

CS 200 - Programming I: Programming Process

Marc Renault

Department of Computer Sciences
University of Wisconsin – Madison

Fall 2019

TopHat Sec 3 (1:20 PM) Join Code: 682357

TopHat Sec 4 (3:30 PM) Join Code: 296444



WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

COMPUTATIONAL THINKING

PROBLEM SOLVING

ART OF PROGRAMMING

Abstraction

- Can you see the essential parts of the problem?
- What are the outputs? inputs? their relationship?
- Can you summarize the problem at a high level?
- What are the different components of the solution?

PROBLEM SOLVING

ART OF PROGRAMMING

Abstraction

- Can you see the essential parts of the problem?
- What are the outputs? inputs? their relationship?
- Can you summarize the problem at a high level?
- What are the different components of the solution?

Computational Thinking

- Think like a computer.
- What is the sequence of actions need to accomplish the task?
- Start with *pseudocode*.

THINKING LIKE A COMPUTER

Programming is like Designing a Recipe

Chocolate Chip Cookies

Ingredients:

- 227g (1 cup) butter, softened
- 200g (1 cup) sugar
- 105g (½ cup) brown sugar
- 2 eggs
- 2 tsp vanilla
- 250g (2 cups) all-purpose flour
- 1 tsp soda
- 1 pinch salt
- 1 ½ cups of chocolate chips

Instructions:

1. Beat butter, sugars, eggs and vanilla until light and fluffy.
2. Add flour, soda, and salt; blend well.
3. Add chips.
4. Drop from a teaspoon 2 inches apart.
5. Bake 190°C for 9 min.

Original: 375°F for 10 min.

Convection: 325°F for 9 – 10 min.

THINKING LIKE A COMPUTER

Programming is like Designing a Recipe

Chocolate Chip Cookies

Ingredients:

- 227g (1 cup) butter, softened
- 200g (1 cup) sugar
- 105g (½ cup) brown sugar
- 2 eggs
- 2 tsp vanilla
- 250g (2 cups) all-purpose flour
- 1 tsp soda
- 1 pinch salt
- 1 ½ cups of chocolate chips

Instructions:

1. Beat butter, sugars, eggs and vanilla until light and fluffy.
2. Add flour, soda, and salt; blend well.
3. Add chips.
4. Drop from a teaspoon 2 inches apart.
5. Bake 190°C for 9 min.

Original: 375°F for 10 min.

Convection: 325°F for 9 – 10 min.

Human Thinking



THINKING LIKE A COMPUTER

Programming is like Designing a Recipe

Chocolate Chip Cookies

Ingredients:

- 227g (1 cup) butter, softened
- 200g (1 cup) sugar
- 105g (½ cup) brown sugar
- 2 eggs
- 2 tsp vanilla
- 250g (2 cups) all-purpose flour
- 1 tsp soda
- 1 pinch salt
- 1 ½ cups of chocolate chips

Instructions:

1. Beat butter, sugars, eggs and vanilla until light and fluffy.
2. Add flour, soda, and salt; blend well.
3. Add chips.
4. Drop from a teaspoon 2 inches apart.
5. Bake 190°C for 9 min.

Original: 375°F for 10 min.

Convection: 325°F for 9 – 10 min.

Computer Thinking



THINKING LIKE A COMPUTER

Programming is like Designing a Recipe

Chocolate Chip Cookies

Ingredients:

- 227g (1 cup) butter, softened
- 200g (1 cup) sugar
- 105g (½ cup) brown sugar
- 2 eggs, beaten
- 2 tsp vanilla
- 250g (2 cups) all-purpose flour
- 1 tsp soda
- 1 pinch salt
- 1 ½ cups of chocolate chips

Instructions:

1. Beat butter, sugars, eggs and vanilla until light and fluffy.
2. Add flour, soda, and salt; blend well.
3. Add chips.
4. Drop from a teaspoon 2 inches apart.
5. Bake 190°C for 9 min.

Original: 375°F for 10 min.

Convection: 325°F for 9 – 10 min.

Computer Thinking



THINKING LIKE A COMPUTER

Programming is like Designing a Recipe

Chocolate Chip Cookies

Ingredients:

- 227g (1 cup) butter, softened
- 200g (1 cup) sugar
- 105g (½ cup) brown sugar
- 2 eggs, beaten
- 2 tsp vanilla
- 250g (2 cups) all-purpose flour
- 1 tsp soda
- 1 pinch salt
- 1 ½ cups of chocolate chips

Instructions:

1. Beat butter, sugars, eggs and vanilla until light and fluffy.
2. Add flour, soda, and salt; blend well.
3. Add chips.
4. Drop from a teaspoon 2 inches apart.
5. Bake 190°C for 9 min.

Original: 375°F for 10 min.

Convection: 325°F for 9 – 10 min.

Computer Thinking



EXERCISE

CALCULATING THE AREA AND CIRCUMFERENCE OF A CIRCLE.

- 1 What are the inputs? outputs? their relationship?
- 2 Pseudocode it!
- 3 Code it!

EXPLAINING AND TRACING

EXPLAINING VS TRACING

Explaining

Summarize and provide a high-level explanation of what the code does in plain English.

TOPHAT Q1

Explain the `Circle.java` code:

- a. Calculates the area and the circumference of a circle.
- b. Reads a radius as input from the user and outputs the the area and the circumference of the corresponding circle.
- c. Creates a double variable called `rad`. Initializes it as 0. Prompts the user for a radius and stores it in `rad`. Outputs "Area:"; calculates the area of a circle of radius `rad` and outputs it. Outputs "Circumference:"; calculates the circumference of a circle of radius `rad` and outputs it.

EXPLAINING VS TRACING

Explaining

Summarize and provide a high-level explanation of what the code does in plain English.

Tracing

Run the code as computer does.

- Put pen to paper.
- Write down the active variables and their values.
- Update them as they change as you mentally walk through the statements sequentially.

TOOL FOR TRACING

Java Visualizer

- https://cscircles.cemc.uwaterloo.ca/java_visualize/



The screenshot shows a web browser window titled "Java Visualizer - Google Chrome". The address bar displays the URL https://cscircles.cemc.uwaterloo.ca/java_visualize/. The page content includes:

- A logo for "Java Visualizer (beta: report a bug)" featuring a blue and yellow glasses icon.
- A text prompt: "Write your Java code here:"
- A code editor containing a simple Java class structure:

```
1 public class ClassNameHere {
2     public static void main(String[] args) {
3
4     }
5 }
```
- Input fields for "options", "args: +command-line argument", and "stdin: (also visualizes consumption of stdin)".
- A "Visualize Execution" button.
- A list of links for various examples: [basic examples](#) | [Default](#) | [Variables](#) | [CmdLineArgs](#) | [StdIn](#) | [ControlFlow](#) | [Sort](#) | [ExecLimit](#) | [Strings](#) | [method examples](#) | [PassByValue](#) | [Recursion](#) | [StackOverflow](#) | [oop examples](#) | [Roles](#) | [Person](#) | [Cometes](#) | [Casting](#) | [data structure examples](#) | [LinkedList](#) | [StackQueue](#) | [Postfix](#) | [SymbolTable](#) | [java feature examples](#) | [ToString](#) | [Reflect](#) | [Exception](#) | [ExceptionFlow](#) | [TwoClasses](#) | [misc examples](#) | [Event](#) | [Knapsack](#) | [LambdaExample](#) | [StaticInitializer](#) | [Synthetic](#)
- A note: "The visualizer supports `StdIn`, `StdOut`, most other `stdlib` libraries, `Stack`, `Queue`, and `ST`. Click for FAQ."
- A "Generate URL" button and a text input field.
- Footer text: "To share this visualization, click the 'Generate URL' button above and share that URL. To report a bug, paste the URL along with a brief error description in an email addressed to daveagn@gmail.com. Based on [Online Python Tutor](#). © 2010-2013 [Philip Guo](#) all rights reserved. Java version by [David Pritchard](#), [Will Gwozdz](#). Source code: for this version's [backend](#), the [frontend](#) and [installation instructions](#).

EDIT-COMPILE-RUN CYCLE

EDIT

Writing some source code (set of instructions) in plain text.

Editors

- Text Editors:



vi



EDIT

Writing some source code (set of instructions) in plain text.

Editors

- Text Editors:



vi



- Integrated Development Environment (IDE):



COMPILE

Build your code so that it can be run on a computer.

Often compilers build an *executable* that can be run *natively* on the *target platform*.

COMPILE

Build your code so that it can be run on a computer.
Often compilers build an *executable* that can be run *natively* on the *target platform*.

javac – The *Java* compiler

- javac produces a *bytecode* file that needs to be run in a *virtual machine*.

COMPILE

Build your code so that it can be run on a computer.
Often compilers build an *executable* that can be run *natively* on the *target platform*.

javac – The *Java* compiler

- javac produces a *bytecode* file that needs to be run in a *virtual machine*.
- *javac* produces a bytecode file with a `.class` file extension.

COMPILE

Build your code so that it can be run on a computer.
Often compilers build an *executable* that can be run *natively* on the *target platform*.

javac – The *Java* compiler

- javac produces a *bytecode* file that needs to be run in a *virtual machine*.
- *javac* produces a bytecode file with a `.class` file extension.
- Example:

```
javac Circle.java
```

produces a `Circle.class` *bytecode* file.

RUN

Run or execute your code.

java

- The `.class` bytecode file produced by `javac` needs to be run in the Java virtual machine.

RUN

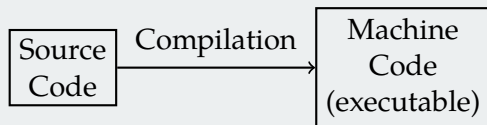
Run or execute your code.

java

- The `.class` bytecode file produced by `javac` needs to be run in the Java virtual machine.
- Example:
 `java Circle`
 runs the `Circle` file.

VIRTUAL MACHINE

The Classic Model: Native executable



The Java Model: Virtual Machine



TOPHAT Q2

I have just written some Java code in a file `SomeCode.java`. How do I compile and run it?

- a. `java SomeCode`
- b. `javac SomeCode.class`
- c. `javac SomeCode.java`
- d. `java SomeCode.java`
- e. `javac SomeCode`
- f. `java SomeCode.class`

JAVA INITIATION

BASIC OUTPUT

String Literals

- Double quotes ("):
"I am a string literal"

BASIC OUTPUT

String Literals

- Double quotes ("):
"I am a string literal"
- Concatenation (+):
`int a = 10; System.out.println("A " + a);`

BASIC OUTPUT

String Literals

- Double quotes ("):
"I am a string literal"
- Concatenation (+):
`int a = 10; System.out.println("A " + a);`
- Newline escape character(\n):
"First line\nSecond line"

BASIC OUTPUT

String Literals

- Double quotes ("):
"I am a string literal"
- Concatenation (+):
`int a = 10; System.out.println("A " + a);`
- Newline escape character(\n):
"First line\nSecond line"

Printing to the Console

- Print a string:
`System.out.print("Does not append a new line");`
- Print a string with a newline:
`System.out.println("Appends a new line");`

TOPHAT Q3

What is the output of:
`System.out.println("5 and 5 = " + 5 + 5);`

Type the output.

BASIC INPUT

Using the Scanner

- Include the library at the top of the file:
`import java.util.Scanner;`
- Create an *instance* of a Scanner *object*:
`Scanner sc = new Scanner(System.in);`
- Reading an integer:
`int anInt = sc.nextInt();`

COMMENTS AND WHITESPACE

Comments

- Ignored by the compiler.
- Written by the programmer to explain the code.

- Single-line (//)

```
// Single line comment
```

- Multi-line (/* */)

```
/**
```

```
 * Multi-line comment
```

```
*/
```

COMMENTS AND WHITESPACE

Comments

- Ignored by the compiler.
- Written by the programmer to explain the code.

- Single-line (//)

```
// Single line comment
```

- Multi-line (/* */)

```
/**
```

```
 * Multi-line comment
```

```
*/
```

Whitespace

- Mostly ignored by the compiler.
- Good use of white space makes code easier read!

COMPILER ERRORS AND WARNINGS

Compilation Errors

- Prevents compilation
- Syntax errors

COMPILER ERRORS AND WARNINGS

Compilation Errors

- Prevents compilation
- Syntax errors
- Examples:
 - missing ;
 - typo (variable name, keyword)
 - missing braces

COMPILER ERRORS AND WARNINGS

Compilation Errors

- Prevents compilation
- Syntax errors
- Examples:
 - missing ;
 - typo (variable name, keyword)
 - missing braces

Warnings

- Warnings don't stop the compilation process.

COMPILER ERRORS AND WARNINGS

Compilation Errors

- Prevents compilation
- Syntax errors
- Examples:
 - missing ;
 - typo (variable name, keyword)
 - missing braces

Warnings

- Warnings don't stop the compilation process.
- Good practice to write programs that compile without warnings.

COMPILER ERRORS AND WARNINGS

Compilation Errors

- Prevents compilation
- Syntax errors
- Examples:
 - missing ;
 - typo (variable name, keyword)
 - missing braces

Warnings

- Warnings don't stop the compilation process.
- Good practice to write programs that compile without warnings.
- For even stricter compilation, use `-Xlint`:
`javac -Xlint Foo.java`

COMPUTER

BASIC MACHINE ARCHITECTURE



“This is a computer.”

von Neumann Architecture

- von Neumann proposed this architecture in 1945.
- Consists of:
 - a processing unit,
 - memory,
 - input devices, and
 - output devices.

BASIC MACHINE ARCHITECTURE

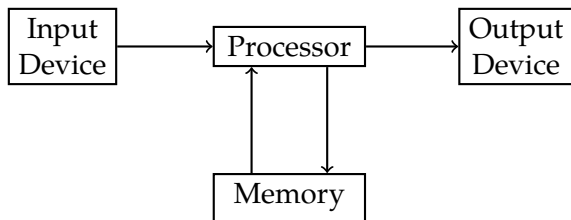


John von Neumann

von Neumann Architecture

- von Neumann proposed this architecture in 1945.
- Consists of:
 - a processing unit,
 - memory,
 - input devices, and
 - output devices.

BASIC VIEW OF A COMPUTER



FURTHER READING



COMP SCI 200: Programming I
zyBooks.com, 2015.
zyBook code:
WISCCOMPSCI200Fall2019

- Chapter 1. Programming Process

APPENDIX

REFERENCES

IMAGE SOURCES I



<http://www.eclipse.org/>



[https:](https://www.gnu.org/software/emacs/emacs.html)

[//www.gnu.org/software/emacs/emacs.html](https://www.gnu.org/software/emacs/emacs.html)



<http://packerville.blogspot.ca/2010/05/gentlemen-this-is-football.html>



[https:](https://en.wikipedia.org/wiki/Microsoft_Notepad)

[//en.wikipedia.org/wiki/Microsoft_Notepad](https://en.wikipedia.org/wiki/Microsoft_Notepad)



<http://2dvocabularynetwokr78.blogspot.fr/>

IMAGE SOURCES II



<http://www.lanl.gov/history/atomicbomb/images/NeumannL.GIF>



WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

<https://brand.wisc.edu/web/logos/>



<https://commons.wikimedia.org/w/index.php?curid=1228427>



Visual Studio

<https://commons.wikimedia.org/w/index.php?curid=57649239>



<http://www.zybooks.com/>