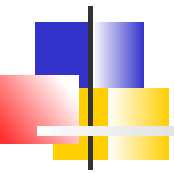
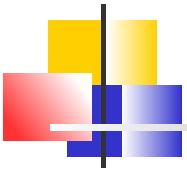
Software and Programming I

Input-Output and Exception Handling



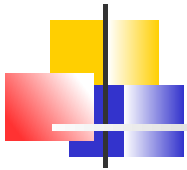
Roman Kontchakov / Carsten Fuhs

Birkbeck, University of London



Outline

- Reading and writing text files
- Exceptions
- The `try` block
- `catch` and `finally` clauses
- Command line arguments
  - Chapter 7
- slides are available at
  `www.dcs.bbk.ac.uk/~roman/sp1`

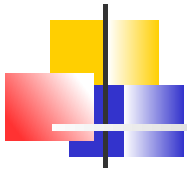


Reading Text Files

class `File` (package `java.io`) describes disk files and directories
(and has many methods, see API documentation)

use the `Scanner` class for reading text files
(package `java.util`)

```
1 File inputFile = new File("input.txt");
2 Scanner in = new Scanner(inputFile);
3 while (in.hasNextLine()) {
4     String line = in.nextLine();
5     // ... now process the line ...
6 }
7 in.close(); // close the file
8              // when you are done processing it
```



Reading Files and Web Pages

instances of `Scanner` can be constructed from (overloading)
`File`, `String` or `InputStream`

(`System.in` is an instance of `InputStream`)

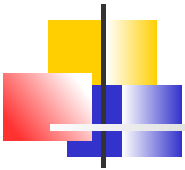
NB: `new Scanner("input.txt")` is not an error,
it creates a scanner for the string

```
1 URL pageURL = new URL("http://www.dcs.bbk.ac.uk/");
2 Scanner in = new Scanner(pageURL.openStream());
```

DIY web browser

NB: backslashes in file names need to be "escaped" as \\

```
"c:\\project\\sp1\\sp1-10.pdf"
```



Writing Text Files

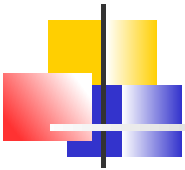
class `PrintWriter` has familiar methods:

`print`, `println` and `printf`

instances can be constructed from

`File`, `OutputStream` or `String` (file name!)

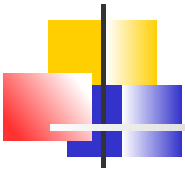
```
1 PrintWriter out = new PrintWriter("output.txt");
2 out.println("Hello, World");
3 double totalPrice = 2.99;
4 out.printf("Total: %8.2f\n", totalPrice);
5 out.close(); // close the file
6              // when you are done processing it
```



Reading Binary Data

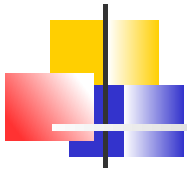
```
1 InputStream ins = new FileInputStream("shakira.mp3");
2 int nextByte;
3 while ( (nextByte = ins.read()) != -1 ) {
4     // read a byte from ins, put it into nextByte,
5     // compare it with -1 (end of input)
6
7     // ... now do something with nextByte ...
8 }
9 ins.close();
```

? What about the `=` in the loop condition?



Operation Precedence

- `()` method call — highest
- `!`, `(type)`, `-`, `+`, `++`, `--` — unary
  - `(type)`: type cast
  - `-`, `+`: unary minus/plus
- `*`, `/`, `%` — multiplicative
- `+`, `-` — additive
- `<`, `<=`, `>=`, `>` — relational
- `==`, `!=` — equality
- `&&` — logical AND
- `||` — logical OR
- `?` `:` — ternary
- `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `...` — assignment — lowest



Operation Precedence (2)

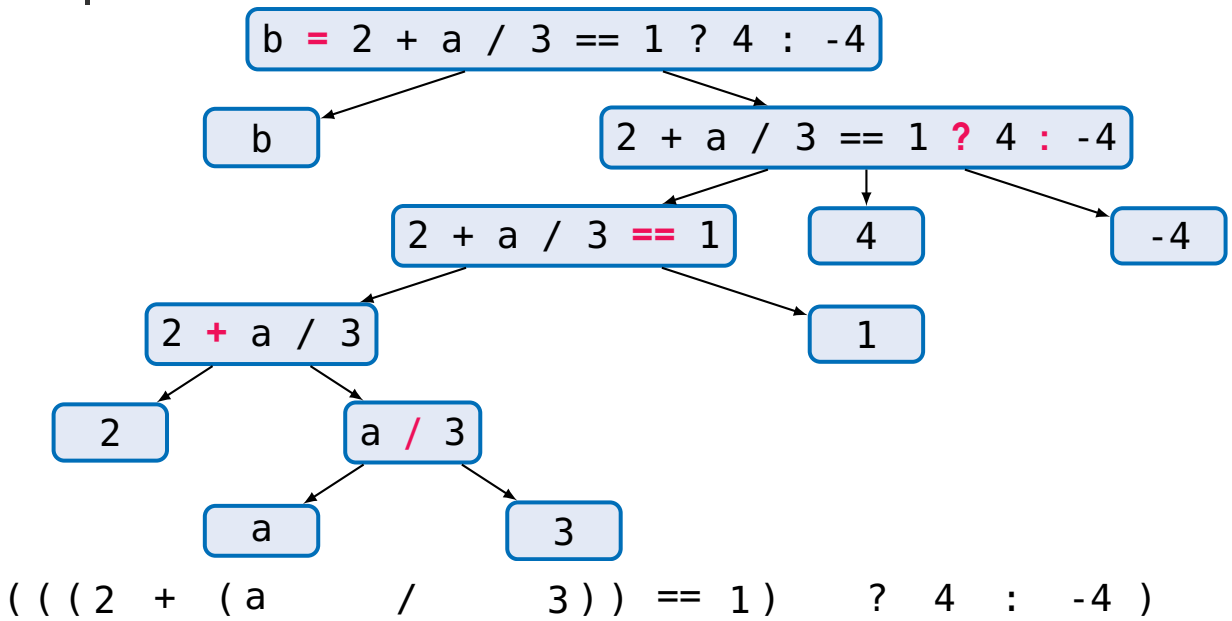
operators with **higher** precedence are evaluated
before operators with relatively lower precedence

What is the value of `2 + a % 3` if `a` is `11`

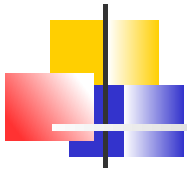
```
2 * 6 + a % 3 + 1 < 10 && a > 3
2 * 6 + a % 3 + 1 < 10 && !a > 3
2 + a / 3
2 + (double) a / 3
b = 2 + a / 3 == 1 ? 4 : -4
```

NB: = is also an operation!

Operation Precedence: Example



(((2 + (a / 3)) == 1) ? 4 : -4)



Operation Associativity

when two operators share an operand, the operator with the **higher precedence** goes first:

`1 * 2 + 3` is treated as `(1 * 2) + 3`

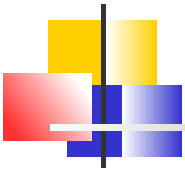
`1 + 2 * 3` is treated as `1 + (2 * 3)`

if an expression has two operators with the **same precedence**, the expression is evaluated according to its **associativity**:

`x = y = z = 17` is treated as `x = (y = (z = 17))`

NB: what is the effect of the following code?

```
1 boolean f = false;
2 if (f = true)
3    System.out.println("weird");
```



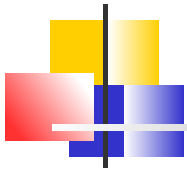
Operation Associativity (2)

- unary operators (cast, `!`, `-`, etc.)
  have **right-to-left** associativity

  `!!a` is `!(!a)`
- assignment operators (=, +=, etc.)
  have **right-to-left** associativity

  `a+= b+= 7` is `a+= (b+= 7)`
- conditional operator (`?:`) has **right-to-left** associativity
- all other operators have **left-to-right** associativity

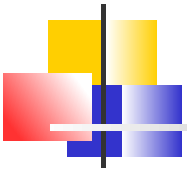
```
1 System.out.println("1 + 2 = " + 1 + 2);
2 System.out.println("1 + 2 = " + (1 + 2));
3 System.out.println(1 + 2 + " = 1 + 2");
```



Exceptions

an event that disrupts the **normal flow** of a program

- internal or **fatal errors** that happen rarely:
  `OutOfMemoryError` and other subclasses of `Error`
- **unchecked exceptions** are errors in the code:
  `IndexOutOfBoundsException`, `NullPointerException`, `IllegalArgumentException`
  and other subclasses of `RuntimeException`
- **checked exceptions** indicate something has gone wrong for an external reason beyond your control:
  `IOException`, `FileNotFoundException`
  and all other subclasses of `Exception`



Throwing Exceptions

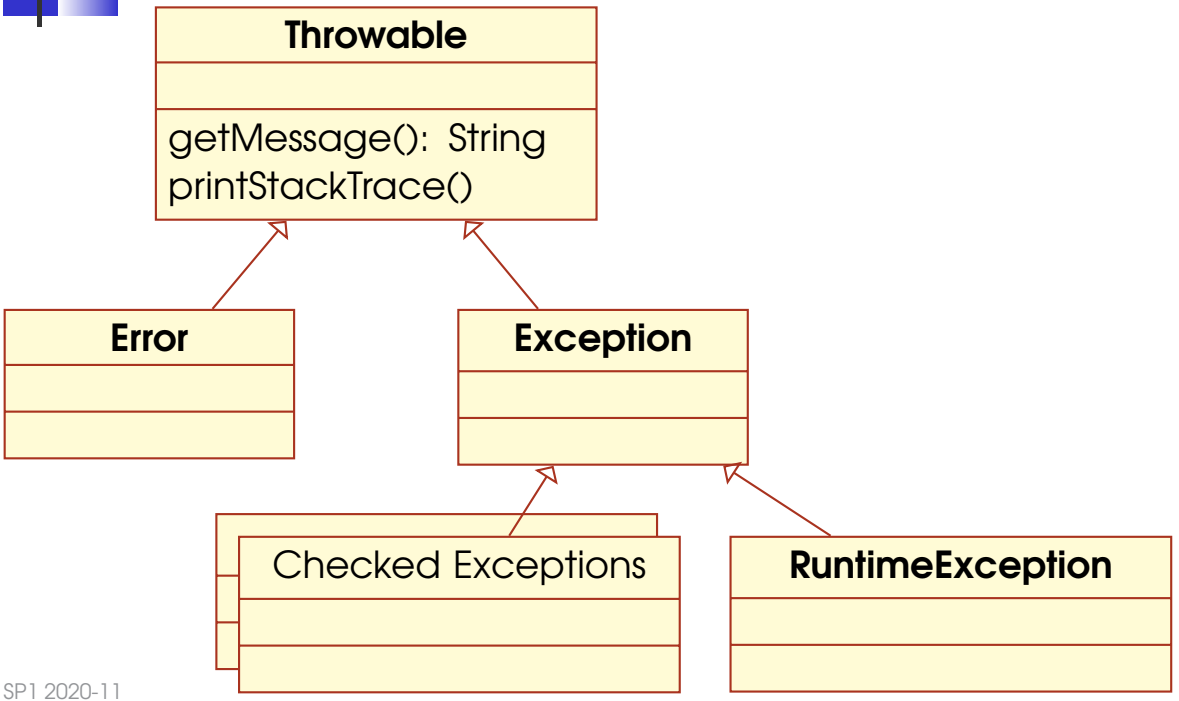
```
1 if (amount > balance) {
2     // create and throw an exception object
3     throw new IllegalArgumentException
4                     ("Amount exceeds balance");
5 }
6 balance -= amount;
```

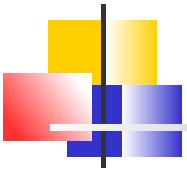
when you throw an exception,
processing continues in an **exception handler**

Catching Exceptions

```
1  try {
2      // what if the file does not exist?
3      Scanner in = new Scanner(new File("name.txt"));
4      // what if the file is empty?
5      String input = in.next();
6      // what if input is not an integer?
7      int value = Integer.parseInt(input);
8  }
9  catch (IOException e) { // class FileNotFoundException
10     e.printStackTrace(); //        extends IOException
11 }
12 catch (NumberFormatException e) {
13     System.out.println("Input was not a number");
14 }
```

Exception Classes





Checked Exceptions

external reason: `IOException`, `FileNotFoundException`, etc.

the compiler makes sure that your program handles checked exceptions:

either add a `throws` clause to a method that can throw a checked exception

```
1 public static String readFile(String filename) throws
2                                 FileNotFoundException {
3     Scanner in = new Scanner(new File(filename));
4     ...
5 }
```

or provide a handler for the exception

some methods detect errors, some handle them, and some just pass them along

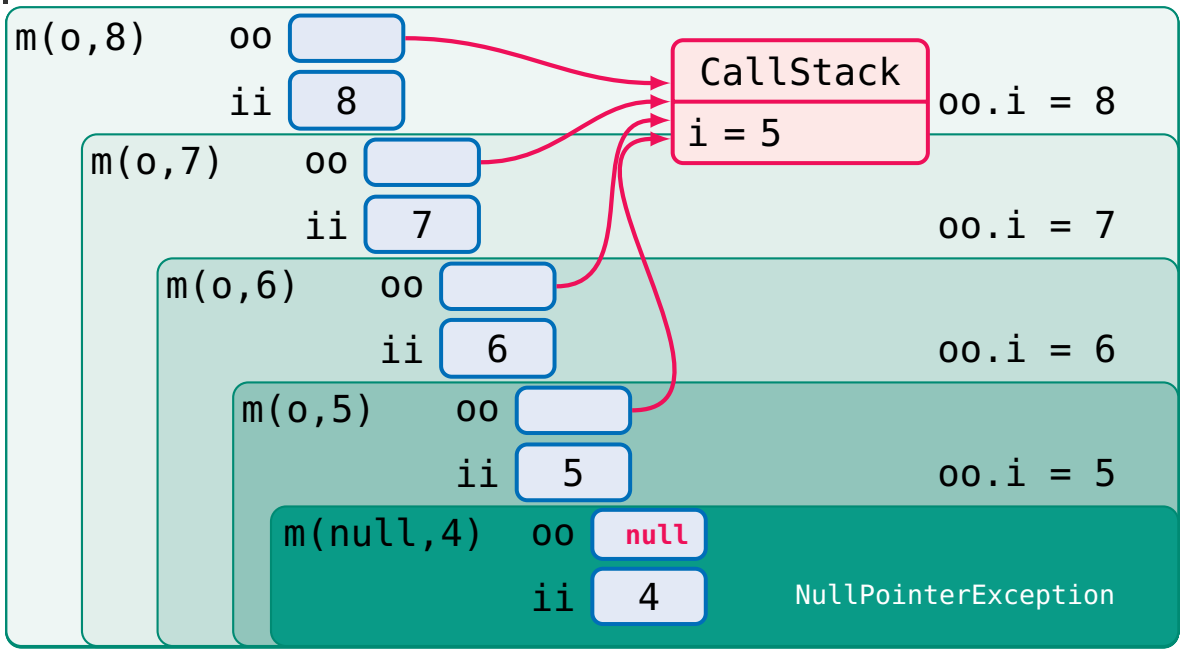
Call Stack: Example

```
1  public class CallStack {
2      private int i;
3      public static void m(CallStack oo, int ii) {
4          oo.i = ii; // throws a NullPointerException if
5          m((ii > 5) ? oo : null, ii - 1); // oo == null
6      }
7      public static void main(String[] args) {
8          CallStack o = new CallStack();
9          try {
10             m(o, 8);
11         }
12         catch (NullPointerException e) {
13             System.out.println("o.i = " + o.i);
14         }
15     }
16 }
```

Call Stack Example:

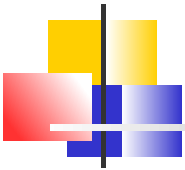
`oo.i = ii;`

`m((ii > 5) ? oo : null, ii - 1);`



Call Stack Example:

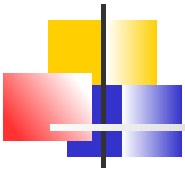
```
oo.i = ii;
m((ii > 5) ? oo : null, ii - 1);
```



The finally Clause

```
1  // the variable must be declared outside so that
2  //            the finally clause can access it
3  PrintWriter out = new PrintWriter(filename);
4  try {
5      writeData(out);   // may throw exceptions
6  }
7  finally {         // this code is always executed:
8      out.close(); // the stream has to be closed
9                   // even if an exception is thrown
10 }
```

NB: do not use `catch` and `finally` in the same `try` statement
put the above fragment into an outer `try` . . . `catch` . . .



Command Line Arguments

programs that start from the command line receive
the command line arguments in the `main` method

consider a program `CaesarCipher` that
encrypts or decrypts a file to another file

- `java CaesarCipher input.txt output.txt`

  `args[] = { "input.txt", "output.txt" }`

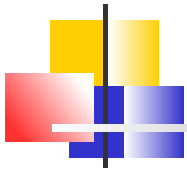
- `java CaesarCipher -d input.txt output.txt`

  `args[] = { "-d", "input.txt", "output.txt" }`

Command Line Arguments: Example

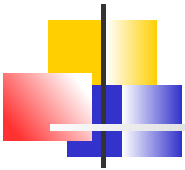
```
1  public static void main(String[] args) {
2      . . .
3      for (String arg: args) {
4          if (arg.charAt(0) == '-') { // command line
5                                      // option
6              if (arg.charAt(1) == 'd')
7                  key = -key;
8              else {
9                  printUsage();
10                 return;
11             }
12         }
13         else { // file name
14             . . .
```

see Section 7.3, pp. 331–332



Command Line Arguments

```
1  public class Echo { // echoes command-line arguments
2      public static void main(String[] args) {
3          if (args.length == 0) return; // print nothing
4          // -n as first argument: no newline at the end
5          int start = args[0].equals("-n") ? 1 : 0;
6          for (int i = start; i < args.length; i++) {
7              System.out.print(args[i]);
8              if (i == args.length - 1) { // last one?
9                  if (! args[0].equals("-n"))
10                     System.out.println();
11             }
12             else System.out.print(" ");
13         }
14     }
15 }
```



Take Home Messages

- use `Scanner` for reading and `PrintWriter` for writing text files
- close all files when you are done processing them
- when an exception is thrown (`throw` statement), processing continues in an exception handler (`catch` clause)
- once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed
- checked exceptions are due to external circumstances
- programs that start from the command line receive the command line arguments in the `main` method