Objektorienterad programmering

Lecture 8: dynamic lists, testing and error handling

Dr. Alex Gerdes | Dr. Carlo A. Furia

SP1 2017/18

Chalmers University of Technology

In the previous lecture 7

- Reading and writing (text) files
- Multidimensional arrays
- Introduction to dynamic lists with `ArrayList`

Dynamic lists: `ArrayList`

Class `ArrayList`

- An array is a *static data structure*, whose size is fixed upon creation and cannot be changed while the program executes. In some applications, we may not know the size of the data when the program starts, and thus we need *dynamic data structures*, whose size can grow and shrink as the program needs.

- We could “simulate” a dynamic structure using an array, for example:
  - Create an array that is as large as the maximum data size
  - Keep track of how which elements are actually added to the array
  - To add an element: use an empty slot
  - To remove an element: mark a slot as empty

- Class `ArrayList` is a library class that provides a flexible implementation of dynamic list data structure. Whenever we need dynamic data management, it’s usually much simpler to use `ArrayList` instead of (monodimensional) arrays

- `ArrayList` is in package `java.util`

Class `ArrayList`

- Clarr `ArrayList` is *generic*.This means that we can create lists of elements of any type (like for arrays). When we declare a variable of type ArrayList, we also declare the type of its elements. Examples:

```
ArrayList<String> words = new ArrayList<String>();
ArrayList<Integer> values = new ArrayList<Integer>();
ArrayList<BigInteger> bigValues =
                        new ArrayList<BigInteger>();
ArrayList<Person> members = new ArrayList<Person>();
```

- `ArrayList` can only store object/reference types, not primitive types (such as `int`, `double`, `boolean` and `char`)

- When we need a list with elements of a primitive type, we use its corresponding *wrapper* type instead

Class `ArrayList<E>`

Operation	Description
`ArrayList<E>()`	Create an empty `ArrayList` for elements of type `E`
`void` `add(E elem)`	Add elem as last element of the list (after all other elements)
`void` `add(`**`int`** `pos, E elem)`	Insert elem at position pos in the list, shifting all other elements at the insertion point to the right
…	…
`E get(`**`int`** `pos)`	Return element at position pos
`E set(`**`int`** `pos, E elem)`	Replace the element currently at pos with elem
`E remove(`**`int`** `pos)`	Remove the element at position pos, shifting all other elements at the removal point to the left

Class `ArrayList<E>`

Operation	Description
`int size()`	Return the number of elements in the list
`boolean isEmpty()`	Return true if the list is empty, otherwise return false
`int indexOf(Object elem)`	Return the position (index) of elem in the list; if elem is not in the list, return -1
`boolean contains(Object elem)`	Return true of elem is in the list, otherwise return false
`void clear()`	Remove all elements in the list
`String toString()`	Return a textual representation of the list content in the form $[e_1, e_2, \ldots, e_n]$

Methods indexOf and contains compare elem to the elements in the list using a method

```
public boolean equals(Object obj)
```

which must be defined for the class E. All standard classes such as `String`, `Integer` and `Double`, include a definition of equals.

Autoboxing and autounboxing

- We can often mix primitive types and their corresponding wrapper types thanks to *autoboxing* and *autounboxing*

- ```
  Integer talObjekt = new Integer(10);
                      // without autoboxing
  ...
  int tal = talObjekt.toValue();
                    // without autounboxing
  ```

Equivalently, and more simply:

```
Integer talObjekt = 10;     // autoboxing
...
int tal = talObjekt;        // auto-unboxing
```
```

For loop over collections (for each)

- A special form of the for loop is convenient to loop over every element of an array, or of a list like ArrayList

```
double[] values = new double[100];
ArrayList<String> listan = new ArrayList<String>();

// For loops with explicit index
for (int index = 0; index < values.length; index = index +1) {
  System.out.println(values[index]);
}
for (int pos = 0; pos < listan.size(); pos = pos +1) {
  System.out.println(listan.get(pos));
}

// For loops with "for each"
for (double v : values)          // for each v in values
  System.out.println(v);

for (String str : listan)        // for each str in listan
  System.out.println(str);
```

Example: read a set of numbers

- Write a method with signature

  ```
  private static ArrayList<Integer> readSet()
  ```

  that reads integers in any order, and returns a list where all read integers appear exactly once:

  - If an element is read multiple times, it appears only once in the output
  - If an element is read once, it appears once in the output
  - If an element is not read, it does not appear in the output

- **Example**: input integers
  1, 4, 1, 2, 4, 5, 12, 3, 2, 4, 1

  output list:
  1, 4, 2, 5, 12, 3

Analysis and implementation

- **Algorithm**:
  1. while (there are more integers)
     1. Read the next number
     2. if (the number is not already in the output list)
        add the number to the list;
  2. Return the output list

```
public static ArrayList<Integer> readSet() {
  ArrayList<Integer> set = new ArrayList<Integer>();
  Scanner in = new Scanner(System.in);
  while (in.hasNextInt()) {
    int value = in.nextInt();
    if (!set.contains(value)) {
      set.add(value);
    }
  }
  return set;
}
```

Class PhoneBook implemented with arrays

```
public class Entry {
  private String name;
  private String number;
  public Entry(String name, String number) {
    this.name = name;
    this.number = number;
  }
  public String getName() {
    return name;
  }
  public String getNumber() {
    return number;
  }
}
```

```
public  class PhoneBook {
  private Entry[] book;
  private int count;
  public PhoneBook(int size) {
    book = new Entry[size];
    count = 0;
  }
  public void put(String name, String nr) {
    book[count] = new Entry(name, nr);
    count = count + 1;
  }
  public String get(String name) {
    String res = null;
    for (int i = 0; i < count; i = i +1)
      if (name.equals(book[i].getName()))
        res = book[i].getNumber();
    return res;
  }
}
```

Maximum number of entries

Actual number of stored elements

Runtime error if count >= size

Class `PhoneBook` implemented with `ArrayList`

```
public class Entry {
  private String name;
  private String number;
  public Entry(String name, String number) {
    this.name = name;
    this.number = number;
  }
  public String getName() {
    return name;
  }
  public String getNumber() {
    return number;
  }
}
```

```
import java.util.ArrayList;

public class PhoneBook {
  private ArrayList<Entry> book = new ArrayList<Entry>();

  public void put(String name, String nr) {
    book.add(new Entry(name, nr));
  }

  public String get(String name) {
    String res = null;
    for (Entry e : book)
      if (name.equals(e.getName()))
        res = e.getNumber();
    return res;
  }
}
```

Shorthand operators

Shorthand operators

- Shorthand operators are more concise forms of assignments
- There are shorthand operators for *increment* and *decrement*, each in *prefix* and *postfix* version

Shorthand	***Full form***
`++x`	`x + 1`
`--x`	`x - 1`
`x++`	`x + 1`
`x--`	`x - 1`
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Shorthand operators

- The difference between the prefix and postfix operators is *when* the increment or decrement is executed within an expression

- With the *prefix* operators, the increment/decrement occurs first, and then the whole expression is evaluated:

```
firstNumber = 10;
secondNumber = ++firstNumber;
```

  In the end, firstNumber == secondNumber == 11

- With the *prostfix* operators, the whole expression is evaluated first, and then the increment/decrement occurs (without affecting the value of the expression)

```
firstNumber = 10;
secondNumber = firstNumber++;
```

  In the end, secondNumber == 10 and firstNumber == 11

The most common, and simple, usage of the prefix/postfix operators is as stand-alone statements:

```
++firstNumber; firstNumber++;
```

- The behavior of complex combinations of pre- and postfix operators can be quite tricky. Rule of thumb: only use them in simple expressions!

Testing

What is testing?

- How do we know if our program works correctly? *By testing it!*
- The modular design of programs (decomposition into methods) helps testing:
  - ideally, we can test each method independent of the others

```
import javax.swing.*;

public class Postage {
  public static void main(String[] args) {
    String input =
      JOptionPane.showInputDialog("Weight:");
    double weight = Double.parseDouble(input);
    String output;

    if (weight <= 0.0)
      output = "Weight must be positive!";
    else if (weight <= 20.0)
      output = "Postage is 5.50 kronor.";
    else if (weight <= 100.0)
      output = "Postage is 11.00 kronor.";
    else if (weight <= 250.0)
      output = " Postage is 22.00 kronor.";
    else if (weight <= 500.0)
      output = "Postage is 33.00 kronor.";
    else
      output = "Too heavy: use a packet.";

    JOptionPane.showMessageDialog(null, output);
  }
}
```

Modular design of `Postage`

```
import javax.swing.*;

public class Postage {
  public static void main(String[] args) {
    String input  = JOptionPane.showInputDialog("Weight:");
    double weight = Double.parseDouble(input);
    JOptionPane.showMessageDialog(null, getPostage(weight));
  }

  public static String getPostage(double weight) {
    String res;
    if (weight <= 0.0)
      res = "Weight must be positive!";
    else if (weight <= 20.0)
      res = "Postage is 5.50 kronor.";
    else if (weight <= 100.0)
      res = "Postage is 11.00 kronor.";
    else if (weight <= 250.0)
      res = "Postage is 22.00 kronor.";
    else if (weight <= 500.0)
      res = "Postage is 33.00 kronor.";
    else
      res = "Too heavy: use a packet.";
    return res;
  }
}
```

Kinds of errors: static vs. dynamic

When do program errors show up?

- *At compile time (static errors):*
  - the Java compiler checks that certain correctness rules are followed everywhere in a program: types are used correctly, functions return values, variables are initialized before being used, ...
  - thus, the compiler guarantees that certain kinds of errors cannot occur (the program won't compile until we fix those errors)
- *At runtime (dynamic errors):*
  - *exceptions*: something unexpected happens, which the program cannot handle correctly: a file is missing, the network is down, etc.
  - *functional/logical errors*: the program runs without apparent failure, but it does not do what it is supposed to do: a sorting program does not correctly sort the input, it writes an empty file, etc.

Kinds of errors: failure, fault, mistake

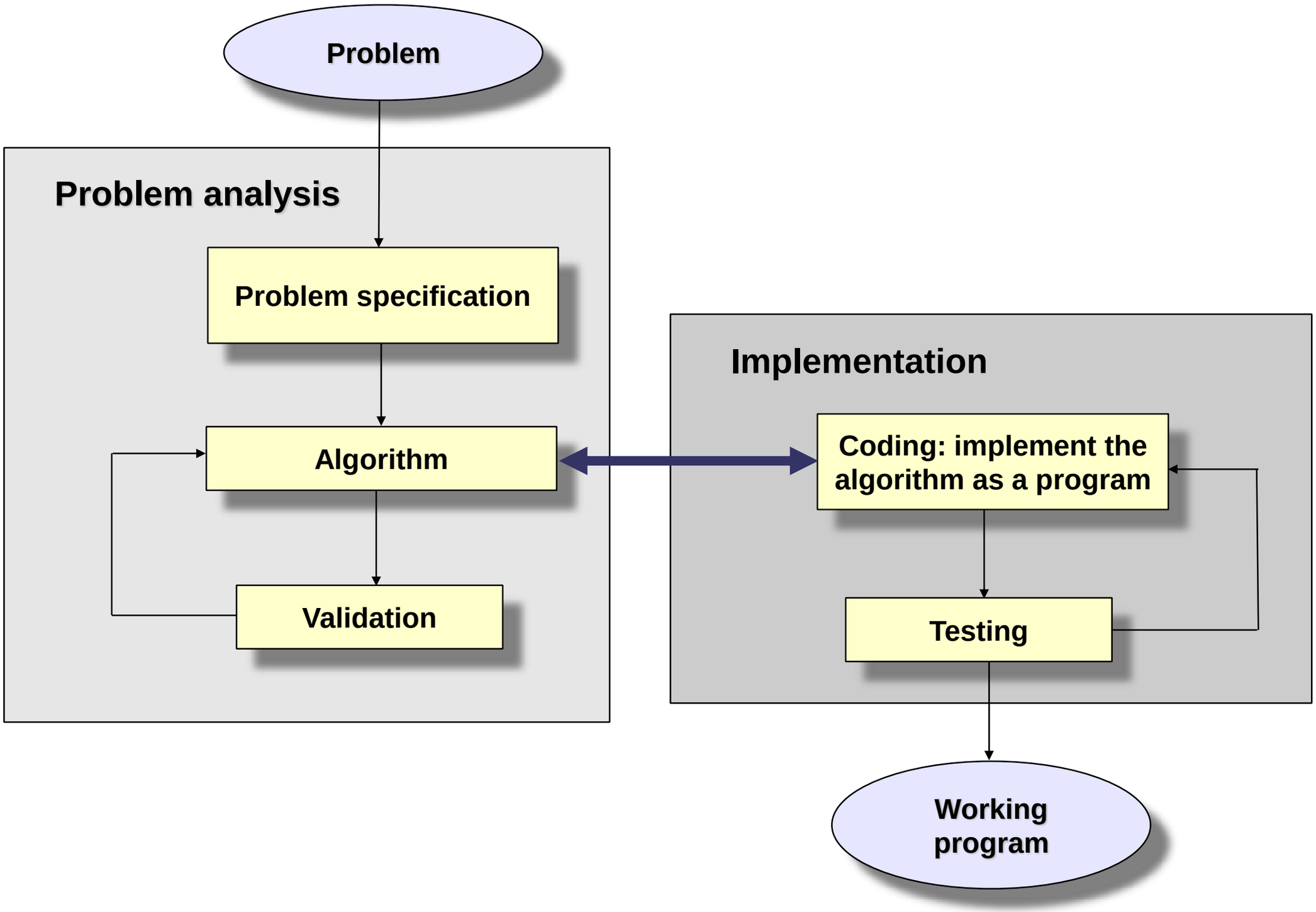
- *Failure:*
  - the program does not behave as expected: it returns the wrong output, crashes, or does not terminate
- *Fault:*
  - a program condition (state) that will cause a failure: the wrong if branch is entered, an array's bounds are incorrectly computed, etc.
- *Mistake:*
  - a programmer produces the wrong code, which will determine a fault

- *Error* or *bug*:
  - more informal terms, which can mean any of the above

What can we test?

- Testing is a fundamental practice to reduce the number of errors in programs
- What can be discovered by testing and what can be discovered by static checks (the compiler) depends to some extent on the features of the programming language we're using
  - e.g. Java vs. Python
- The main goal of testing is *finding errors* ("bugs"): testing cannot establish that there are no more errors left!
- Effective testing requires that we know what a program is supposed to do (the program's *specification*). This typically comes from the problem analysis phase of programming.

Programming phases

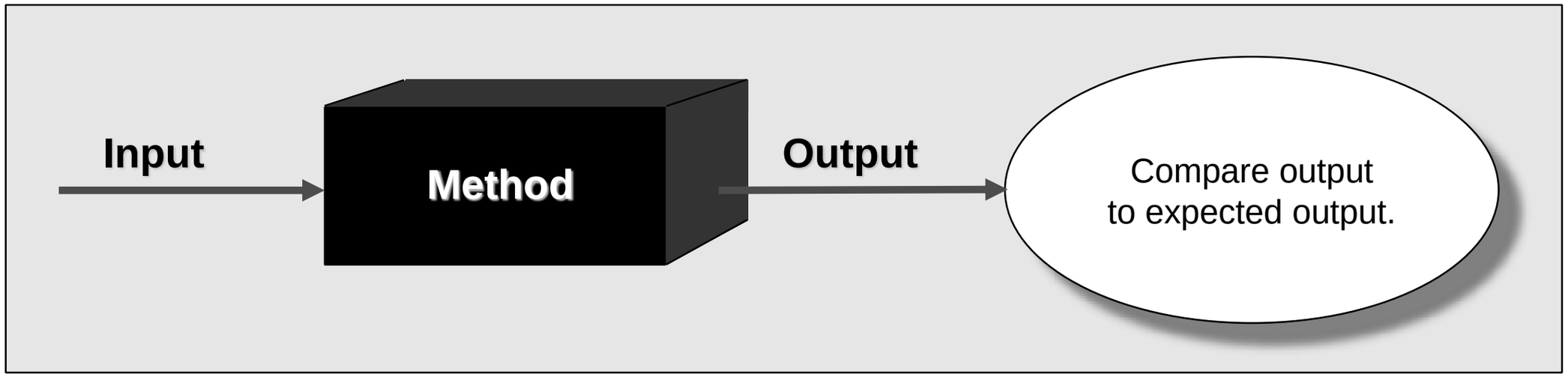


When to test?

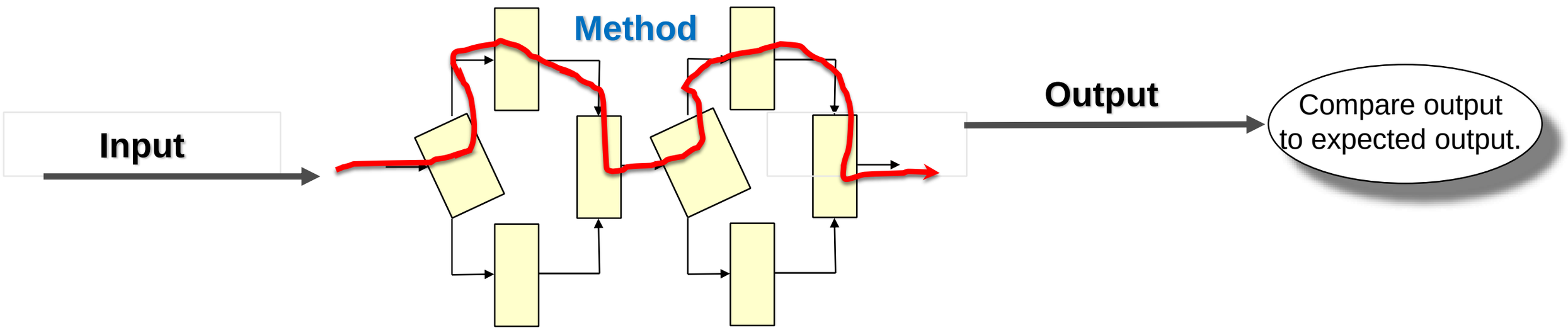
- As soon as possible!
  - The later we detect an error, the harder it is to fix
  - If the algorithm is correct, we rule out logical errors, which are usually the hardest to detect and fix
- If an error detected by testing a program turns out to be due to an error in the specification/algorithm, we may have to go back to the program analysis phase

Black-box vs. white-box testing

- In *black-box* testing, we only look at the input/output behavior of each method or program, ignoring the internal details of the code implementing the method.



- In *white-box* testing, we choose inputs in a way that explores as many portions of the code implementing the method we are testing as possible.



Testing heuristics

- A *heuristics* is a rule of thumb, which often works well in practice but is not guaranteed to succeed

- A useful heuristics for testing a method:

  - *partition* the method's inputs into a finite number of *classes*

    - according to the program logic

    - according to the program structure

- pick (at least) a *concrete input in each class*, and test the method with that input

Example of partition testing

- We are testing a program that determines if a person can vote or not given their age
- An obvious partition of the input is in two classes:

- In addition, we want to test for invalid inputs (negative age)
- Concrete test inputs that we try:

Test nr	Input	Expected output
1	12	Cannot vote
2	24	Can vote
3	-7	Invalid input
4	0	Cannot vote
5	17	Cannot vote
6	18	Can vote
7	-1	Invalid input

Testing boundary values between classes is often useful to check corner cases!

Testing of Postage

CHALMERS

```
public class TestPostage {
 public static void main(String[] args) {
    boolean res =
      testPostage(0,     "Weight must be positive!") &&
      testPostage(0.5,   "Postage is 5.50 kronor.") &&
      testPostage(20.0,  "Postage is 5.50 kronor.") &&
      testPostage(20.5,  "Postage is 11.00 kronor.") &&
      testPostage(100.0, "Postage is 11.00 kronor.") &&
      testPostage(100.5, "Postage is 22.00 kronor.") &&
      testPostage(250.0, "Postage is 22.00 kronor.") &&
      testPostage(250.5, "Postage is 33.00 kronor.") &&
      testPostage(500.0, "Postage is 33.00 kronor.") &&
      testPostage(500.5, "Too heavy: use a packet.");

    if (res) System.out.println(" All tests passed!");
  }

  public static boolean testPostage(double weight, String expected) {
    String  result = Postage.getPostage(weight);
    boolean passed = expected.equals(result);
    if (passed)
      System.out.print(".");
    else {
      System.out.println("Error with weight: " + weight);
      System.out.println("Expected output: " + expected);
      System.out.println("Program output: " + result);
    }
    return passed;
  }
}
```

Unit testing vs. system testing

- Testing of a complex, large program is done in phases:
  - *unit testing*: test the individual units (such as individual methods and classes)
  - *system/integration testing*: test the whole system where units are connected to achieve an overall functionality
- Development and testing of a large program can be:
  - *bottom-up*: start developing and testing individual components, and combine them in increasingly more complex units
  - *top-down*: start designing the overall system and test how its components interact *before* fully developing the components. Add details until every component is complete

Exceptions (undantag)

Exceptional behavior

- The *exceptional behavior* of a program is when unexpected conditions occurr during execution (at runtime) that make it impossible to continue

  - One example:
    out of bound access `ar[ar.length + 3]`

- In Java programs, exceptional behavior is signaled by objects of specific classes called *exceptions* (or exception objects)

- *Exception handling* code is the part of a program that deals with exceptions – for example trying to execute some backup actions

Sources of exceptional behavior

- Possible sources of exceptional behavior
  - Code problems that are not checked by the compiler: an object is null, an array is accessed outside its bounds, a method is called with invalid arguments...
    - These are really programming errors
  - Problems with accessing resources: the network is down, a file is missing, we are out of memory, the CD is not inserted in the reader...
    - These are not real errors but more like unexpected conditions: the best the program can do is trying again

Exceptional behavior: example

```java
import javax.swing.*;
public class Square {
  public static void main (String[] args) {
    String indata = JOptionPane.showInputDialog("Type an integer:");
    int tal = Integer.parseInt(indata);
    int res = tal * tal;
    JOptionPane.showMessageDialog(null, "The square is " + res);
  }
}
```

- If we type a string that cannot be interpreted as integer (such as `12.4` or `"help"`), `parseInt()` throws an exception that makes the whole program fail:

```
 Exception in thread "main" java.lang.NumberFormatException: For input string: "12.4"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:458)
at java.lang.Integer.parseInt(Integer.java:499)
at Square.main(Square.java:5)
```

Exception classes in Java

- Some common exception classes in Java, and what they are used for:
  - `ArrayIndexOutOfBoundsException`
    - The program tried to access an array `a` at an invalid index (negative or greater than `a.length - 1`)
  - `NullPointerException`
    - An object reference has the default value `null`, which does not correspond to a valid object
  - `NumberFormatException`
    - The input cannot be interpreted as a number of the right type
  - `IllegalArgumentException`
    - A method is called with an argument value that is not allowed by the method's specification (for example, computing the maximum of an empty list).

Exception handling in Java

There are two sides to exception handling:

- Supplier side: a method can:
  - throw (also “raise”) an exception to signal exceptional behavior to its caller (`throw` statement)
  - *optionally*, declare in its signature that it may throw exceptions of certain types (`throws` clause)
- Client side: any piece of code can:
  - react to (also “catch”) an exception thrown by some called methods (`try`, `catch` and `finally` clause)

Throwing exceptions

- A class `Account` models a bank account. A method `withdraw` decrements the `balance` balance by `amount`. If we try to withdraw more money than there is in the account, we throw an exception.

```
public class Account {
  private int balance;
  // ...

  public void withdraw(int amount) {
    if (amount < balance) {
      balance = balance - amount;
    } else {
      throw new IllegalArgumentException("Not enough money!");
    }
  }
}
```

Object of class `IllegalArgumentException` which includes a message

Declaring exceptions

- Method `withdraw` can (but doesn't have to) declare in its signature that it may throw an exception of illegal argument. This way, callers know that they may have to handle that exception.

```
public class Account {
  private int balance;
  // ...

  public void withdraw(int amount) throws IllegalArgumentException {
    if (amount < balance) {
      balance = balance - amount;
    } else {
      throw new IllegalArgumentException("Not enough money!");
    }
  }
}
```

Handling exceptions

- The `try-catch-finally` statement supports exception handling:

```
try {
   // Code that may throw exceptions
} catch (ExceptionType e) {
  // Code executed when an exception of class ExceptionType is thrown in try
} finally {
  // Code executed after the try-catch block,
  // regardless of whether an exception was thrown
}
```

1. The code in the `try` block executes
2. If an exception is thrown in 1., the code in the `catch` block executes
   - There may be multiple `catch` blocks for different exception classes
   - Only the `catch` block of the exception class that was thrown is executed
   - If no suitable `catch` block exists, the exception is *propagated*, as if it was not caught
3. Regardless of whether a catch block was executed, the finally block executes before continuing

Handling exceptions: example

```
import javax.swing.*;

public class FaultTolerantSquare {
  public static void main(String[] args) {
    boolean done = false;
    while (!done) {
      String indata = JOptionPane.showInputDialog("Input an integer:");
      try {
        int tal = Integer.parseInt(indata);
        int res = tal * tal;
        JOptionPane.showMessageDialog(null, "The square is " + res);
        done = true;
      } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(null, "Invalid integer. Try again!");
      }
    }
  }
}
```

Exception classes

- Java offers several exception classes for common errors
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - `NumberFormatException`
  - `IllegalArgumentExecption`
  - ...

- When constructing an object of an exception class, we can pass a string to the constructor to serve as error message
  - method `getMessage()` returns the message string
  - method `printStackTrace()` prints the sequence of calls that lead to an exception
    - it is printed when you run a program that terminates with an uncaught exception that propagates to the `main` method

Checked vs. unchecked exceptions

- Java exception classes are partitioned in two categories:
  - unchecked exceptions, such as `ArithmeticException`, `NullPointerException`, and `IllegalArgumentException`, do *not have* to be declared or handled
  - checked exceptions, such as `FileNotFoundException`, have to be *declared* and *handled* (the compiler checks this)
- Even with unchecked exceptions, it's good practice to declare them at least in the documentation (e.g. comments)
- We will see a few more details about exceptions in the second part of the course

Checked exceptions: example

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ReadFromTextFile {
  public static void main(String[] args) throws FileNotFoundException {
    System.out.println("The sum is: " + sumInFile("indata.txt"));
  }

  private static int sumInFile(String fileName) throws FileNotFoundException {
    File in    = new File(fileName);
    Scanner sc = new Scanner(in);
    int sum = 0;

    while (sc.hasNext()) {
      sum = sum + sc.nextInt();
    }

    return sum;
  }
}
```

Have to declare checked exception

May throw checked `FileNotFoundException`

May throw unchecked `InputMismatchException`

Checked exceptions: example

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFromTextFile2 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    boolean done = false;
    while (!done) {
      System.out.print("Give filename: ");
      String fileName = sc.next();
      try {
        System.out.println("The sum is: " + sumInFile(fileName));
        done = true;
      } catch (FileNotFoundException e) {
        System.out.println("File doesn't exist!");
      }
    }
  }

  private static int sumInFile(String fileName) throws FileNotFoundException {
    // as before
  }
}
```

Have to handle checked exception

May throw checked FileNotFoundException

Example problem

Problem statement

- Write a method

```
public static int[] reverse(int[] arr)
```

that inputs an integer array `arr` and returns a copy of the input where the elements appear in reverse order; if `arr` is `null`, the method throws an `IllegalArgumentException`.

- **Example**: given inputs:

```
int[] f1 = {1, 6, 3, 9};
int[] f2 = null;
```

- `reverse(f1)` returns [`9, 3, 6, 1`]
- `reverse(f2)` throws an `IllegalArgumentException`

Analysis and design

- **Analysis**: we create a new array `rev` of the same size as the input array `arr`; we fill in `rev` oin reverse order (from the right-hand side) with the same elements as `arr`

- **Algorithm**:
  1. if `arr` is `null`, throw exception
  2. initialize `rev` with the same size as `arr`
  3. for each index value `k` from 0 to `arr`'s length:
     1. set `rev[arr.length - k - 1]` to `arr[k]`
     2. switch to the next value of `k`

Implementation

```java
import java.util.Arrays;

public class Reversal {
  public static void main(String[] args) {
    int[] f1 = {1, 2, 3, 4};
    int[] f2 = null;

    System.out.println(Arrays.toString(reverse(f1)));
    System.out.println(Arrays.toString(reverse(f2)));
  }

  public static int[] reverse(int[] arr) {
    if (arr == null)
      throw new IllegalArgumentException();

    int[] rev = new int[arr.length];
    for (int k = 0; k < arr.length; k += 1) {
      rev[arr.length - k - 1] = arr[k];
    }
    return rev;
  }
}
```