ENSF 593 Lecture Notes

Yves Pauchard

June 15, 2020

Contents

1 Overview **4**
- 1.1 Goals of this course . . . 4
- 1.2 Java, Java, Java by Morelli and Walde . . . 4
- 1.3 Setting up tools . . . 4

2 Working with git **5**
- 2.1 References . . . 5
- 2.2 Configure git . . . 5
- 2.3 Tracking changes . . . 6
- 2.4 Working with remotes . . . 9

3 Chapter 0 **10**
- 3.1 Why Java? (Sec 0.6) . . . 10
- 3.2 What is OO programming (Sec 0.7) . . . 10

4 Chapter 1 **11**
- 4.1 Exercises . . . 12

5 Chapter 2 **13**
- 5.1 Goal . . . 13
- 5.2 Section 2.2 Strings . . . 13
- 5.3 Section 2.3 AWT . . . 14
- 5.4 Section 2.4 Class design . . . 14
- 5.5 Section 2.5 Case study OneRowNim . . . 14
- 5.6 Section 2.6 Scanner . . . 14
- 5.7 Exercises . . . 15

6 Chapter 3 **15**
6.1 Goal . . . 15
6.2 Section 3.2 Passing information to an object . . . 15
6.3 Section 3.3 Constructor . . . 16
6.4 Section 3.4 Retrieving information . . . 16
6.5 Section 3.5 passing a value and a reference . . . 16
6.6 Section 3.6 Control flow . . . 16
6.7 Section 3.7 Testing . . . 16
6.8 Section 3.8 . . . 17
6.9 Section 3.9 Inheritance and polymorphism . . . 17
6.10 Section 3.10 Draw lines . . . 17
6.11 Exercises . . . 17

7 Chapter 4 **18**
7.1 Goals . . . 18
7.2 4.2 User interface . . . 18
7.3 4.3 Command line interface . . . 18
7.4 4.4 Graphical User Interface . . . 19
7.5 4.5 OneRowNim with interface . . . 19
7.6 4.6 FileIO . . . 20
7.7 Exercises . . . 20

8 Chapter 5 **20**
8.1 Goal . . . 20
8.2 5.2 Boolean Data Operators . . . 21
8.3 5.3 Numeric Data and Operators . . . 22
8.4 5.4 Java Math . . . 22
8.5 5.5 Numeric processing examples . . . 23
8.6 5.6 NumberFormat . . . 23
8.7 5.7 Character data and operators . . . 24
8.8 5.8 Character conversion . . . 24
8.9 5.9 Representation is important . . . 24

9 Chapter 6 **24**
9.1 Goal . . . 24
9.2 6.4 Example car loan . . . 25
9.3 6.6 Conditional loops . . . 25
9.4 6.7 Example computing averages . . . 26
9.5 Exercises . . . 26

10 Chapter 7 **26**
10.1 Goal . . . 26
10.2 7.2 String basics . . . 27
10.3 7.3 Finding things in strings . . . 28
10.4 7.4 Keyword search . . . 28
10.5 7.5 StringBuffer . . . 28
10.6 7.6 Extracting parts of strings . . . 28
10.7 7.8 StringTokenizer . . . 29
10.8 7.9 Comparing strings . . . 29

11 Chapter 8 **29**
11.1 Goal . . . 29

12 Chapter 9 **31**
12.1 Goals - arrays . . . 31
12.1.1 Pure vs modifier methods . . . 31
12.1.2 String split rather than StringTokenizer . . . 32
12.1.3 2D arrays . . . 33
12.1.4 Vector and ArrayList . . . 33
12.2 Goals - search and sort . . . 34
12.3 9.2 One-Dimensional Arrays . . . 35
12.4 9.4 Counting character frequencies . . . 35
12.5 9.5 Sorting . . . 35
12.6 9.6 Searching . . . 35
12.7 9.7 2D arrays . . . 36
12.8 9.9 polymorphic sort . . . 36
12.9 9.10 Vector . . . 37

13 Chapter 10 **37**
13.1 Goals . . . 37
13.2 10.2 Handling exceptions . . . 40
13.3 10.3 Exception hierarchy . . . 40
13.4 10.4 Handling exceptions . . . 41
13.5 10.5 Error handling . . . 41

1 Overview

1.1 Goals of this course

We would like you to be able to program with an object oriented language, Java, and understand and use UML in the design process.

The course will adopt an object first view and use an existing textbook (see next section).

We will meet Mon/Wed mornings for lectures. The assumption is that you prepare by reading the assigned chapter in the book. During the lecture the key concepts will be reviewed using examples.

Every week, there will be a new lab assignment to apply the learned concepts.

Finally, there is a quiz every week.

1.2 Java, Java, Java by Morelli and Walde

The book is available for free as a PDF here: `https://open.umn.edu/opentextbooks/textbooks/java-java-java-object-oriented-problem-solving`

1.3 Setting up tools

We need:

- Java JDK like OpenJDK 8 with HotSpot JVM

`https://adoptopenjdk.net/installation.html`

- git (with gitbash on Windows)

`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

- Eclipse 2020-03

`https://www.eclipse.org/eclipseide/` On Windows, I had to navigate to the OpenJDK javaw.exe in `C:\Program Files\AdoptOpenJDK\jdk-8.0.252.09-hotspot\bin\`

- Eclipse PlantUML plugin

`http://hallvard.github.io/plantuml/`

- Graphviz to support PlantUML plugin

`https://graphviz.gitlab.io/download/`

- (optional) Eclipse TM terminal plugin
  1. In Eclipse, Help –> Eclipse Marketplace –> type terminal in find
  2. Install TM Terminal ...
  3. Restart Eclipse
  4. Window –> Show View –> Terminal
- Github account

2 Working with git

2.1 References

First, there is the git book: `https://git-scm.com/book/en/v2`, a complete reference to git.

When I first learned git, I followed the git immersion online tutorial available at: `http://gitimmersion.com`

The Software Carpentry course `http://swcarpentry.github.io/git-novice/` provides another step-by-step tutorial to learn git.

A short and useful introduction to git can be found in Section 2 of lecture notes available here: `http://columbia-applied-data-science.github.io/appdatasci.pdf`

One of the references used in this chapter, `http://marklodato.github.io/visual-git-guide/index-en.html`, provides a visual guide to git.

2.2 Configure git

Prior to using git on a fresh installation, it is important to configure name, email and line endings. Configurations has to be done only once. Name and email will be used to match commits on remote services like github. Line endings configurations makes sure that contributions made from Windows and unix-like systems are handled properly. To configure name and email type the following in a terminal (or git-bash on windows):

```
$ git config --global user.name "your␣name"
$ git config --global user.email "your_email@whatever.com"
```

Note that email address should match the email address you used on github. You can also use

`ID+your-user-name@users.noreply.github.com`,

if you want to keep your email private. Check settings -> emails on github.com. When you check *keep my email address private* it will show your noreply email address.

Line endings configuration[1] is different for Windows:

```
$ git config --global core.autocrlf true
$ git config --global core.safecrlf true
```

and MacOSx/Linux:

```
$ git config --global core.autocrlf input
$ git config --global core.safecrlf true
```

It looks like git with git-bash on Windows takes care of this configuration, on MacOSx/Linux you will have to configure it with the commands above.

Additionally, I like to have an alias configured to view commit history[2]:

```
$ git config --global alias.hist \
"log␣--pretty=format:'%h␣%ad␣|␣%s%d␣[%an]'␣\
--graph␣--date=short"
```

Maybe you will like it too. This will allow viewing previous commits with `git hist` in a concise format.

2.3 Tracking changes

As a little exercise, why don't we use git to track code changes for a HelloWorld example. Assuming you have `HelloWorld.java` and your terminal (git-bash on Windows) open in this directory. `ls` shows `HelloWorld.java`. Its contents are:

```
/**
 * @author Your Name
 *
 */
public class HelloWorld {

	public static void main(String[] args) {
		//TODO: Change to greet you instead of the world.
		System.out.println("Hello␣World!");

	}

}
```

[1] http://gitimmersion.com/lab_01.html
[2] http://gitimmersion.com/lab_11.html

To start tracking changes with git, we first have to initialize the repository with `git init`. This command will create a folder `.git`, you can check with `ls -a`. Now, the repository is initialized. No files are yet under version control. `git status` shows, yes, the current status. Let's check-in version one of the code. This is a two stage process: first add the change to the index (also called stage), second, commit the staged changes to the repository, the history, pointed to by a reference called HEAD. A commit includes a short message. The commands are:

```
  $ git status
On branch master

No commits yet

Untracked files:
  (use "git␣add␣<file>..." to include in what will be committed)

      HelloWorld.java

nothing added to commit but untracked files present (use "git␣add" to track)

  $ git add HelloWorld.java

  $ git status
On branch master

No commits yet

Changes to be committed:
  (use "git␣rm␣--cached␣<file>..." to unstage)

      new file: HelloWorld.java

  $ git commit -m 'Initial commit'
[master (root-commit) 38e9b3a] Initial commit
 1 file changed, 13 insertions(+)
 create mode 100644 HelloWorld.java

  $ git status
On branch master
nothing to commit, working tree clean
```

The three status commands in between are not necessary, but it is good to see what happens to the repository after each command. These different layers: working directory, stage and history can be confusing at first. The visual

representation might help `http://marklodato.github.io/visual-git-guide/index-en.html`. OK, now we make some changes, edit author name. To review the changes use `git diff`. We can now add and commit:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git␣add␣<file>..." to update what will be committed)
  (use "git␣checkout␣--␣<file>..." to discard changes in working directory)

        modified: HelloWorld.java

no changes added to commit (use "git␣add" and/or "git␣commit␣-a")

$ git add HelloWorld.java
$ git commit -m 'edited author name'
[master c11152f] edited author name
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Let's see what we have so far:

```
$ git hist
*c11152f 2020-05-08 | edited author name (HEAD -> master) [Yves]
*38e9b3a 2020-05-08 | Initial commit [Yves]
```

The latest commit appears at the top, where HEAD is. master denotes the branch we are on. Each commit has a unique hash, our history shows the tail of this hash.

Notice that git status includes additional information related to undoing steps. Undoing is explained in the visual guide `http://marklodato.github.io/visual-git-guide/index-en.html`. In summary:

- `git reset -- filename` un-stages, it copies the version in HEAD to the stage (index), and un-does staging with `git add -- filename`.
- `git checkout -- filename` copies from stage (index) to working directory, and un-does working directory changes.
- `git reset HEAD~1 --hard` will undo the last commit. HEAD~1 can be thought of HEAD minus one, go back one commit.

In the next section we will look at working with remotes. When working with remotes, the preferred way to undo the last commit is to use `git revert HEAD`, indicating the commit to be reverted, rather than `git reset HEAD~1 --hard`, moving HEAD. Reverting creates an inverse change as a new commit, hence, playing nicely with remote repositories.

Eclipse has built-in git support. All the above steps can be performed from within Eclipse, either in a terminal, or with the tools in the git view.

It is a good idea to practice committing and undoing commits in a toy repository. To remove git from a folder, delete the `.git` folder.

2.4 Working with remotes

In the previous section, we looked at tracking your changes with git on the local machine. Often there is a desire to have a copy remotely. Synchronizing your code history with a remote host makes working in teams easier. However, even if you work alone on a project, having a copy remotely adds redundancy.

There are providers such as github, gitlab, and bitbucket, which provide remote integration of git. We will look at github here as an example. In our scenario, we start by creating a new repository on the re- mote host first. In our case this is github. Once the repository is created, we clone it using the repository url with the form `git clone https://github.com/org-a/repo-a.git`. This command checks-out a copy to the local machine and sets up mechanisms to get updates from the remote location git pull, as well as, sending changes there git push.

The complete workflow looks like this:

- Create repository on remote host, e.g. github.
- `git clone <repo-url>`:

Using git-bash (Windows) or terminal (Mac/Linux), clone the repository using the repository url.

- Work on the code:
  - Make changes.
  - `git add <file(s)>` to stage changes.
  - `git commit -m 'a short message'` to save changes to the history.
- `git push` Update code on the remote host.

3 Chapter 0

3.1 Why Java? (Sec 0.6)

Developed by James Goslin. James is from Calgary and completed his BSc in CS at UofC in 1977. Or did he? Watch `https://www.youtube.com/watch?v=Q-TQKg3ler8`

Designed from scratch as an Object-Oriented language. It is platform independent

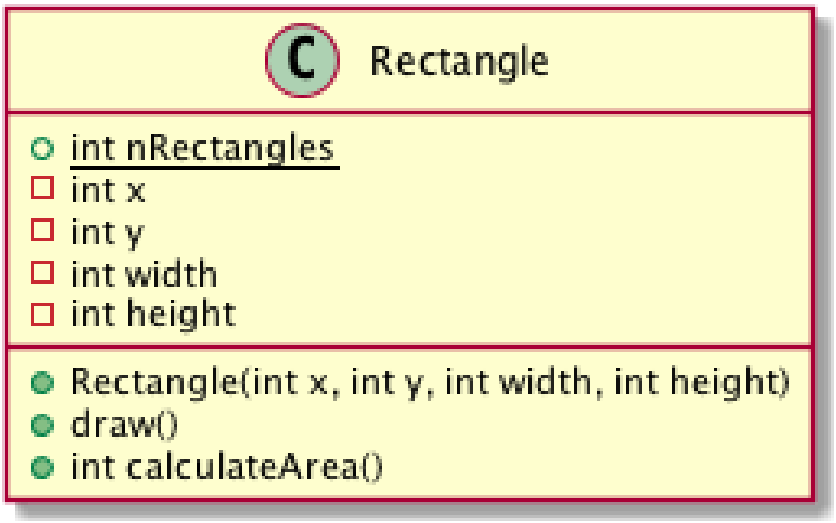
> Write once, run everywhere.

3.2 What is OO programming (Sec 0.7)

OO programming -> interacting objects.

In OO programming, data and operations are encapsulated in objects described by classes.

Objects/classes can be represented in Unified Modelling Language (UML) diagrams.



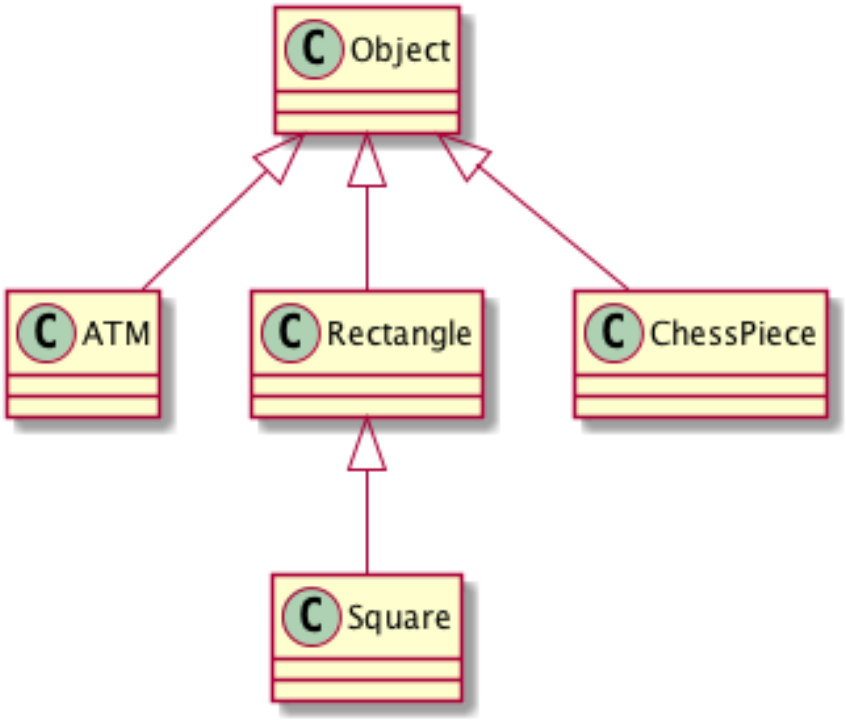
Objects have attributes/variables and actions/methods. A class is a template for objects. An object is an instance of a class.

Variables and methods can be either associated with objects or with classes. `x, y, width, height` are instance variables, different for every

`Rectangle` object. In contrast, `nRectangles` is a class variable, one variable for the duration of the program.

A `constructor` used to create objects of a certain class is an example of a class method.

Class hierarchy from most general `Object` to more specific `Square`. More specific classes inherit from super/parent and extend.



ChessPiece example illustrating inheritance, see Fig 13.
Principles of Object Oriented design. Can we list them?

4 Chapter 1

The lifecycle, see Fig. 1.1

1. Specification
2. Design
3. Implementation

4. Testing

The riddle program. Look for nouns in specification. These could be objects. Look for verbs in specification. These could be object actions.

Section 1.4: Hello World. Is OO with comments. Comments should focus on why rather than what, the code tells the what. We want to use Javadoc for class and method documentation (Ctrl-Shift-J in Eclipse). Variable, class and method names. CamelCase starting lowercase for variable and methods, and uppercase for classes.

Talks about variables as 'containers' for typed values. Note that this is only true for primitive types. Variables contain references to objects rather than the object itself. This is discussed on Sec 2.2.1.

A statement is terminated by a semicolon.

Expressions contain operators and produce values.

Class header: public ClassName extends Object. Note that all classes inherit from `Object`.

We need `main()`, which is `public` (accessible outside the class), and `static` (a class not an instance method).

Creating an object with `new` is called instantiation. We get a default constructor.

HelloWorld with a `JFrame`. Uses classes from packages `javax.swing` and `java.awt` via `import`.

Fig 1.10 shows how to edit, run and debug a java program (with an IDE). Write, compile with `javac` to get Java bytecode, run with `java`. Java Virtual Machine will interpret bytecode. With a typical Eclipse folder structure, source files are in a `src` sub-folder and compiled class files are in a `bin` sub-folder. Compile-run from command line can be done with:

```
javac -sourcepath src/ -d bin/ src/HelloWorld.java
java -cp bin/ HelloWorld
```

`java.lang.System` contains two `java.io.PrintStream` objects for output: `System.out` and `System.err`.

4.1 Exercises

Some exercises that are interesting:

- Caesar cipher in Ex 1.13
- Rectangle with area and perimeter Ex 1.18
- Temperature class Ex 1.22

- Circle UML Ex 1.27
- Triangle UML Ex 1.28
- Person, UML to Java Ex 1.29

5 Chapter 2

5.1 Goal

In this lecture I would like to:

1. Part 1:
   (a) Introduce UML and how to draw it with PlantUML
   (b) Use `Riddle` and `RiddleUser` to show relationship `uses`
   (c) Talk about the design-create-use of objects (Section 2.4.5)
   (d) Highlight `Scanner` for user input with `RiddleUser`
2. Part 2:
   (a) From `OneRowNim` problem spec 2.5.1, draw UML.
   (b) Use `OneRowNim` and review design principles (private attributes, interface), discuss tracing (Fig 2.22)
   (c) Can we add a user interface using `Scanner`?
3. Part 3:
   (a) Introduce differences between primitive types and Objects Ex 2.16

   Section 2.3 is optional at this point.

   Pre-reading Section 2.4 is important for lecture preparation.

A nice overview of class relationships: `http://usna86-techbits.blogspot.com/2012/11/uml-class-diagram-relationships.html`

5.2 Section 2.2 Strings

Strings: UML shown in Fig 2.1. `length()` and `concat()`, `equals()`. Shortcuts exist, instatiaton can be done with a literal assignment rather than using `new`. Concatenation by using `+`.

Variables of objects, not primitive types, will get a value of `null` when declared. Trying to access methods or variables will result in a null pointer exception. What are primitive types?

Can you read the code and predict the output in Fig 2.4.

5.3 Section 2.3 AWT

Challenge: Can you design a class that draws its own UML diagram? **Challenge:** Could you design a class representing a rectangle that can draw itself and include its area as text at the centre?

5.4 Section 2.4 Class design

Fig 2.11, draw this UML in eclipse in a text file with `@startuml @enduml`. Identify the attributes, methods and constructor. Which of these are private, which are public?

Fig 2.14 shows a UML with *Uses* relationship. Can you draw this in eclipse in a text file using PlantUML `@startuml @enduml`.

Fig 2.15 On which lines are `Riddle` objects instantiated? What *gives it away*?

What are the three steps in Design, Create, Use referenced in Section 2.4.5?

5.5 Section 2.5 Case study OneRowNim

Fig 2.22 tracing OneRowNim

Review 2.5.4 Object oriented design principles:

- Information hiding
- Encapsulation
- Interface
- Generality and extensibility

5.6 Section 2.6 Scanner

Using Scanner, experiment with reading different inputs, e.g. int `nextInt()`, double `nextDouble()`, String `next()` and `nextLine()`.

5.7 Exercises

Some exercises that are interesting:

- 2.8 valid identifiers
- 2.9 difference between class and instance variable
- 2.13 Syntax errors
- 2.16 pairs of concepts
- 2.19-2.21 Cube, CubeUser and input.
- 2.23 language UML
- 2.25 Object diagram (not class diagram)
- 2.26 UML to represent UI and address book

6 Chapter 3

6.1 Goal

- A good way to start would be to implement extend OneRowNim from UML in Fig 3.5, the code would be Fig. 3.16 (note that return type for takeSticks is missing).
- Overloading (constructor) and overriding (toString())
- Polymorphism toString(), study example in section 3.9
- Introduce equals() needed for assignment.
- Demonstrate pass by value and pass by reference Fig 3.10 is important. Use code snippet in Section 3.9 (see below).
- Mention if-else and while.

Section 3.10 optional at this point.
Studying code in Fig. 3.16 and 3.17 would be good as preparation.

6.2 Section 3.2 Passing information to an object

public **accessor** methods allow accessing private attributes. public **mutator** methods allow changing private attributes. scope

6.3 Section 3.3 Constructor

Constructors do not have returns -> exercise 3.5 Every object gets a default constructor. Constructors are not inherited. **overloading** Constructor overloading, match by signature.

6.4 Section 3.4 Retrieving information

return type declared in method header prior to method name. 3.4.1 shows how a return value replaced the method invocation.

Fig 3.5 shows UML of extended OneRowNim class. Can we implement the class from this UML description?

6.5 Section 3.5 passing a value and a reference

Primitive types such as int or boolean are passed by value. Changes do not have any effect in the main program.

Objects, such as OneRowNim, are passed by reference (or the reference is passed by value). Changes will have an effect outside the method. Fig 3.10 shows what happens in memory. This is important, demonstrate.

Note that String is an object as well, however, Strings are *immutable* as we will see later. The effect is that one cannot change a String from within a method.

Side effect: an unintended change to an object - should be avoided.

6.6 Section 3.6 Control flow

Two problems with OneRowNim:

1. Players need to take turns - use `while`
2. `takeSticks()` does not check rules - use `if-else`

if-else condition needs to be in parenthesis, statement block in curly braces. Indentation is not enough.

while executes the loop body as long as condition is true. Usually has initializer and updater statements.

6.7 Section 3.7 Testing

Fig 3.16 new OneRowNim. Fig 3.17 new main in a Test class.

We might want to study these two new classes

6.8 Section 3.8

Since all classes inherit public and protected methods from Object, we have `toString()` method available. To make it more useful we **override** it.

There is also `equals()` to compare objects. Note that the equals operator `==` compares the references (addresses) of objects. An overriden `equals()` takes values of instance variables into account. On a side note, true overriding would mean not to change the signature of the method. Hence, we would have to implement `equals(Object that)`.

6.9 Section 3.9 Inheritance and polymorphism

All classes inherit from Object, as seen in the previous section. All public and protected methods are inherited, and can be overridden, redefined.

Poly (many) morph (forms) ism. `toString()` is polymorphic. When overriden, it has different behaviour when invoked on different objects.

```
Object obj; // obj can refer to any Object
obj = new Student("12345");// obj refers to a Student
System.out.println(obj.toString()); // Prints "12345"
obj = new OneRowNim(11); // obj refers to a OneRowNim
System.out.println(obj.toString());
// Prints: nSticks = 11, player = 1
```

Can we write `Student` to make this example work?

6.10 Section 3.10 Draw lines

Using `drawLine()` Can you write a class NGon that can draw an n-sided regular polygon?

6.11 Exercises

1. 3.2 concept pairs
2. 3.3 if-else translation
3. 3.8 small method
4. 3.16 String permutation
5. 3.21 Inheritance
6. 3.22 Overriding

7 Chapter 4

7.1 Goals

The main points are:

- User interfaces are a separate component interfacing with a computation component.
- For command-line interface with standard in, a BufferedReader based class is developed. In practice, a Scanner object can be used as well.
- GUI understand which components are needed and how they work together.
- We will skip FileIO with a scanner object. Note that we need to handle exceptions with try-catch.

Study `KeyboardReader` UML Fig 4.6. and full code in Fig 4.7. Study code in Fig 4.22

Study code in 4.4 GUI, particularly Fig. 4.20 Greeter GUI.

7.2 4.2 User interface

Idea is that a user interface connects to a computational part (see Fig 4.1). This computational part may be designed to work with a variety of user interfaces: command line or graphical.

7.3 4.3 Command line interface

Using a `BufferedReader` we build our on keyboard reader.

```
BufferedReader input = new BufferedReader(
                      new InputStreamReader(
                         System.in));
String s = input.readLine();
```

`KeyboardReader` UML Fig 4.6. and full code in Fig 4.7. This would be good to study ahead of time.

GreeterApp UML Fig 4.8 and implementation 4.9

When I read this I wonder if a Reader should be asked to output/display data? Would it make more sense to rename the KeybardReader to CommandLineInterface? The GreeterApp would then use a CommandLineInterface. What do others think?

Review object oriented principles on bottom on p. 159

Exercise 4.1, Guess my Number is very appropriate.

7.4 4.4 Graphical User Interface

The code in Fig 4.20 summarizes the key components:

- Inherit `JFrame` to specialize the top-level window.
- Implement `ActionListener` interface to be able to handle events such as button pushes or keyboard presses.
- GUI elements are added to `JFrame` content pane. It has a `BoarderLayout` providing north, east, west, south and center locations. Only one element can be placed in each of these positions.
- `JPanel` are used to group GUI elements to be added to the content pane.
- The main GUI elements are:
  - `JTextField` for user input
  - `JTextArea` for output
  - `JLabel` for prompts or other direction
  - `JButton` to enable click events on a button.

It might be easier to start with `SimpleGUI`

```
import javax.swing.*;
public class SimpleGUI extends JFrame {
   public SimpleGUI(String title ) {
      setSize (200 ,150);
      setLocation (100 , 150);
      setTitle(title );
      setVisible(true); // Displays the JFrame
   } // SimpleGUI ()
   public static void main(String args[]) {
      new SimpleGUI("My␣GUI");
   }// main()
} // SimpleGUI class
```

7.5 4.5 OneRowNim with interface

Based on OneRowNim (see UML in Fig 4.21) implement a command line interface. A user plays against the computer.

It would be good to develop the UML of App, OneRowNim, KeyboardReader and then look at the code in Fig 4.22

I would like to skip 4.5.2 GUI interface for Nim.

7.6 4.6 FileIO

Fig 4.26 Shows the code that reads riddles from a file. We might change it to use nextLine() instead of setting the delimiter (would this make it platform dependent)?

7.7 Exercises

The self-study exercises are good to do. Try it first prior to peaking at the solution.

8 Chapter 5

8.1 Goal

- Some takeaways:
  - Boolean types in Java are only true and false, 1 and 0 are integers and do not mix.
  - Division can go wrong: 1/4 performs integer division. with mixed types, promotion helps: 1/4.0 promotes int 1 to double for double division.
  - precedence: it is always best to use parenthesis to make it clear.
  - When doing computations keep in mind finite precision, e.g. adding 10 times 1.0/10.0 does not necessarily equal 1.0.
- Study Math class: No objects instantiated, methods are static. Used in assignment.
- NumberFormat is a useful calss for automatic formatting of output. Look at some examples.

```
import java.text.NumberFormat;
import java.util.Locale;
class Main {
   public static void main(String[] args) {
      NumberFormat percent = NumberFormat.getPercentInstance();
      percent.setMaximumFractionDigits(2);
      System.out.println( percent.format(0.0655));

      NumberFormat currency = NumberFormat.getCurrencyInstance(Locale.US);
      System.out.println( currency.format(2500));
```

```
    }
}
```

- Characters `char` are different from String and can have a character and integer (16-bit unsigend) representation.
- In addition to type promotion, type conversion with cast operation exists. Casting works if no information is lost.

```
class Main {
    public static void main(String[] args) {


        double a = 65.2;
        char b = 'A';
        int c = (int)a;

        //do we need casting here or will this be promoted.
        int d = b;
        char e = c;

        System.out.println(a);
        System.out.println(b);

        System.out.println(c);
        System.out.println(d);

        System.out.println(e);

    }
}
```

8.2 5.2 Boolean Data Operators

Boolean flag. Boolean operators and `&&`, or `||`, xor `^`, not `!`. Table 5.1 summarizes truth tables for these operators. Can you remember which ones are binary, which are unary operators? Table 5.2 summarizes precedence of these operators. Best is to use parenthesis to define precedence.

Short-circuit evaluation refers to the following: if the first operand of an AND expression is false, the second does not need to be evaluated. The result

is `false` regardless of the second operand. Can find a similar explanation for OR short-circuit?

Using booleans, `OneRowNim` changing player can be simplified. `onePlaysNext = !onePlaysNext;`.

8.3 5.3 Numeric Data and Operators

In Java, data types are platform independent. There are four integer types: `byte`, `short`, `int`, `long`. Two floating point formats: `float` and `double`.

Round off error: Try and add 10 times 1.0/10. Do you get 1.0?

Can you name the five binary number operators and their symbol?

Division `/` is the one that can go wrong. How? Performing `1/4` results in zero, and 0.25 was expected. How to remedy? Hint, Java uses operator overloading. Therefore the type of the operands matter.

Modulo operation `%`. Reports the reminder of integer division. It can be used for example to determine if a number is even: `(n%2) == 0` evaluates to `true` for even numbers, `false` for odd numbers. Can you see why?

Operand promotion. Example `int / double` the first operand will be promoted, converted, to a double.

Similar to boolean operators, there is an order of execution - precedence - best is to use parenthesis to make order explicit.

Be careful with pre- and post-inc/decrement operators. Their effect is not always straight forward. Table 5.7 summarizes interpretations for these operators and cases.

Assignment operators such as `+=` provide short-hands.

Relational operators such as `==` compare two (primitive) values. Note that arithmetic operations are done prior to relational operations.

8.4 5.4 Java Math

Math functions are available in `java.lang.Math` implicitly imported in all Java programs.

All math functions are *static* methods of the `Math` class. Pause. This means we do not have to instantiate an object of type `Math` to use these functions. Furthermore, the `Math` class is *final* which means it cannot be extended or subclassed.

Table 5.11 has examples of common math functions. Try them out.

8.5 5.5 Numeric processing examples

Two ways to round a number to two decimal places: `Math.floor(num * 100.0 + 0.5)/100.0` and `Math.round(num * 100.0) / 100.0`. How would we round to one decimal place, or three? Check in table 5.11 what type the two math functions return. How does this affect the division we are performing?

`Temperature`. Some of the problems highlighted, we already went through in an earlier lab session. It is important to find good test cases. For temperature conversion, this might be freezing and boiling points of water: 0, and 100 deg C, translate to 32 and 212 deg F respectively. The conversion has a stationary point: -40 deg C is equal to -40 deg F. Another takaway is test, test, test. Testing can only detect the presence of bugs, but not their absence.

`TemperatureUI`. Can we implement the UML in Fig 5.6? In essence do Ex 5.1.

Constants should be declared as `public static final` and use all upper case, e.g. `public static final MAGIC_CONSTANT = 13;`. The example provided makes `OneRowNim` more readable and maintainable by using named constants rather than literals. What is a literal again?

This section also includes a winning algorithm for `OneRowNim`. The idea is that the number of sticks left modulo 4 equal 1 is an indication whether the game is winnable or not. Best to read through section 5.5.5 for the full explanation. The computer player uses a random number of sticks if the game is not winnable.

8.6 5.6 NumberFormat

It looks like there is an error in the percent formatting example. You need to pass 0.0655 in order to have `6.55%`. Makes sense. A short snippet to test on `https://repl.it` :

```
import java.text.NumberFormat;
class Main {
  public static void main(String[] args) {
    NumberFormat percent = NumberFormat.getPercentInstance();
    percent.setMaximumFractionDigits(2);
    System.out.println( percent.format(0.0655));
  }
}
```

If your locale is not *US* you can create this a currency formatter with `NumberFormat currency = NumberFormat.getCurrencyInstance(Locale.US);`

Certificate of deposit (CD): A principal is compounded either yearly or daily with an interest rate of `r`. The new principal is:

$$p_n = p(1 + r)^n \tag{1}$$

when compounded yearly and:

$$p_n = p(1 + r/365)^{365 \cdot n} \tag{2}$$

when compounded daily.

What if you compound continuously?

Exercise 5.8, designing a UI to get yearly and daily compounds would be a great exercise.

8.7 5.7 Character data and operators

In Java `char` is an unsigned 16-bit integer representing a unicode character. Character literals such as `'a'` are treated as characters unless we explicitly cast them to an integer using `(int)'a'`. Note that when assigning a `char` to and `int` Java implicitly casts the datatype as no information is lost. In general, Java permits implicit conversion from a narrower type to a wider type. If we would like to cast and integer (wider type) to a char (narrower type) it has to be explicit and the programmer is responsible for handling potential information loss.

Characters are ordered: '0' - '9', 'A' - 'Z', then 'a' - 'z' with increasing integer value.

8.8 5.8 Character conversion

The code on Figure 5.12 shows how to use the char-is-integer-and-ordered fact to convert characters to upper case as well as turning a character digit to the corresponding integer.

8.9 5.9 Representation is important

9 Chapter 6

9.1 Goal

- Discuss the three loop structures (using Ex 6.9):
  - When number of repetition is known -> for.

  - When repetition is not known, and loop might be skipped -> while.
  - When repetition is not known, and loop is executed at least one time -> do-while.

- How could the loop for calculating averages on page 266 be written differently?

```
prompt
while(condition)
        compute
        prompt
```

- Introduce break (and continue):

```
prompt
while(true)
        if(condition)
                compute
        else
                break
        prompt
```

Is it bad style to use these constructs? What do you think? What does the web say? `https://softwareengineering.stackexchange.com/questions/58237/are-break-and-continue-bad-programming-practices`

- Review data validation example.

9.2 6.4 Example car loan

Printing a table, we are using nested for loops. Rows are handled by the outer loop, columns by the inner loop.

9.3 6.6 Conditional loops

While -> Collaz conjecture.
do-while -> Lawn problem.

9.4 6.7 Example computing averages

Grades are read until a sentinel value is entered, e.g. 9999 or -1. Grades are positive, bounded values, any negative value or positive outside the range would do as a sentinel.

A while loop is chosen because there might not be any entries. However, we need to do a read prior to the loop. This is referred to as *priming*. Looking at the pseudo code on page 266, can you find another way to write the loop?

A helper method takes care of prompt and read.

9.5 Exercises

- Self study Ex. 6.9: Number theory problem that terminates at 0.
- Self study Ex 6.13: Pizza price entry method using codes 1, 2, 3 and 0 to terminate.
- Exercise 6.1, 6.2, 6.3, 6.18, 6.20

10 Chapter 7

10.1 Goal

We would like to review the following concepts:

- Basic string operations including indexing and searching.
- Because strings are immutable, we want to use StringBuffer variables whenever we need to change strings. Review StringBuffer methods.
- Look at the counting char example, reverse example in section 7.8

We can revisit the random character application we wrote in lecture 5

```
public class RandomCharApp {

   public static void main(String[] args) {

      int numberOfChars = 10;
      char randomChar = 'A';
      int i = 0;
      String password ="";

      while(i < numberOfChars) {
         //Create a random number and convert it to an upper case character
```

```
            // adding 1 to difference to handle random edge case 1.0 not included.
            randomChar = (char)('A' + (1+'Z'-'A') * Math.random());
            password += randomChar;
            i++;
        }
        System.out.println(password);
    }
}
```

We should use StringBuffer. Maybe we can have a string literal with all characters allowed. In the loop, we would generate a random number and index into the literal string, append the character to the StringBuffer.

Comparing strings, identity vs equality. Selfstudy exercise 7.16. Drawing object diagrams similar to Fig. 7.15.

StringTokenizer, by default splits on space to return words. It can be used to split sentences:

```
import java.util.StringTokenizer;

class Main {
    public static void main(String[] args) {
        String s = "hello. I like this!\nwhat about you?";
        StringTokenizer st = new StringTokenizer(s, ".?!");

        int nSentences = 0;
        while(st.hasMoreTokens()){
            System.out.println(st.nextToken().trim());
            nSentences++;
        }
        System.out.println("Text has " + nSentences + " sentences.");
    }
}
```

For pig latin, we need to find if there is at least one vowel. How could this be implemented?

10.2 7.2 String basics

- Idea of copy constructor
- String indexing, use charAt() and 0 to length()-1.
- Discussion on literals, each literal is created only once.
- String.valueOf() to convert other types to strings. Can also use

`""+variable` as the second argument to the `+` interpreted as string

- concatenation will be promoted to string.

10.3 7.3 Finding things in strings

- indexOf() searches from the left
- lastIndexOf() searches from the right
- both methods can take a start index for searching.
- Self-study exercises 7.8 and 7.9 are good to test understanding.

10.4 7.4 Keyword search

- Thinking about test cases. What other tests could we add?
- Could we use a do-while loop For the code in Figure 7.8 and how would

we write it?

10.5 7.5 StringBuffer

Strings in Java are immutable, cannot be changed. When you need a string to change, e.g. replacing characters, a StringBuffer can (should) be used.

When we mutate a string using `stringVar = stringVar + newPart;` new instances are created automatically. The old instance will have to be garbage collected (memory automatically reclaimed by the Java Virtual Machine).

StringBuffer class can change characters of a string *in place*. The class has `append()`, `insert()` and `setChar()` methods to achieve this. To get a *regular* string back the `toString()` method can be called.

10.6 7.6 Extracting parts of strings

- `charAt()`
- `substring(int)`: Returns the sub-string from start index given up to end of the string.
- `substring(int, int)`: Returns the sub-string from start to end index given, with **character at end index not included**.

10.7 7.8 StringTokenizer

Splitting strings into parts. We can access words, sentences or values in a comma-separated line. If split on anything else than space, using `trim()` on the elements might be useful.

10.8 7.9 Comparing strings

Testing for identity (same object in memory) vs equality (same representation).

`s1==s2` is the same as `obj1.equals(obj2)`, tests for identity, i.e. do variables point to the same object in memory?

`s1.equals(s2)` From javadoc: The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

`s1.equalsIgnoreCase(s2)` same as above, but ignoring case.

Selfstudy exercise 7.16 is reviewing these concepts

11 Chapter 8

11.1 Goal

Concepts:

- Inheritance: a subclass has access to instance variables and methods of the super class. We can use the keyword `super` to access specifically parent elements. We can use `super()` to call parents constructor.
- Overriding: methods of the superclass can be redefined in the subclass.
- Dynamic (late/run-time) binding leads to polymorphims (poly=many, morph=form).
- There are 3 kinds of polymorphism:
  1. Overriding an inherited method
  2. Implementing an abstract method
  3. Implementing a Java interface
- A class is abstract as soon as one of its method is abstract, i.e. does not provide a method body.

- Interfaces are similar to abstract classes: They only contain abstract methods and constant (final) variables. Interfaces often describe a *role*.

Examples:

1. Inherit and override: Revisit overriding `equals()`, use `Student` class as a basis. Start from:

```
public class Student {
   protected String name;
   public Student ( String s ) {
      name = s;
   }
   public String getName() {
      return name;
   }
   public String toString () {
      return "My␣name␣is␣" + name + "␣and␣I␣am␣a␣Student.";
   }
}
```

add call to parent `toString()`, and add `equals()`. Next, create subclass college student, show how to call parent constructor.

How can we demonstrate late binding?

1. Abstract class: Discuss animal example. Start with:

```
public abstract class Animal {
   protected String kind;//Cow, pig, cat, etc.

   public Animal ( ) { }

   public String toString () {
      return "I␣am␣a␣" + kind + "␣and␣I␣go␣" + speak();
   }

   public abstract String speak(); // Abstract method
}
```

Add an animal and implement `speak()`. Note that abstract classes cannot be instantiated. `protected` makes variable visible in subclasses.

1. Interfaces

We have used interfaces for GUIs, the `ActionListener`.

Start with new definition of animal and a new interface:

```
public interface Speakable {
   public String speak ( ) ;
}

public class Animal {
   protected String kind;//Cow, pig, cat, etc. public Animal ( ) { }
   public String toString () {
      return "I␣am␣a␣" + kind + "␣and␣I␣go␣" + ((Speakable)this ).speak();
   }
}
```

Change the animal subclass created above to implement `Speakable`.

Note that interface methods are like abstract methods. Interfaces cannot be instantiated, but can be cast to.

It would be good to do a separate lecture on Section 8.5 + 8.6.

12 Chapter 9

12.1 Goals - arrays

- 1D and 2D arrays
- pure and modifier functions
- Using String.split()
- Vector and ArrayList
- Count character frequency from text in file.

12.1.1 Pure vs modifier methods

```
class Main {

  public static void printArray(double[] d){
    for(int i=0; i< d.length;i++){
      System.out.print(d[i]+"␣");
    }
    System.out.print("\n");
  }

  //Changes array d
  public static void setAt(double[] d, int idx, double val){
    d[idx]=val;
```

```
  }

  //Pure method: Does NOT change array d
  public static double[] setAtPure(double[] d, int idx, double val){
    double[] r = d.clone();
    r[idx]=val;
    return r;
  }

  public static void main(String[] args) {
    double[] d = {1.0, 3.5, 5.5};
    double[] d2;

    printArray(d);
    setAt(d, 2, -1.0);
    printArray(d);
    d2 = setAtPure(d, 2, 100.0);
    printArray(d);
    printArray(d2);
  }
}
```

12.1.2 String split rather than StringTokenizer

According to JavaDoc:
`https://docs.oracle.com/javase/8/docs/api/java/util/StringTokenizer.html`

StringTokenizer is discouraged, and String.split() should be used. This method returns a String array. It would be great to revisit this here.

```
class Main {
  public static void main(String[] args) {
    String s = "Nice, I like it.\nThis is a great example! What do you think?";

    /*
    split() takes a regular expression. Try:
    "\\s" any whitespace
    "[.!?]" a group defining sentence endings
    "," commas
    More regex:
    https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum
    */
    String[] parts = s.split(",");
    for(int i=0; i< parts.length; i++){
      System.out.println(""+i+": "+parts[i]);
    }
```

```
  }
}
```

12.1.3 2D arrays

```
class Main {
   public static void main(String[] args) {
      double[] b = {1.0, 2.0, 3.0, 4.0, 5.0};
      double[][] a = new double[10][5];

      a[0][1] = 10.0; //assigning a single element
      a[1] = b; //assign a 'row'

      for(int row=0; row<a.length; row++){
         for(int col=0; col<a[row].length; col++){
            System.out.print(a[row][col]+"␣");
         }
         System.out.print("\n");
      }
   }
}
```

How do we print or calculate mean of a single row, or single column?

12.1.4 Vector and ArrayList

Both Vector and ArrayList provide dynamic arrays. Vector is thread-safe, ArrayList is not. Both use standard arrays to store data. To be dynamic, arrays need to grow internally. This is done with a different strategy for Vector and ArrayList. I see more ArrayList now.

Both implement the `List` interface and use generic types. When creating a new Vector or ArrayList, provide the type in angle brackets `< >`. Types need to be Objects, not primitive types. Use `Integer` instead of `int`.

There is `LinkedList` too that implements the `List` interface. Internal storage is different. All three can be used interchangeably.

It is possible to iterate a list with a traditional for loop. We would need `size()` and `get()` methods. Additionally, there is another for-loop structure available to iterate lists.

```
class Main {
  public static void main(String[] args) {
    Vector<Double> d = new Vector<Double>();
    // ArrayList<Double> d = new ArrayList<Double>();
    // List<Double> d = new ArrayList<Double>();
```

```
    // List<Double> d = new LinkedList<Double>();

    d.add(1.0);
    d.add(3.5);
    d.add(5.5);

    System.out.println(d);
    d.set(2, -1.0); //d[2] = -1.0;
    System.out.println(d);
    d.add(-2.0);
    System.out.println(d);
    double x = d.get(2); //x = d[2];
    System.out.println(x);
    d.remove(2);
    System.out.println(d);

    for(Double y: d){
      System.out.println(y);
    }
  }
}
```

12.2 Goals - search and sort

- sort:
  - selection (self study ex 9.9)
  - insertion (self study ex 9.8)
- search:
  - sequential search ($O(N)$)
  - binary search ($O(\log(N))$, self study ex 9.13
- comparable interface, implement `int compareTo(Object o)`, returns:
  - -1 if this is *smaller* than that
  - +1 if this is *larger* than that
  - 0 if this is *equal* to that
  - Self-study ex 9.20, add comparable to LetterFreq

12.3 9.2 One-Dimensional Arrays

Figure 9.2 is a good reference: creating the array, does not also create the objects.

Arrays can be initialized directly with a list of elements in curly braces: `String[] arr = {"Hello", "Calgary", "Great"};`

12.4 9.4 Counting character frequencies

We should do Self-study Exercise 9.7: Open a file and then do the character frequency count. Maybe add some stats?

12.5 9.5 Sorting

Insertion sort, see also `https://en.wikipedia.org/wiki/Insertion_sort` The array is divided into a *sorted* (left) and an *unsorted* (right) portion. Starting with the first element as the *sorted* portion, at every iteration the next element in the *unsorted* side is inserted in the proper location on the *sorted* side by shifting larger elements to the right. See Fig 9.13 (p. 409) for code.

Selection sort, see also `https://en.wikipedia.org/wiki/Selection_sort` All elements are traversed left to right. At each iteration, the smallest element in the remaining array is searched and if smaller than current, it is swapped.

We could implement selection sort from pseudo code on page 411. See self-study exercise 9.11.

The topic of passing by reference is discussed again. It is used here to change the array elements within the sorting method. This works, because the array parameter gets passed as a reference.

Self-study exercise 9.8 and 9.9.

12.6 9.6 Searching

Sequential search: go through all elements until you find the *key*. Linear time, at most N comparisons. Sequential search works without first sorting or converting data into a searchable data structure such as a tree. It works well for small, un-sorted arrays. `https://en.wikipedia.org/wiki/Linear_search`

If elements are sorted, you can use other search algorithms, e.g. binary search. Logarithmic time, at most $log_2(N)$ comparisons. Starting with a *key* and the full range of the array `min=0`, `max=N-1`, where `N` is the number of

elements in the array, the mid-point of the range is computed and the. If the *key* is not at the mid-point, the search is continued in either the left or the right range depending on how *key* compares to the mid-point. Since we can divide the range into two sub-ranges only $log_2(N)$ times, as $2^{log_2(N)} = N$, the *key* can be found in $log_2(N)$ steps. For 100 items, we would need at most 7 guesses ($2^7 = 128 > 100$). For 1000 items, we need only three more, 10 guesses ($2^{10} = 1024 > 1000$).

`https://en.wikipedia.org/wiki/Binary_search_algorithm`

Self-study exercise 9.13.

12.7 9.7 2D arrays

A 2D array is an array of arrays:

```
class Main {
   public static void main(String[] args) {
      double[] b = {1.0, 2.0, 3.0, 4.0, 5.0};
      double[][] a = new double[10][5];

      a[0][1] = 10.0; //assigning a single element
      a[1] = b; //assign a 'row'

      for(int row=0; row<a.length; row++){
         for(int col=0; col<a[row].length; col++){
            System.out.print(a[row][col]+"␣");
         }
         System.out.print("\n");
      }
   }
}
```

see Fig. 9.22.

How would we print a single row? How would we print a single column? Self-study exercises 9.17 - 9.19 are good.

12.8 9.9 polymorphic sort

To sort objects, classes can implement `Comparable` with one method `int compareTo(Object o)` that: Compares this object with the specified object for order.

```
Returns a
negative integer,
zero,
```

```
or a positive integer
as this object is
less than,
equal to,
or greater than
the specified object.

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
```

```
public void sort(Comparable[] arr) {
   Comparable temp; // Temporary variable for insertion
   for(intk=1;k<arr.length;k++) {
      temp = arr[k]; // Remove it from array
      int i;
      for (i = k-1; i >= 0 && arr[i].compareTo(temp) > 0; i--)
         arr[i+1] = arr[i]; // Move it right by one
      arr[i+1] = temp; // Insert the element
   }
}// sort()
```

Self-study exercise 9.20 asks to add `compareTo()` to `LetterFreq` class.

Note that `java.lang.Array.sort()` implements sorting and should probably be used in practice.

12.9 9.10 Vector

Regular arrays cannot grow or shrink in size. A Vector class is a Collection, another data type, that can hold objects and adapt its size.

Vector class definition uses generic types. Upon declaration, the type is specified in angle brackets. `Vector<String> stringVector = new Vector<String>();`

An alternative to Vector is ArrayList.

The difference seems to be in thread safety and how the array grows when needed. See `https://stackoverflow.com/questions/2986296/what-are-the-differences-between`

ArrayList are defined with generic types as well. Upon declaration the specifc type is defined in angle brackets: `ArrayList<String> stringVector = new ArrayList <String>();`

13 Chapter 10

13.1 Goals

- introduce try-catch-finally

- Organize catch from most specific to most general.
- checked vs unchecked exceptions
- what to do: fix, stop, or log.
- throw an exception (Fig. 10.7)

From the introduction:

> "An exception is an erroneous or anomalous condition that arises while a program is running. ... In Java, the preferred way of handling such conditions is to use exception handling, ...".

Which exception is raised (thrown) when executing `Integer.parseInt("26.2")`?

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "26.2"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.j
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.main(Main.java:10)
exit status 1
```

A `NumberFormatExcpetion`, a subclass of `Exception`. The Java virtual machine handles *unchecked* exceptions automatically by stopping the program and printing the method trace. *Unchecked* exceptions are `RuntimeException` and its subclasses.

What to do when an exception occurs?

- Fix the condition and continue.
- Report the condition and stop the program.
- Log the condition and continue (program cannot/should not be stopped).

Can we fix the `umberFormatException`? It depends. If this is part of reading values from a file, we could assign *not-a-number* and continue reading other data. We delay the problem. If this is interactive user input, we would prompt the user again.

try-catch-finally allows us to handle exceptions:

```
class Main {
  public static void main(String[] args) {
    String input = "26.2";
    int num = 0;
```

```
    try{
      num = Integer.parseInt(input);
    }catch(NumberFormatException e){
      System.out.println(e.getMessage());
      e.printStackTrace();
    }finally{
      System.out.println("Your␣number␣is:␣"+num);
    }
  }
}
```

We can handle multiple exceptions, organize them according to hierarchy: most specific to most general. Let's pick another one from Table 10.2. Note that catch is like a method header where the exception is defined like an parameter to a method.

Some exception are *checked* exceptions. Excpetions not `RunetimeExcpetion` or sub-classes are *checked* exceptions. The compiler complains if we do not handle them. In addition to try-catch, there is another option, use `throws` in the method header and let the caller handle it. Re-throwing needs to go all the way up the call chain up to `main()` unless we have a try-catch somewhere in the dynamic scope.

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

class Main {
  public static void main(String[] args){
  //public static void main(String[] args) throws FileNotFoundException{

    //throws FileNotFoundException
    //Scanner s = new Scanner(new File("test.txt"));

    try{
      Scanner s = new Scanner(new File("test.txt"));
    }catch(FileNotFoundException e){
      System.out.println(e.getMessage());
    }

  }
}
```

The scanner constructor throws a `FileNotFoundException`: `https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#Scanner-java.io.File-`

We can either declare that `main()` throws this exception or add a try-catch to make the compiler happy.

We can throw an exception our self:

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

class Main {

  public static int addPositive(int a, int b) throws IllegalArgumentException{
    if(a<=0 || b<=0){
      throw new IllegalArgumentException("ERROR: One or both of the arguments are zero or negat
    }
    return a+b;
  }

  public static void main(String[] args){
    try{
      System.out.println(addPositive(5, -2));
    }catch(IllegalArgumentException e){
      System.out.println(e);
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }

  }
}
```

13.2 10.2 Handling exceptions

Example is division by zero - it can not be computed. We can explicitly test for the denominator and end the program when it is zero to avoid any other problems.

Java has this mechanism built in. If a division by zero occurs, the Java virtual machine detects and terminates the program with an error description and a trace.

13.3 10.3 Exception hierarchy

Class hierarchy with `Exception` at the top. Exceptions become more specific.

Table 10.1 and 10.2 lists frequent exceptions.

Checked exceptions like `IOException` are checked by the compiler. The program must either handle (try-catch) or declare the exception (`throws`).

Unchecked exceptions are `RuntimeExcpetion` or subclasses (Fig. 10.4), which are not checked by the compiler.

13.4 10.4 Handling exceptions

Fig 10.7 is illustrating a method throwing an exception if this would result in division by zero. The main method catches the exception with try-catch and exits.

Unless your try-catch is significantly different from Java's default you may not need to include it (for unchecked exceptions).

The try block contains the normal algorithm, the catch block the code that handles the exception (error).

The try-catch-finally statement. If an exception is thrown in the try block, the block is left and not returned. The corresponding catch block matching the type of exception thrown is executed. Either way, an optional finally block may be executed.

Multiple catch blocks, each with a different exception type are possible and should be arranged from most specific to most general. It is the first block that matches that is executed, `Exception e` would match all exceptions and should be last. See top of page 472.

Dynamic vs static scoping: Dynamic scopes can span multiple code blocks they are defined by how the program is executed. Static scopes are defined by how the program is written. See Fig. 10.10.

See Fig 10.12 for the method call stack and propagation of exception. After doing static scoping (the local method), Java uses the method stack to do dynamic scoping and find the matching catch block for an exception.

Self-study exercises are useful to deepen understanding.

13.5 10.5 Error handling

What to do? Three types as summarized in table 10.3: Fixable, not fixable - program stoppable, not fixable - program not stoppable.

Defensive design, anticipate user errors and handle them. Example is retry entering a number.

A nice example of growing an internal array. However, it is more expensive (and bad) to rely on exceptions to implement this than using for example `Vector` which does the growing internally.