



# Introduction to Data Structures and Algorithms

COMP 251

# Preliminary Quiz

STATE YOUR NAME AND UFVID and answer the following questions, please:

1. Define *Algorithm*.
2. Define Dijkstra's *Control Structures*.
3. Define *Von Neuman Architecture*.
4. Describe the *Instruction Cycle*.
5. Define the main characteristics of OOP.
6. Define *Recursion*.
7. Write the Java Code for "Is there anybody out there?"
8. What does " $a \bmod b$ " mean in words?

# Goals of this Lecture

- Briefly summarize the introductory material (Section 1: Chapters 1.- 4 in several PPTs).
- Introduce the following fundamental notions:
  - The *need for data structures* within the context of computing,
  - The *programming paradigms* and to compare and contrast the various approaches to computing within these paradigms,
  - The core ideas behind **data structures** (which are nothing more than [presumably efficient] *methods of information storage, manipulation and retrieval*), and
  - The core ideas behind **algorithms** (*which are nothing more than methods of addressing solvable problems where data structures are an integral part of algorithmic efficiency*).

# Goals of this Lecture

- **NOTE:** This is an **intermediate-level course** with emphasis on OO problem solving in **Java**.
- My approach is to *briefly introduce data structures from the procedural viewpoint (i.e., a language like C), which is less abstract and, therefore, more amenable for developing the core concepts of data structures.*
- Then, we increase the level of *abstraction* using OO techniques within the context of Java, which will form the main body of this course.

# Preamble

# Section One Summary and Preamble



## Data Structures & Problem Solving Using Java

fourth edition

## contents

### part one **Tour of Java**

|                  |   |          |
|------------------|---|----------|
| <b>chapter 1</b> | <b>primitive java</b>                             | <b>3</b> |
| 1.1              | <b>the general environment</b>                    | 4        |
| 1.2              | <b>the first program</b>                          | 5        |
| 1.2.1            | comments  | 5        |
| 1.2.2            | main  | 6        |
| 1.2.3            | terminal output                                   | 6        |
| 1.3              | <b>primitive types</b>                            | 6        |
| 1.3.1            | the primitive types                               | 6        |
| 1.3.2            | constants   | 7        |
| 1.3.3            | declaration and initialization of primitive types | 7        |
| 1.3.4            | terminal input and output                         | 8        |
| 1.4              | <b>basic operators</b>                            | 8        |
| 1.4.1            | assignment operators                              | 9        |
| 1.4.2            | binary arithmetic operators                       | 10       |
| 1.4.3            | unary operators                                   | 10       |
| 1.4.4            | type conversions                                  | 10       |
| 1.5              | <b>conditional statements</b>                     | 11       |
| 1.5.1            | relational and equality operators                 | 11       |
| 1.5.2            | logical operators                                 | 12       |
| 1.5.3            | the if statement                                  | 13       |
| 1.5.4            | the while statement                               | 14       |
| 1.5.5            | the for statement                                 | 14       |
| 1.5.6            | the do statement                                  | 15       |
| 1.5.7            | break and continue                                | 16       |
| 1.5.8            | the switch statement                              | 17       |
| 1.5.9            | the conditional operator                          | 17       |
| 1.6              | <b>methods</b>                                    | 18       |
| 1.6.1            | overloading of method names                       | 19       |
| 1.6.2            | storage classes                                   | 20       |



# Section One Summary and Preamble



## Data Structures & Problem Solving Using Java

fourth edition

## contents

### part one **Tour of Java**

|                  |   |           |
|------------------|---|-----------|
| <b>chapter 1</b> | <b>primitive Java</b>                   | <b>3</b>  |
| <b>chapter 2</b> | <b>reference types</b>                  | <b>27</b> |
| 2.1              | <b>what is a reference?</b>             | 27        |
| 2.2              | <b>basics of objects and references</b> | 30        |
| 2.2.1            | the dot operator (.)                    | 30        |
| 2.2.2            | declaration of objects                  | 30        |
| 2.2.3            | garbage collection                      | 31        |
| 2.2.4            | the meaning of =                        | 32        |
| 2.2.5            | parameter passing                       | 33        |
| 2.2.6            | the meaning of ==                       | 33        |
| 2.2.7            | no operator overloading for objects     | 34        |
| 2.3              | <b>strings</b>                          | 35        |
| 2.3.1            | basics of string manipulation           | 35        |
| 2.3.2            | string concatenation                    | 35        |
| 2.3.3            | comparing strings                       | 36        |
| 2.3.4            | other String methods                    | 36        |
| 2.3.5            | converting other types to strings       | 37        |
| 2.4              | <b>arrays</b>                           | 37        |
| 2.4.1            | declaration, assignment, and methods    | 38        |
| 2.4.2            | dynamic array expansion                 | 40        |
| 2.4.3            | ArrayList                               | 42        |
| 2.4.4            | multidimensional arrays                 | 45        |
| 2.4.5            | command-line arguments                  | 45        |
| 2.4.6            | enhanced for loop                       | 46        |
| 2.5              | <b>exception handling</b>               | 47        |
| 2.5.1            | processing exceptions                   | 48        |
| 2.5.2            | the finally clause                      | 48        |
| 2.5.3            | common exceptions                       | 49        |
| 2.5.4            | the throw and throws clauses            | 51        |
| 2.6              | <b>input and output</b>                 | 51        |
| 2.6.1            | basic stream operations                 | 52        |
| 2.6.2            | the Scanner type                        | 53        |
| 2.6.3            | sequential files                        | 56        |

# Section One Summary and Preamble



## Data Structures & Problem Solving Using Java

fourth edition

### contents

#### part one **Tour of Java**

|           |  |    |
|-----------|--|----|
| chapter 1 | primitive java                           | 3  |
| chapter 2 | reference types                          | 27 |
| chapter 3 | objects and classes                      | 69 |
| 3.1       | what is object-oriented programming?     | 69 |
| 3.2       | a simple example                         | 71 |
| 3.3       | javadoc                                  | 73 |
| 3.4       | basic methods                            | 76 |
| 3.4.1     | constructors                             | 76 |
| 3.4.2     | mutators and accessors                   | 76 |
| 3.4.3     | output and toString                      | 78 |
| 3.4.4     | equals                                   | 78 |
| 3.4.5     | main                                     | 78 |
| 3.5       | example: using java.math.BigInteger      | 78 |
| 3.6       | additional constructs                    | 79 |
| 3.6.1     | the this reference                       | 81 |
| 3.6.2     | the this shorthand for constructors      | 82 |
| 3.6.3     | the instanceof operator                  | 82 |
| 3.6.4     | instance members versus static members   | 83 |
| 3.6.5     | static fields and methods                | 83 |
| 3.6.6     | static initializers                      | 86 |
| 3.7       | example: implementing a BigInteger class | 86 |
| 3.8       | packages                                 | 90 |
| 3.8.1     | the import directive                     | 91 |
| 3.8.2     | the package statement                    | 93 |
| 3.8.3     | the CLASSPATH environment variable       | 94 |
| 3.8.4     | package visibility rules                 | 95 |
| 3.9       | a design pattern: composite (pair)       | 95 |



# Section One Summary and Preamble



## Data Structures & Problem Solving Using Java

fourth edition

### contents

#### part one **Tour of Java**

|                  |  |            |
|------------------|--|------------|
| <b>chapter 1</b> | <b>primitive java</b>  | <b>3</b>   |
| <b>chapter 2</b> | <b>reference types</b>                                       | <b>27</b>  |
| <b>chapter 3</b> | <b>objects and classes</b>                                   | <b>69</b>  |
| <b>chapter 4</b> | <b>inheritance</b>   | <b>109</b> |
| <b>4.1</b>       | <b>what is inheritance?</b>                                  | <b>110</b> |
| 4.1.1            | creating new classes   | 110        |
| 4.1.2            | type compatibility   | 115        |
| 4.1.3            | dynamic dispatch and polymorphism                            | 116        |
| 4.1.4            | inheritance hierarchies                                      | 117        |
| 4.1.5            | visibility rules   | 117        |
| 4.1.6            | the constructor and super                                    | 118        |
| 4.1.7            | final methods and classes                                    | 119        |
| 4.1.8            | overriding a method  | 121        |
| 4.1.9            | type compatibility revisited                                 | 121        |
| 4.1.10           | compatibility of array types                                 | 124        |
| 4.1.11           | covariant return types                                       | 124        |
| <b>4.2</b>       | <b>designing hierarchies</b>                                 | <b>125</b> |
| 4.2.1            | abstract methods and classes                                 | 126        |
| 4.2.2            | designing for the future                                     | 130        |
| <b>4.3</b>       | <b>multiple inheritance</b>                                  | <b>131</b> |
| <b>4.4</b>       | <b>the interface</b>   | <b>134</b> |
| 4.4.1            | specifying an interface                                      | 134        |
| 4.4.2            | implementing an interface                                    | 135        |
| 4.4.3            | multiple interfaces  | 135        |
| 4.4.4            | interfaces are abstract classes                              | 136        |
| <b>4.5</b>       | <b>fundamental inheritance in java</b>                       | <b>136</b> |
| 4.5.1            | the Object class   | 136        |
| 4.5.2            | the hierarchy of exceptions                                  | 137        |
| 4.5.3            | i/o: the decorator pattern                                   | 138        |
| <b>4.6</b>       | <b>implementing generic components using inheritance</b>     | <b>142</b> |
| 4.6.1            | using Object for genericity                                  | 142        |
| 4.6.2            | wrappers for primitive types                                 | 143        |
| 4.6.3            | autoboxing/unboxing  | 145        |
| 4.6.4            | adapters: changing an interface                              | 146        |
| 4.6.5            | using interface types for genericity                         | 147        |
| <b>4.7</b>       | <b>implementing generic components using java 5 generics</b> | <b>150</b> |
| 4.7.1            | simple generic classes and interfaces                        | 150        |
| 4.7.2            | wildcards with bounds  | 151        |
| 4.7.3            | generic static methods                                       | 152        |
| 4.7.4            | type bounds  | 153        |
| 4.7.5            | type erasure   | 154        |
| 4.7.6            | restrictions on generics                                     | 154        |
| <b>4.8</b>       | <b>the functor (function objects)</b>                        | <b>157</b> |
| 4.8.1            | nested classes   | 161        |
| 4.8.2            | local classes  | 161        |
| 4.8.3            | anonymous classes  | 163        |
| 4.8.4            | nested classes and generics                                  | 164        |
| <b>4.9</b>       | <b>dynamic dispatch details</b>                              | <b>164</b> |

# Section One Summary and Preamble



## Data Structures & Problem Solving Using Java

fourth edition

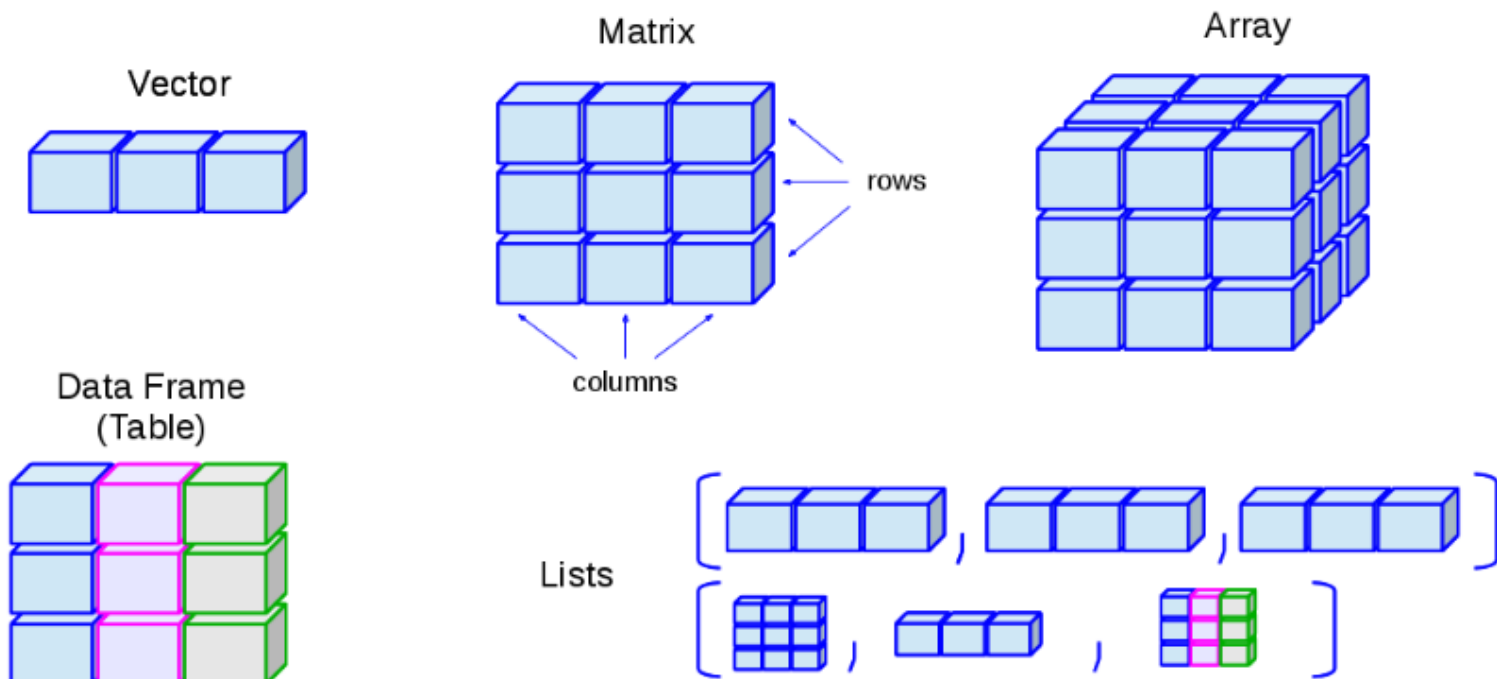
### contents

#### part one **Tour of Java**

|           |                     |     |
|-----------|---------------------|-----|
| chapter 1 | primitive java      | 3   |
| chapter 2 | reference types     | 27  |
| chapter 3 | objects and classes | 69  |
| chapter 4 | inheritance         | 109 |

|        |   |     |
|--------|---|-----|
| 4.1    | what is inheritance?                                  | 110 |
| 4.1.1  | creating new classes                                  | 110 |
| 4.1.2  | type compatibility                                    | 115 |
| 4.1.3  | dynamic dispatch and polymorphism                     | 116 |
| 4.1.4  | inheritance hierarchy                                 | 117 |
| 4.1.5  | visibility rules                                      | 117 |
| 4.1.6  | the constructor of a super class                      | 118 |
| 4.1.7  | final methods and classes                             | 119 |
| 4.1.8  | finalizing method                                     | 121 |
| 4.1.9  | type compatibility revisited                          | 121 |
| 4.1.10 | compatibility of arrays                               | 124 |
| 4.1.11 | covariant return type                                 | 124 |
| 4.2    | designing hierarchy                                   | 125 |
| 4.2.1  | abstract methods and classes                          | 126 |
| 4.2.2  | designing for the future                              | 130 |
| 4.3    | multiple inheritance                                  | 131 |
| 4.4    | the interface   | 134 |
| 4.4.1  | specifying an interface                               | 134 |
| 4.4.2  | implementing an interface                             | 135 |
| 4.4.3  | multiple interfaces                                   | 135 |
| 4.4.4  | interfaces are abstract classes                       | 136 |
| 4.5    | fundamentals of generics in java                      | 136 |
| 4.5.1  | the generic class                                     | 136 |
| 4.5.2  | hierarchy of exceptions                               | 137 |
| 4.5.3  | use of the decorator pattern                          | 138 |
| 4.6    | implementing generic components using inheritance     | 142 |
| 4.6.1  | using Object for genericity                           | 142 |
| 4.6.2  | wrappers for primitive types                          | 143 |
| 4.6.3  | autoboxing/unboxing                                   | 145 |
| 4.6.4  | adapters: changing an interface                       | 146 |
| 4.6.5  | using interface types for genericity                  | 147 |
| 4.7    | implementing generic components using java 5 generics | 150 |
| 4.7.1  | simple generic classes and interfaces                 | 150 |
| 4.7.2  | wildcards with bounds                                 | 151 |
| 4.7.3  | generic static methods                                | 152 |
| 4.7.4  | type bounds   | 153 |
| 4.7.5  | type erasure  | 154 |
| 4.7.6  | restrictions on generics                              | 154 |
| 4.8    | the functor (function objects)                        | 157 |
| 4.8.1  | nested classes  | 161 |
| 4.8.2  | local classes   | 161 |
| 4.8.3  | anonymous classes                                     | 163 |
| 4.8.4  | nested classes and generics                           | 164 |
| 4.9    | dynamic dispatch details                              | 164 |

# Beyond Java: Introductory Notions of Data Structures and Algorithms



# **DEFINITION:** Computing Science

# **DEFINITION:** Computing Science

The study of *algorithms*, including

# DEFINITION: Computing Science

The study of *algorithms*, including

- Their formal and mathematical properties,

# DEFINITION: Computing Science

The study of *algorithms*, including

- Their formal and mathematical properties,
- Their hardware realizations,



# DEFINITION: Computing Science

The study of *algorithms*, including

- Their formal and mathematical properties,
- Their hardware realizations,
- Their linguistic realizations,

# DEFINITION: Computing Science

The study of *algorithms*, including

- Their formal and mathematical properties,
- Their hardware realizations,
- Their linguistic realizations,
- Their applications.

Schneider and Gersting, 1999

# Definition: Algorithm

# Definition: Algorithm

A well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.

- Schneider and Gersting, 1999

# Exercise

Apply this definition to the quadratic equation and consider how to implement it with pseudocode.

# Exercise

Apply this definition to the quadratic equation and consider how to implement it with pseudocode.

- To be *effectively computable*, the discriminant must not be less than zero AND the leading coefficient must not be zero.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

# Exercise

Apply this definition to the quadratic equation and consider how to implement it with pseudocode.

- To be *effectively computable*, the discriminant must not be less than zero AND the leading coefficient must not be zero.
- Note that software development begins with defining the problem (requirements of SWE).

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$



# Exercise

So a generic BASIC-like pseudocode might be as below, where  $d$  is the discriminant.

```
1 let d = b * b - 4 * a * c
2 if d < 0 then
3   print "No real roots."
4 else if d = 0 then
5   print "Root is: "; -b / (2 * a); "."
6 else
7   print "Roots are: "; (-b + sqrt ( d )) / (2 * a); " and "; (-b - sqrt ( d )) /
8 end if
```

# Exercise

So a generic BASIC-like pseudocode might be as below, where  $d$  is the discriminant.

```
1 let d = b * b - 4 * a * c
2 if d < 0 then
3   print "No real roots."
4 else if d = 0 then
5   print "Root is: "; -b / (2 * a); "."
6 else
7   print "Roots are: "; (-b + sqrt ( d )) / (2 * a); " and "; (-b - sqrt ( d )) /
8   end if
```

Note that this is best done using Procedural Programming – OO is an unnecessary abstraction for most simple problems in mathematics.

# Areas of Computing Science

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)



# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)
- **Architecture (Design of Computers)**

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)
- Architecture (Design of Computers)
- Intelligent Systems (AI)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)
- Architecture (Design of Computers)
- Intelligent Systems (AI)
- Automata Theory (Abstract Computers)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)
- Architecture (Design of Computers)
- Intelligent Systems (AI)
- Automata Theory (Abstract Computers)
- Information Storage and Retrieval (DBMS)

# Areas of Computing Science

- Data Structures (Efficient Organization of Data)
- Operating Systems (Managing the Computer)
- Programming Languages (Software)
- Computational Complexity (Limitations of Computing)
- Architecture (Design of Computers)
- Intelligent Systems (AI)
- Automata Theory (Abstract Computers)
- Information Storage and Retrieval (DBMS)
- **Software Engineering (generally large systems designed with professional engineering practices)**

# Post-Preamble Summary

- We have defined fundamental terminology:
  - Computing Science,
  - Algorithms,
- We have also provided an overview of the various areas of research in Computing Science and *have seen where Data Structures fits into this taxonomy.*

# **The Fundamental Motivation for Data Structures**



# The Philosophy of Data Structures

- The primary goal of this lecture is to *motivate the need to study data structures*.
- This involves understanding that **unlimited computing power has logical barriers** and that, hence, there is a fundamental requirement to understand the nature of how data may be manipulated by any computer and, at the very least, for the sake of **efficient storage and retrieval of data**.
- **Computer programs are formal mathematical structures and, hence, these issues begin with the mathematical nature of algorithms.**

# Different Types of “Well-Known” Functions

Given a constant,  $n$ , and a variable,  $x$ , we can construct the following:

- $x^n$  – **Monomial**

# Different Types of “Well-Known” Functions

Given a constant,  $n$ , and a variable,  $x$ , we can construct the following:

- $x^n$  – **Monomial**
- $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  – **Polynomial**
  - $P_2(x) = a_2 x^2 + a_1 x + a_0$  – Quadratic
  - $P_1(x) = a_1 x + a_0$  – Linear
  - $P_0(x) = a_0$  – Constant

# Different Types of “Well-Known” Functions

Given a constant,  $n$ , and a variable,  $x$ , we can construct the following:

- $x^n$  – **Monomial**
  - $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  – **Polynomial**
    - $P_2(x) = a_2 x^2 + a_1 x + a_0$  – Quadratic
    - $P_1(x) = a_1 x + a_0$  – Linear
    - $P_0(x) = a_0$  – Constant
  - $n^x$  – **Exponential**
-

# Different Types of “Well-Known” Functions

Given a constant,  $n$ , and a variable,  $x$ , we can construct the following:

- $x^n$  – **Monomial**
- $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  – **Polynomial**
  - $P_2(x) = a_2 x^2 + a_1 x + a_0$  – Quadratic
  - $P_1(x) = a_1 x + a_0$  – Linear
  - $P_0(x) = a_0$  – Constant
- $n^x$  – Exponential
- $n!$  – Factorial

# Different Types of “Well-Known” Functions

Given a constant,  $n$ , and a variable,  $x$ , we can construct the following:

- $x^n$  – **Monomial**
- $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  – **Polynomial**
  - $P_2(x) = a_2 x^2 + a_1 x + a_0$  – Quadratic
  - $P_1(x) = a_1 x + a_0$  – Linear
  - $P_0(x) = a_0$  – Constant
- $n^x$  – Exponential
- $n!$  – Factorial
- And their inverse functions (such as  $x^{1/n}$  or  $\log x$ )

# Computational Time of Various Classes of Problems According to Input Size

**TABLE 2** The Computer Time Used by Algorithms.

| <i>Problem Size</i> | <i>Bit Operations Used</i> |              |                       |             |
|---------------------|----------------------------|--------------|-----------------------|-------------|
| $n$                 | $\log n$                   | $n$          | $n \log n$            | $n^2$       |
| 10                  | $3 \times 10^{-11}$ s      | $10^{-10}$ s | $3 \times 10^{-10}$ s | $10^{-9}$ s |
| $10^2$              | $7 \times 10^{-11}$ s      | $10^{-9}$ s  | $7 \times 10^{-9}$ s  | $10^{-7}$ s |
| $10^3$              | $1.0 \times 10^{-10}$ s    | $10^{-8}$ s  | $1 \times 10^{-7}$ s  | $10^{-5}$ s |
| $10^4$              | $1.3 \times 10^{-10}$ s    | $10^{-7}$ s  | $1 \times 10^{-6}$ s  | $10^{-3}$ s |
| $10^5$              | $1.7 \times 10^{-10}$ s    | $10^{-6}$ s  | $2 \times 10^{-5}$ s  | 0.1 s       |
| $10^6$              | $2 \times 10^{-10}$ s      | $10^{-5}$ s  | $2 \times 10^{-4}$ s  | 0.17 min    |

# Computational Time of Various Classes of Problems According to Input Size

| TABLE 2 The Computer Time Used by Algorithms. |                         |              |                       |             |                       |                      |
|---|-------------------------|--------------|-----------------------|-------------|-----------------------|----------------------|
| Problem Size                                  | Bit Operations Used     |              |                       |             |                       |                      |
| $n$   | $\log n$                | $n$          | $n \log n$            | $n^2$       | $2^n$                 | $n!$                 |
| 10  | $3 \times 10^{-11}$ s   | $10^{-10}$ s | $3 \times 10^{-10}$ s | $10^{-9}$ s | $10^{-8}$ s           | $3 \times 10^{-7}$ s |
| $10^2$  | $7 \times 10^{-11}$ s   | $10^{-9}$ s  | $7 \times 10^{-9}$ s  | $10^{-7}$ s | $4 \times 10^{11}$ yr | *                    |
| $10^3$  | $1.0 \times 10^{-10}$ s | $10^{-8}$ s  | $1 \times 10^{-7}$ s  | $10^{-5}$ s | *                     | *                    |
| $10^4$  | $1.3 \times 10^{-10}$ s | $10^{-7}$ s  | $1 \times 10^{-6}$ s  | $10^{-3}$ s | *                     | *                    |
| $10^5$  | $1.7 \times 10^{-10}$ s | $10^{-6}$ s  | $2 \times 10^{-5}$ s  | 0.1 s       | *                     | *                    |
| $10^6$  | $2 \times 10^{-10}$ s   | $10^{-5}$ s  | $2 \times 10^{-4}$ s  | 0.17 min    | *                     | *                    |



# Computational Time of Various Classes of Problems According to Input Size

| TABLE 2 The Computer Time Used by Algorithms. |                         |              |                       |             |                       |                      |
|---|-------------------------|--------------|-----------------------|-------------|-----------------------|----------------------|
| Problem Size                                  | Bit Operations Used     |              |                       |             |                       |                      |
| $n$   | $\log n$                | $n$          | $n \log n$            | $n^2$       | $2^n$                 | $n!$                 |
| 10  | $3 \times 10^{-11}$ s   | $10^{-10}$ s | $3 \times 10^{-10}$ s | $10^{-9}$ s | $10^{-8}$ s           | $3 \times 10^{-7}$ s |
| $10^2$  | $7 \times 10^{-11}$ s   | $10^{-9}$ s  | $7 \times 10^{-9}$ s  | $10^{-7}$ s | $4 \times 10^{11}$ yr | *                    |
| $10^3$  | $1.0 \times 10^{-10}$ s | $10^{-8}$ s  | $1 \times 10^{-7}$ s  | $10^{-5}$ s | *                     | *                    |
| $10^4$  | $1.3 \times 10^{-10}$ s | $10^{-7}$ s  | $1 \times 10^{-6}$ s  | $10^{-3}$ s | *                     | *                    |
| $10^5$  | $1.7 \times 10^{-10}$ s | $10^{-6}$ s  | $2 \times 10^{-5}$ s  | 0.1 s       | *                     | *                    |
| $10^6$  | $2 \times 10^{-10}$ s   | $10^{-5}$ s  | $2 \times 10^{-4}$ s  | 0.17 min    | *                     | *                    |

← TRACTABLE → ← INTRACTABLE →

# Informal Definition

## Tractable Problem:

An algorithm that does not have a mathematical structure that is worse than a polynomial.

# Informal Definition

## Tractable Problem:

An algorithm that does not have a mathematical structure that is worse than a polynomial.

**Note:** The idea here is that a tractable problem can be solved in *a reasonable time* with *reasonable storage* (*i.e.*, you can solve it in less than the age of the known universe).

# Computational Complexity

- We will deal with this important subject in greater detail further on in this course.
- In particular, we will provide a formal definition of an *intractable function* which will require auxiliary definitions and an analysis of the **growth of functions**.
- Moreover, this will lead to a discussion of the **logical limitations of computation and of science itself** where, in the latter case, algorithms are non-existent for certain types of problems.



# The Philosophy of Data Structures

# PARKINSON'S LAW



The consequences of the forgoing subject-matter  
that are germane to this course.

# PARKINSON'S LAW

“Work expands so as to fill the time available for its completion.”

## C. Northcote Parkinson

Naval historian

Cyril Northcote Parkinson was a British naval historian and author of some 60 books, the most famous of which was his best-seller Parkinson's Law, which led him to be also considered as an important scholar in public administration and management. [Wikipedia](#)



# “Parkinson’s Law of Algorithmic Complexity”

The complexity of algorithms expands at a rate that is roughly proportional to the increase in the available computational power.

C. Northcote  
Parkinson





# “Parkinson’s Law of Algorithmic Complexity”

We study data structures so that we can learn to write more efficient programs. But why must programs be efficient when new computers are faster every year? The reason is that our ambitions grow with our capabilities. Instead of rendering efficiency needs obsolete, the modern revolution in computing power and storage capability merely raises the efficiency stakes as we attempt more complex tasks.

## Data Structures and Algorithm Analysis

Edition 3.2 (Java Version)

Clifford A. Shaffer  
Department of Computer Science  
Virginia Tech  
Blacksburg, VA 24061

January 2, 2012  
Update 3.2.0.3

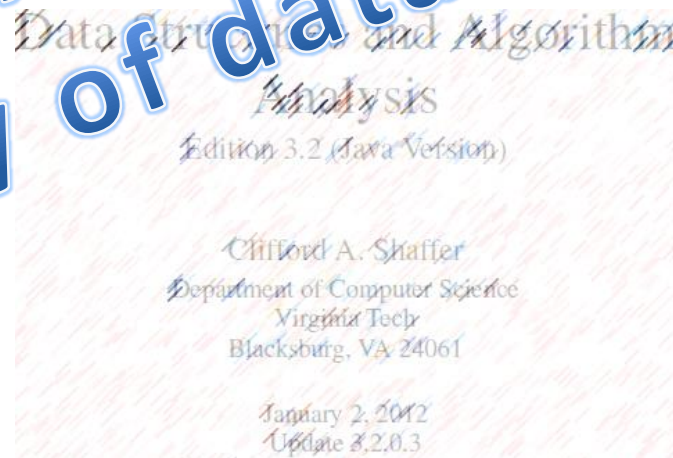
*Our Online Reference*

<https://people.cs.vt.edu/shaffer/Book/Java3e20120102.pdf>

# “Parkinson’s Law of Algorithmic Complexity”

We study data structures so that we can learn to write more efficient programs. But why must programs be efficient when new computers are faster every year? The reason is that our ambitions grow with our capabilities. Instead of considering efficiency needs obsolete, the modern revolution in computing power and storage capability merely raises the efficiency stakes as we attempt more complex tasks.

Herein lies the need for the study of data structures.

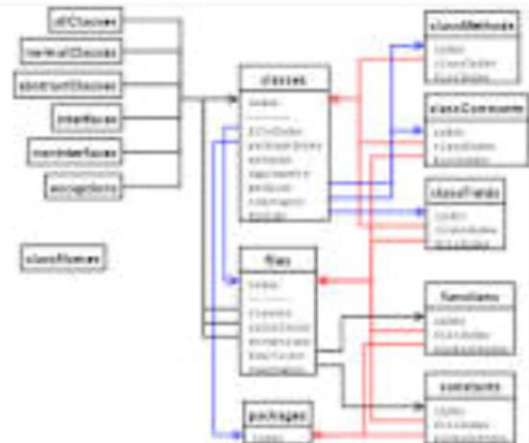


# Definition: Data Structure

## Data structure

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

[Wikipedia](#)



# Presentation Terminated

