Contents

1. Classes and Objects
2. Inheritance
3. Interfaces
4. Exceptions and Error Handling
5. Intro to Concurrency
6. Concurrency in Java
7. Graphics and Animation
8. Applets
Chapter 4: Exceptions and Error Handling

After this chapter you will be able to:

- Use throw to generate Exceptions
- Handle Exceptions with catch
- Add throws to method declarations
- Customize exception handling
Motivation

• Exception are a mechanism for handling run-time errors

• Advantages:
  – flexible model
  – clean separation of error handling code from application code

• Disadvantages:
  – extra run-time overhead
History

• Exception handling was introduced in Ada83 and is now present in:
  – C++
  – Common Lisp
  – Java
  – Standard ML
  – SmallTalk

• **Conclusion:** Exception handling is the de facto standard mechanism for error handling in modern programming languages
Metaphors

• The two basic ideas are:

  – **throw**: at some point when an error occurs during execution an exception object is created and thrown

  – **catch**: an exception object may be caught and handled at some other point in the program
Java Exception Classes

```
Exception
 /\      /
|       |    
|  AWTException  |
|              |
|              |

Exception
 /\      /
|       |    
|  IOException  |
|              |
|              |

Exception
 /\      /
|       |    
|  EOFException  |
|              |
|              |

Exception
 /\      /
|       |    
|  RunTimeException  |
|              |
|              |

Exception
 /\      /
|       |    
|  ArithmeticException  |
|              |
|              |

Exception
 /\      /
|       |    
|  IndexOutOfBoundsException  |
|              |
|              |

Exception
 /\      /
|       |    
|  NullPointerException  |
|              |
|              |
```
Overview Of Exception Handling

• When an exception is thrown normal execution stops!

• A search is then performed for a corresponding catch to handle the error

• If no corresponding catch is found then the program terminates
Generating Exceptions Explicitly

• Can call an `Exception` constructor

    ```java
    new Exception(“error description”);
    ```

• and then `throw` an exception object explicitly

    ```java
    throw new Exception(“error description”);
    ```

• If the `Exception` is not caught and handled then execution terminates
Generating Exceptions Implicitly

• No `throw` appears here

```java
int a = b / c;
```

• However dividing by zero will generate an `ArithmeticException`
Catching Exceptions

```
try {
    application code
} catch error handling code
```

• If the application code generates an exception and there is a corresponding `catch` then the error handling code is executed.

```
try {
    int y = 4 / x;
} catch (ArithmeticException e) {
    System.out.println("divide by zero occurred");
}
```

• Variable `e` is bound to the caught `Exception` object.
Stack Trace

• Can use the Exception object to print a stack trace

```java
try {
    int y = 4 / x;
} catch (ArithmeticException e) {
    System.out.println("divide by zero occurred");
    e.printStackTrace();
}
```

• A stack trace is the sequence of methods called up to the point where the Exception was thrown
Multiple Catches

• Can have multiple catch branches to handle different kinds of Exceptions

```java
try {
    foo();
}

    catch (ArithmeticException e) {
        System.out.println("divide by zero occurred");
    }
    catch (Exception e) {
        System.out.println("Some kind of Exception occurred");
    }
```

• The first catch that matches the thrown object is executed
Compiler Error

• More specific catch branches must precede the more general

```java
try {
    foo();
}
catch (Exception e) {
    System.out.println(
        "Some kind of Exception occurred");
}
catch (ArithmeticException e) {
    System.out.println("divide by zero occurred");
}
```

• What is wrong?
Compiler Checking

• The Java compiler enforces that Exceptions will be handled *

* Except for subclasses of RuntimeException
  - IndexOutOfBoundsException
  - NullPointerException
  - ArithmeticException
  - ...

• The compiler performs no checking for subclasses of RuntimeException (generally, these should not be caught)
Compiler Error

- FileInputStream constructor may throw a FileNotFoundException

```java
import java.io.FileInputStream;

FileInputStream inputFile = new FileInputStream("f1.dat");
```

- Compiler error: FileNotFoundException not caught or declared
Catching IOException

• Catching the IOException is **one way** to satisfy the compiler

```java
import java.io.FileInputStream;

try {
    FileInputStream inputFile =
        new FileInputStream("f1.dat");
} catch (java.io.FileNotFoundException e) {
    System.out.println("File not found");
}
```

Declaring that an exception may be thrown is another way
RunTimeException Subclasses

• Indexing into an array may cause an IndexOutOfBoundsException

```
int [] x = {1,2,3,4}
for (int i = 0; i < x.length; i++)
    sum += x[i];
```

• IndexOutOfBoundsException is a subclass of RunTimeException (no compiler checking)
Declaring Exceptions

• A method may declare that an exception may be thrown using keyword `throws`

```java
public char Read () throws java.io.IOException {
    System.out.println("Enter input char: ");
    return (char) System.in.read();
}
```

• No `try-catch` necessary here
Exceptions Propogate

• If a thrown Exception is not caught inside a method then the Exception will propogate to the point where the method was called (and so on until caught)

```java
public void method1 () {
    try {
        method2(); // Exception propogates to here
    }
    catch (Exception e) {
        System.out.println("caught it here");
    }
}

public void method2 () throws Exception {
    throw new Exception();
}
```
Rethrowing

- An Exception may be caught and then thrown again

```java
import java.io.FileInputStream;

try {
    FileInputStream inputFile =
        new FileInputStream(fileName);
} catch (java.io.FileNotFoundException e) {
    if (fileName.equals("default.dat"))
        e.fillInStackTrace();
    throw e;
}
```

- `fillInStackTrace` resets the stack trace
Flexibility

• The propagation of Exceptions provides flexibility

• An exception object may be caught locally or at another point further up the call trace

• An Exception should be caught locally if it can be handled properly
Customizing Exceptions

• Create a subclass of class `Exception`

```java
import java.util.Date;

class DatedException extends Exception {
    Date d;

    public DatedException () {
        d = new Date(); // store's current time
    }

    public String toString () {
        // return string representation
        return d.toString();
    }
}
```
Catching DatedException

• Can print out the time of the error

```java
public static void main (String [] args) {
    try {
        foo();
    }
    catch (DatedException e) {
        System.out.println("Error occurred on: "+e);
    }
}
```

Output

Over Using Exceptions

• **Question:** What is wrong here?

```java
char c;
try {
    FileInputStream f = new FileInputStream("f1");
    while (true) {
        c = (char) f.read();
        System.out.print(c);
    }
}

catch (java.io.EOFException e) {
    System.out.println("end of file reached ");
}

catch (java.io.IOException e) {
    System.out.println("I/O error occurred");
}
```
Answer

• Exceptions are meant for *exceptional* (not expected) conditions

• When reading from a data file, it is *expected* that the end of file will be reached

• Similarly when indexing into arrays and strings, it is expected that the end will be reached

• **Conclusion:** Don’t use exceptions for expected conditions due to overhead and extra code required
File Input Revised

• Can test to see if `read` returned a valid byte

```java
char c;
try {
    FileInputStream f = new FileInputStream("f1");
    while ((c = (char) f.read()) > -1) {
        c = (char) f.read();
        System.out.print(c);
    }
} catch (java.io.IOException e) {
    System.out.println("I/O error occurred during read");
}
```
Finally

- Finally is used to guarantee that code is executed whether or not an Exception occurs.

```java
try {
    while (true) {
        x.doTransaction();
    }
}
catch (DatedException e) {
    System.out.println("Error occurred on: " + e);
}
finally {
    logFile.write("attempted: " + x.getId());
}
```

- Can have one finally in a try-catch.
It’s Exercise Time