOBS UNIVERSITY

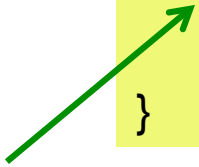
Example

```
try {
    access the database
} catch (SQLException e) {
    throw new ServletException("database error: " + e.getMessage());
}
```



Rethrowing an exception

```
try {
     access the database
} catch (SQLException e) {
     Throwable se = new ServletException("database error");
     se.setCause(e);
     throw se;
}
```



Setting cause of exception

The finally Clause

Use finally construct to guarantee that code segment must be executed

Guarded region

Handler for situation **A**

Handler for situation **B**

Handler for situation **C**

finally clause is used at the end of all exception handlers

```
try {
        // The guarded region: Dangerous activities
        // that might throw A, B, or C
} catch (A a1) {
        // Handler for situation A
} catch (B b1) {
        // Handler for situation B
} catch (C c1) {
        // Handler for situation C
} finally {
        // Activities that happen every time
}
```

The finally Clause

Example

```
class ThreeException extends Exception {}
public class FinallyWorks {
        static int count = 0;
        public static void main(String[] args) {
                                                while (true) {
                                                                try { //
        Post-increment is zero first time:
                                                        if (count++ == 0) throw new
        ThreeException();
        System.out.println("No exception");
                        } catch (ThreeException e) {
                                                                System.err.println
        ("ThreeException");
                        } finally {

        System.err.println("In finally clause");
                                                        if(count == 2) break; // out of
        "while"
```

Output

```
ThreeException
In finally clause
No exception
In finally clause
```

The finally Clause

The finally clause can be used without the catch clause

If exception occurs, it must be caught

```
Graphics g = image.getGraphics();
try {
     code that might throw exceptions
} finally  {
    g.dispose();
}
```

- The *g.dispose()* command in the finally clause is always executed

Decoupled try/ catch and try/ finally blocks

Closes input stream

Reports errors

```
InputStream in = ...;
try {
    try  {
         code that might throw exceptions
    }  finally {
           in.close();
    }
}   catch (IOException e)  {
        show error dialog
}
```

The finally Clause

Example

try block with return statement

finally block with return statement

```
public static int f(int n) {
    try  {
        int r = n * n;
        return r;
    }  finally  {
        if (n == 2)  return 0;
    }
}
```

Return value inside finally clause is returned by method

Ex. f(2) will return 0, instead of 4

What happens when the finally clause throws an exception?

- The original exception is lost!

Stack Trace Elements Analysis

JACOBS UNIVERSITY

Stack Trace

- A **stack trace** is a listing of all pending method calls at a particular point in the execution of a program

- The listing can be seen when a Java program terminates with *uncaught exceptions*

Stack Trace Elements Analysis

JACOBS UNIVERSITY

- The **stack trace** only traces back to the statement that throws the exception, not necessarily to the root cause of the error

- `printStackTrace` method of the **Throwable** class gives a text description of a stack trace

- `getStackTrace` method now used to get an array of **StackTraceElement** objects

Stack Trace Elements Analysis

JACOBS UNIVERSITY

- The **StackTraceElement** class has methods to obtain the *file name* and *line number*, as well as the *class* and *method name*, of the executing line of code

- The `toString` method yields a formatted string containing all of this information

Processing stack trace elements

```
public static int factorial(int n)  {
    System.out.println("factorial(" + n + "):");
    Throwable t = new Throwable();
    StackTraceElement[] frames = t.getStackTrace();
    for (StackTraceElement f : frames)
        System.out.println(f);
    int r;
    if (n <= 1) r = 1;
    else r = n * factorial(n - 1);
    System.out.println("return " + r);
    return r;
}
```

Exception Restrictions

JACOBS UNIVERSITY

- When you **override** a method, you can throw only the exceptions that have been specified in the base-class version of the method

- There exists other compile-time restrictions for exceptions

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

    StormyInning() throws RainedOut,
    BaseballException { }
    StormyInning(String s) throws Foul,
    BaseballException {}
}
```

Can add new exceptions to constructors, but must handle base-class constructors

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

  //! void walk() throws PopFoul {} //
       Compile error

}
```

Regular methods must conform to the base class

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

//! public void event() throws RainedOut
        {}
}
```

Interface CANNOT add exceptions to existing methods from the base class

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

public void rainHard() throws RainedOut
      {}

}
```

If the method doesn't already exist in the base class, the exception is OK

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

      public void event() {}

}
```

You can choose to not throw any exceptions, even if base version does

Exception Restrictions

```
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}
```

```
public class StormyInning
      extends Inning implements Storm {

    void atBat() throws PopFoul {}
}
```

Overridden methods can throw inherited exceptions

Using Exceptions

JACOBS UNIVERSITY

1. Exception handling is not supposed to replace a simple test

- Prefer

```
if (!s.empty()) s.pop
        ();
```

to

```
try { s.pop();
} catch (EmptyStackException e) {}
```

2. Do not micromanage exceptions

- Prefer

```
try  {
     for (i = 0; i < 100; i++)  {
         n = s.pop();
         out.writeInt(n);
     }
} catch (IOException e) {
    // problem writing to file
} catch (EmptyStackException s) {
   // stack was empty
}
```

rather than

```
for (i = 0; i < 100; i++)  {
     try  {
          n = s.pop();
     }  catch (EmptyStackException s)  {
             // stack was empty
     }
     try  {
         out.writeInt(n);
     }  catch (IOException e)  {
         // problem writing to file
     }
}
```

Using Exceptions

JACOBS UNIVERSITY

3. Make good use of the exception hierarchy

- Ex.: Don't just throw a **RuntimeException,** but, find an appropriate subclass or create your own

4. Do not squelch exceptions

Exception not handled

```
public Image loadImage(String s)  {
    try  {
        code that threatens to throw checked exceptions
    }  catch (Exception e) {
    }
}
```

Using Exceptions

JACOBS UNIVERSITY

5. When you detect an error, "tough love" works better than indulgence

- Ex.: It is better for `Stack.pop` to throw a **EmptyStackException** at the point of failure than to return null, thus have a **NullPointerException** occurring at a later time

6. Propagating exceptions is not a sign of shame

- Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands

Not a sign of shame

```
public void readStuff(String filename)
                    throws IOException {
        InputStream in = new FileInputStream(filename);
. . .
}
```

Rules 5 and 6 can be summarized as **"throw early, catch late"**

Logging

As from JDK 1.4 a logging API is provided

Advantages of **Logger**

- Log records are easily suppressed and turned on
- Suppressed logs are very cheap, thus can be left in application
- Log records can be directed to different handlers

Logging

JACOBS UNIVERSITY

Advantages of Logger cont…

- Both loggers and handlers can filter records
- Log records can be formatted in different ways (e.g., plain text, XML)
- Multiple loggers can be used
- By default, the logging configuration is controlled by a configuration file

Basic Logging

Default **Logger.global** can be used instead of **System**.out

```
Logger.global.info("File->Open menu item selected");
```

Record printed like

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen
INFO: File->Open menu item selected
```

The statement

```
Logger.global.setLevel(Level.OFF);   // turns off logging
```

Turns logging off

Advanced Logging

JACOBS UNIVERSITY

In practice we want to define our own loggers

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

Logger names are hierarchical

There are seven log levels

- SEVERE
- WARNING
- INFO

Logged by default

- CONFIG
- FINE
- FINER
- FINEST

Advanced Logging

Use

```
logger.setLevel(Level.ALL);
```

To turn on all log levels OR

```
logger.setLevel(Level.OFF);
```

To turn all logging off

Example log methods

- logger.warning(message);
- logger.fine(message);
- ….
- Use log method and supply level like for example:

```
logger.log(Level.FINE, message);
```

JACOBS UNIVERSITY

Changing the Log Manager Configuration

- Edit the configuration file
- The default configuration file is located at **jre/lib/logging.properties**
- To use another file, start your application:

```
java -Djava.util.logging.config.file=configFile MainClass
```

- Handlers also have levels. To see **FINE** messages on the console, set

```
java.util.logging.ConsoleHandler.level=FINE
```

- Logging properties file is processed by **java.util.logging.LogManager** class

Handlers

JACOBS UNIVERSITY

The **ConsoleHandler** is the default handler

java.util.logging.ConsoleHandler.level=INFO

- To log records with level **FINE**, change both the default logger level and the handler level in the configuration

- OR bypass configuration file and install own handler

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

Handlers

JACOBS UNIVERSITY

Send records to default **FileHandler** as follows:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

- The records are sent to a file **java*n*.log** in user's home directory, where n is a number

Variable Descriptions

- %h The value of the user.home system property
- %t The system temporary directory
- %g The generation number for rotated logs
- %u A unique number to resolve conflicts
- %% The % character

Example logging.properties File

JACOBS UNIVERSITY

```
# setting to limit messages printed to the console.
level= INFO
############################################################
# Handler specific properties. Describes specific configuration info for Handlers.
############################################################

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

############################################################
# Facility specific properties. Provides extra control for each logger.
############################################################

# For example, set the com.xyz.foo logger to only log SEVERE messages:
com.xyz.foo.level = SEVERE
```

Logging Usage

JACOBS UNIVERSITY

1. Use single logger for simple applications

2. Get the logger using the statement

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

Logging Usage

JACOBS UNIVERSITY

3. Install a reasonable default logger for your application

- Place the code into the main method of your application

4. All messages with level **INFO**, **WARNING**, and **SEVERE** show up on the console

- Reserve these levels for messages that are meaningful to the users of your program

- The level **FINE** is a good choice for logging messages that are intended for programmers

Using Assertions

Assertions are a commonly used idiom for **defensive programming**

The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code

Syntax

- Use assert keyword

```
assert condition;
```

- OR

```
assert condition : expression;
```

- Both statements evaluate the condition and throw an **AssertionError** if it is false

Using Assertions

To **assert** that x is nonnegative, you can simply use the statement

```
assert x >= 0;
```

- Or you can pass the actual value of x into the **AssertionError** object, so that it gets displayed later

```
assert x >= 0 : x;
```

- Tell the compiler that you are using the assert keyword

Using Assertions

JACOBS UNIVERSITY

Assertions are disabled by default

- Enable assertions by running the program with the

  -enableassertions or **-ea** option:

```
java -enableassertions MyApp
```

- Turn on assertions in specific classes or in entire packages

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

Using Assertions

- Disable assertions in certain classes and packages with the

  –disableassertions or **-da** option:

```
java -ea:... -da:MyClass MyApp
```

- Use the **-enablesystemassertions/-esa** switch to enable assertions in system classes

Assertion Usage

Three mechanisms to deal with system failures:

- Throwing an exception
- Logging
- Using assertions

Assertion Usage

JACOBS UNIVERSITY

When to use assertions?

- For fatal, unrecoverable errors

- Turned on only during development and testing

- *Thus, assertions should only be used to locate internal program errors during testing*

Debugging Techniques

JACOBS UNIVERSITY

Print or log the value of any variable with code like this:

System.out.println("x=" + x); OR *Logger.global.info("x=" + x);*

Put separate main method in each class for testing each class

Use **JUnit** (http://junit.org) for preparing tests

Use a **logging proxy** i.e., an object of a subclass that intercepts method calls, logs them, and then calls the superclass

Use a **Debugger** (The Eclipse Debugger)

Java debugger in Eclipse

JACOBS UNIVERSITY

Eclipse provides a built-in Java debugger

Provides standard debugging functionality including:

- Step execution
- Inspection of variables and objects
- Setting breakpoints
- Suspending and resuming threads
- On-the-fly code fixing

and much more ….

Reading Assignment

- Core Java 2 Volume I, Chapter 11. Exceptions and Debugging by Horstmann and Cornell