

"I really hate this darn machine; I wish that they would sell it.
It won't do what I want it to, but only what I tell it."
- *The Programmer's Lament*

ITI 1120

Introduction to Computing I

Class Notes

Fall 2014

D. Inkpen

(contributors: G. Arbez, D. Amyot, S. Boyd, M. Eid
A. Felty, R. Holte, D. Inkpen, W. Li, S. Somé, A. Williams)

Not to be used or reproduced without permission of the authors

Table of Contents

- Section 1: Introduction.....3
- Section 2: Introduction to Java46
- Section 3: Algorithms and Translating them to Java...91
- Section 4: Tracing and Debugging.....119
- Section 5: Branching.....137
- Section 6: Loops and Arrays.....162
- Section 7: Program Structure.....213
- Section 8: Recursion.....238
- Section 9: Matrices.....273
- Section 10: Introduction to Objects.....303
- Section 11: Object-oriented Design.....337

"If you don't think carefully, you might believe that programming is just typing statements in a programming language ."
- *W. Cunningham, WikiWiki inventor*

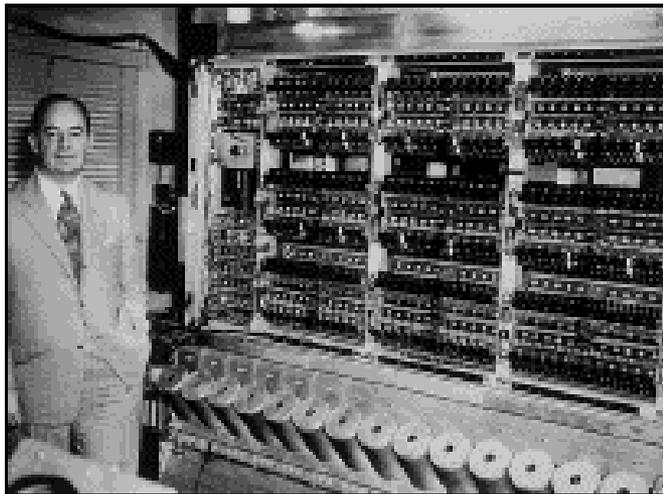
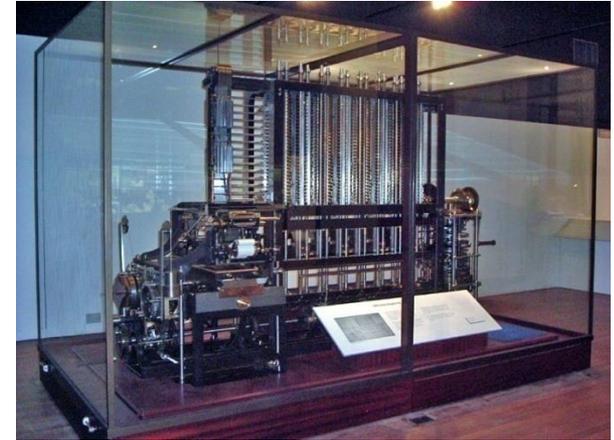
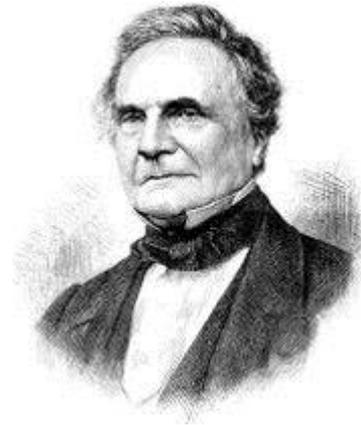
Section 1: Introduction

Objectives:

- Software development
- Specifications and algorithms
- Variables and expressions

Historical note ...

- Charles Babbage, British mathematician and engineer, designed and built, in 1833, parts of a **machine** that contained modern components such as: central processing unit, memory, and a data input device with punch cards.



- John von Neumann, Hungarian mathematician, participated in the development of the **first computer**: ENIAC (1945)
- The principle of the Von Neumann architecture: the data and the programs are encoded in the memory

Software Development

- This course is about solving problems using computer software.
 - Real-life software can include tens of millions of lines of program code, or it can be just a few lines of code in a life-critical system.
 - Software design teams can range from a single person to over a thousand people.
 - Software can live for decades (example: the SABRE airline reservation system is over 50 years old) and must be maintained to be successful.
 - To produce successful software, a systematic and rigorous development process is needed.
- **Software engineering:**
 - The process of designing software that functions correctly, and producing the software on time and within a budget.

Software Life Cycle

1. Requirements analysis
 - What problem are you trying to solve?
 - What are the needs of the users?
 - What resources are available?
 - Equipment, time, cost, people
 - Develop a plan
2. Design
 - Proposal for the solution of the problem within the constraints of the requirements
 - Model the software system
 - Structure of the software ("architecture")
 - Organization of data
3. Algorithm development
 - Determine the steps required to solve particular problems or sub-problems.

Software Life Cycle(continued)

4. Implementation

- Creation of program code:
 - Manually, or semi-automatically with tools.

5. Quality Assurance

- Testing: Running experiments to see if the software has the expected functionality and performance.
- Verification: Show that undesired functionality is not possible (when feasible).
- Debugging: The process of determining how to modify software to remove problems.

Software Life Cycle

6. Deployment

- How does the software get to the customer?
- How is the software installed and configured in the customer's environment?

7. Maintenance

- As customers report problems, how are the fixes developed, tested, and deployed?
- How are new features added?
- How are obsolete features retired
- Documenting is an activity that occurs throughout the cycle.
- ITI 1120 covers this life cycle but focuses mainly on algorithm development (#3) and implementation (#4).
 - Start with problem analysis
 - Produce a program to solve it.

Problem Analysis

- Our aim is to use a computer to solve problems.

"Computers are good at following instructions, but not at reading your mind."
- D. Knuth

- Problems are usually stated in a "natural language" (e.g. English), and it is up to you, as a software designer, to extract the exact problem specification from the informal problem statement.
- This involves **understanding** the problem, and clarifying:
 - What **data** is "given" or "input" (available for use when the problem-solving takes place)
 - What **results** are required
 - What **assumptions** are safe to make
 - What **constraints** must be obeyed.

Problem: Average of three numbers

- Informal statement:
 - John wants to know the total and average cost of the three books he just bought.
- **GIVENS**: descriptions and names of the values that are known
 - **cost1**, **cost2**, **cost3**: numbers representing the cost of each of John's books
- **RESULTS**: descriptions and names of the values to be computed from the givens
 - **avg**, the average of **cost1**, **cost2**, **cost3**
- No assumptions or constraints (for now!)

What is an Algorithm?

- An algorithm is a sequence of well-defined steps for solving a problem.
- Developing an algorithm for a problem is a creative process.
- Sometimes, part of this process involves **problem decomposition**, which involves deciding how to break the problem into smaller sub-problems.
- **Keep in mind:**
 - There can be many algorithms for the same problem, and you may have to choose the most appropriate solution.
 - There are problems for which there is no algorithm that solves the problem.

Problem: Area of a Triangle

- Informal statement: Given 3 numbers representing the lengths of the sides of a triangle, find the area of the triangle.
- **GIVENS:** `side1`, `side2`, `side3`: numbers representing the sides of a triangle
- **RESULT:** `area`: the area of the triangle
- **ASSUMPTIONS:** `side1`, `side2`, and `side3` are greater than zero.
- **CONSTRAINTS:** The three values `side1`, `side2`, and `side3` must form a closed triangle.
 - Example: If `side1` is 1, `side2` is 1, and `side3` is 5, a closed triangle cannot be formed.

Algorithm models (I)

- Before getting into the details of coding, you should build a model of the algorithm.
- The algorithm model we will use in this course has the following format (**important!**):

GIVENS

- A list of the names of given values (with comments)

RESULTS

- The name of the result (or a list of results) (with comments)

HEADER <RESULTS> ← <algorithm name> (<GIVENS>)

- It specifies the name of the algorithm, an ordered list of the givens, and an ordered list of the results.

(continued next page)

Algorithm models

ASSUMPTIONS

- A list of general conditions that are assumed to be true for the algorithm. The user of the algorithm is expected to meet these assumptions.

CONSTRAINTS

- A list of specific conditions that are assumed to be true when the algorithm starts, usually with respect to the values of the *GIVENS*. Normally, the constraints should be checked

BODY

- A sequence of instructions which, when executed, computes the desired results from the givens.

Data in Algorithm Models

- **Literals**: constant data values
 - Two types: numeric data, and textual data
- Examples of numeric literals:
`2, 3.14159, -14, -14.0`
- A **character** is enclosed in single quotes. The quotes are not part of the data. Examples of character data
`'a', 'A', '1', '!'`
- A **string** is a sequence of characters enclosed in double-quotes. The quotes are not part of the data. Examples of strings:
`"Hello", " foo", "bar ", "2", " ", ""`

Storing values in a computer

- The computer's memory consists of a large, but **NOT** unlimited, number of storage locations, each of which has an address to identify the location.
- A computer variable is a **location in memory** with an **address** that contains a value.
- A variable has the following attributes:
 - **Name**: Used by the computer as the address in the memory.
 - **Value**: The contents of the storage location.
 - **Type**: Constraints on the values that can be stored in a variable, or on the operations that can be performed.

Assigning Values to Variables

- To give a value to a variable is called **assignment**.
- In our model notation, we use
 $\text{aVariable} \leftarrow \text{anExpression}$
to denote assigning the value of the expression anExpression to a variable named aVariable .
 - Example: $x \leftarrow 3$
- On the right-hand side, anExpression could be a literal value, a variable (its value is taken), or an expression involving both.
 - Example: $x \leftarrow \text{anotherVariable}$
- Example: $x \leftarrow 4, y \leftarrow 3, x \leftarrow y$
 - What are the values of x and y after the execution?
- The names (variables) of the *GIVENS* of an algorithm are called **parameters** (or, formal parameters)

Simple Expressions

- An expression is a statement whose execution produces a value
- An expression may use operators, for example, arithmetic operators (+, -, * or ×, / or ÷)
 - Example: $40 + 10/5$, $\sqrt{49} * \log_2(32)$
- An expression can also use variables
 - Example: $40 + |y|/5$
- The results of an expression can be assigned to a variable
 - Example: $x \leftarrow 40 + y/5$

Operators

- Operators perform some sort of calculation on values.
- The number of values an operator uses may vary:
 - **Binary**: has two operands (values)
 - **Unary**: has one operand
 - Examples:
 - Subtraction: a binary operator.
 - Negation: a unary operator.
- Operators must be evaluated in a specific **order**, and some operators may have **precedence** over others.
 - For our algorithm models, we will use the order and precedence rules from mathematics.

Division and remainders

- It is often useful to perform division entirely within integers, as opposed to using real numbers.
- Two types of division:
 - Real: $11.0 / 4.0$ result: 2.75
 - Integer: $11 \div 4$ result: 2
 - The result of integer division is **truncated**; the fraction is cut off.
- When doing division in integers, it is also very useful to obtain the **remainder** from the division. This is done with the **mod** operator.
 - $11 \bmod 4$: result: 3
- Finding the remainder has several important uses:
 - The value of $x \bmod y$ is always in the range from 0 to $y - 1$.
 - The value of $x \bmod 10$ gives the last digit of x .

Exercise 1-1 - Algorithm for Average ?

GIVENS: num1, num2, num3 (three numbers)

RESULTS:

avg (the average of num1, num2, and num3)

HEADER:

BODY:

Exercise 1-2 - Another example



-
- Write an algorithm (computeP) that takes as input three values (a, b, c) and returns the proportion of each value out of their total.
 - First solution:

Exercise 1-2 - Another example (cont.)

- Second solution:

Intermediate Variables

- Often it is useful within an algorithm to use a variable that is neither a given nor a result used to temporarily hold a value.
- These **INTERMEDIATE** variables should be listed and described along with the givens and results at the start of an algorithm definition.
- Their values are not returned to the calling statement, nor are they remembered from one call to the next.
- Intermediate variables are used mainly for improving readability of the algorithm or for the efficiency of the algorithm.

More on INTERMEDIATES

- INTERMEDIATES have values that can vary.
 - They may vary depending on the problem instance (in other words, depending on the values of the GIVENS).
 - They may change during computation.
- INTERMEDIATES that do not vary are called **CONSTANTS**.
 - Their values are fixed.
 - They are the same no matter what the values of the GIVENS are.
 - Their values will not change during the computation.
 - They can be given names that help document the algorithm and make it more readable.
 - Representing constants by names reduces maintenance effort.

Exercise 1-3: Average out of 100



-
- Write an algorithm that takes three scores (each out of 25) and computes their average (out of 100).
 - (Idea: average the scores, then convert the result to be out of 100.)

Exercise 1-4 - Last example...



- ... Without a constant

- ...with constants *GST* and *PST*

...

Body

...

```
tax1 ← c1 * 0.07 // GST
```

```
tax2 ← c2 * 0.07 // GST
```

```
tax3 ← c3 * 0.07 // PST
```

```
tax4 ← c4 * 0.07 // GST
```

...

Summary of Variables

- **GIVENS** are the variables that contain input values. They vary from one call (see next slide) to another. In other words, their values can be different for each problem instance.
- **RESULTS** are the variables that contain answers that are generated by an algorithm.
- **INTERMEDIATES** are the other variables used in solving the problem.

Sub-program

- The sub-program consists of a sequence of computer instructions that accomplish some task
 - A sub-program typically operates on a set of variables (givens and intermediates)
 - A sub-program typically produces some results.
 - The algorithm model is a model of the subprogram
 - Sub-programs can be "called" from other sub-programs - more on this in Section 3

Representing Input in Algorithm Models

- Would like to define algorithm models (i.e. sub-programs) to interact with the user
- Need to define algorithms that represent the reading of values from the keyboard and writing of strings to the console
- Lets define the following algorithm models (headers) to read from the keyboard:
 - numVar \leftarrow readReal() to read a real number from the keyboard
 - intVar \leftarrow readInteger() to read an integer from the keyboard
 - strVar \leftarrow readLine() to read a line of input, i.e. a string from the keyboard
 - boolVar \leftarrow readBoolean() to read "true" or "false".
 - charVar \leftarrow readCharacter() to read a single character from the keyboard.
 - numVar \leftarrow random() to generate a real number greater or equal to 0.0 and less than 1.0
- Examples of "calling" the algorithms (sub-programs)
 - x \leftarrow readReal()
 - a \leftarrow readInteger()
 - (the above are calls to algorithm models)
 - (also note that we are not concerned with how the algorithm models work)

Representing Output in Algorithm Models

- Lets define the following algorithm models to write to the terminal console:
 - `println(<argument list>)`
 - Prints a string on a line to the console (the cursor is moved to the next line)
 - `print(<argument list>)`
 - Prints a string to the console (the cursor is left at the end of the printed string)
 - The printed string is the concatenation of the arguments in the `<argument list>`. An argument can be a string, literal, the name of a variable, or an expression.
 - Examples:
 - `print("Please enter the value: ")`
 - `println("The result is ", x)`
 - `println("Adding a to b gives ", a+b)`
 - The argument list is translated to a concatenation expression in Java

Instruction block

- An instruction block consists of a sequence of computer instructions
 - The computer executes each instruction in sequence
 - Note that a single instruction can be made up of many operations (+, -, ← +)
 - The body of the Algorithm model is an instruction block

Models vs. Computation

- We are using mathematical operators and values as a **model** for computation.
- A model is only an abstraction that captures the essentials of the problem, but not all the details. When it comes to actual computation, additional constraints may be introduced:
 - Can we use numbers of unlimited range?
 - What about types of values?
 - How fast are the computations performed?
 - Do we have enough memory for all of the variables?
 - Does an operator in a programming language work exactly the same way as in mathematics?

Conversion to source code

- **Coding** = translating an algorithm into a particular programming language so it can be executed on a computer.
- **Do not be too quick to code!** It is much better to spend time on your algorithm, mentally checking it, thinking about it, and “tracing” it on test data to be sure that it is correct.

“The sooner you start to code,
the longer the program will take.” - R. Carlson

- Coding is largely a mechanical process, not a creative one.
- Both algorithm development and coding require very careful attention to detail.

Programming Languages

- Inside the computer, the hardware understands only sequences of electrical signals, represented by digits 0 and 1 ("machine language").
 - This language is usually specific to a particular computer processor.
- Since long sequences of zeros are difficult for people to work with, programs are normally created in a programming language, and then translated to machine language.
- Programming languages have a very precise grammar ("syntax") and unambiguous meanings of statements ("semantics").

Types of Programming Languages

- Machine-level languages
 - Numbers. The only language computer understands directly
- Assembly languages
 - Instructions made up of symbolic instruction codes
 - Assembler converts the source code to the machine language
- **High-level languages** (third generation)
 - Use a series of English-like words to write instructions
- Forth-generation languages:
 - Syntax closer to human language (for databses, e.g., SQL)
 - Provide visual or graphical interface for creating source code (e.g., VisualBasic.Net)

Programming Paradigms

High-level languages:

- imperative / procedural programming
(e.g., Basic, Pascal, Fortran, C)
- **object-oriented programming**
(e.g., Java, C++, SmallTalk)
- functional programming (e.g., Lisp, ML)
- logic programming (e.g., Prolog)

Programming languages

Language	When invented	Typical uses
FORTRAN	1956	Scientific computation
LISP	1958	Expert systems
COBOL	1960	Business
APL	1962	Mathematics, statistics
Basic	1971	Teaching, business
C	1972	Operating system creation
C++	1984	General purpose, communications
Java	1993	General purpose, internet applications

Other languages

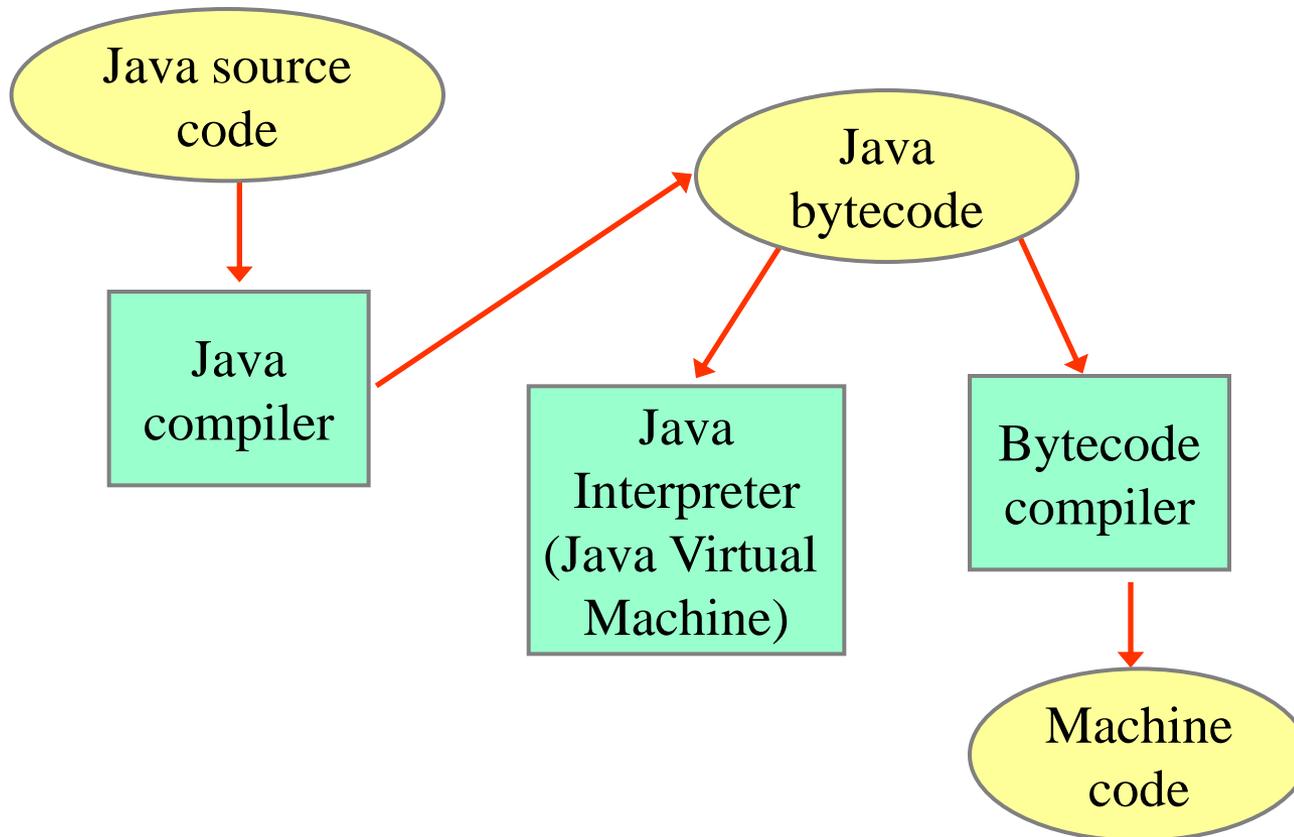
Language	When invented	Known for...
Pascal	1971	Teaching
Smalltalk	1972	Object-oriented
Prolog	1972	Logic, expert systems
Ada	1979	US Defence Dept. software standard language
*SQL = Structured Query Language	1979	Databases
*HTML = HyperText Markup Language	1992	Web page formatting
*XML = eXtensible Markup Language	1996	Internet data exchange
C#	2002	Microsoft's answer to Java

* Special-purpose languages: not for general programming.

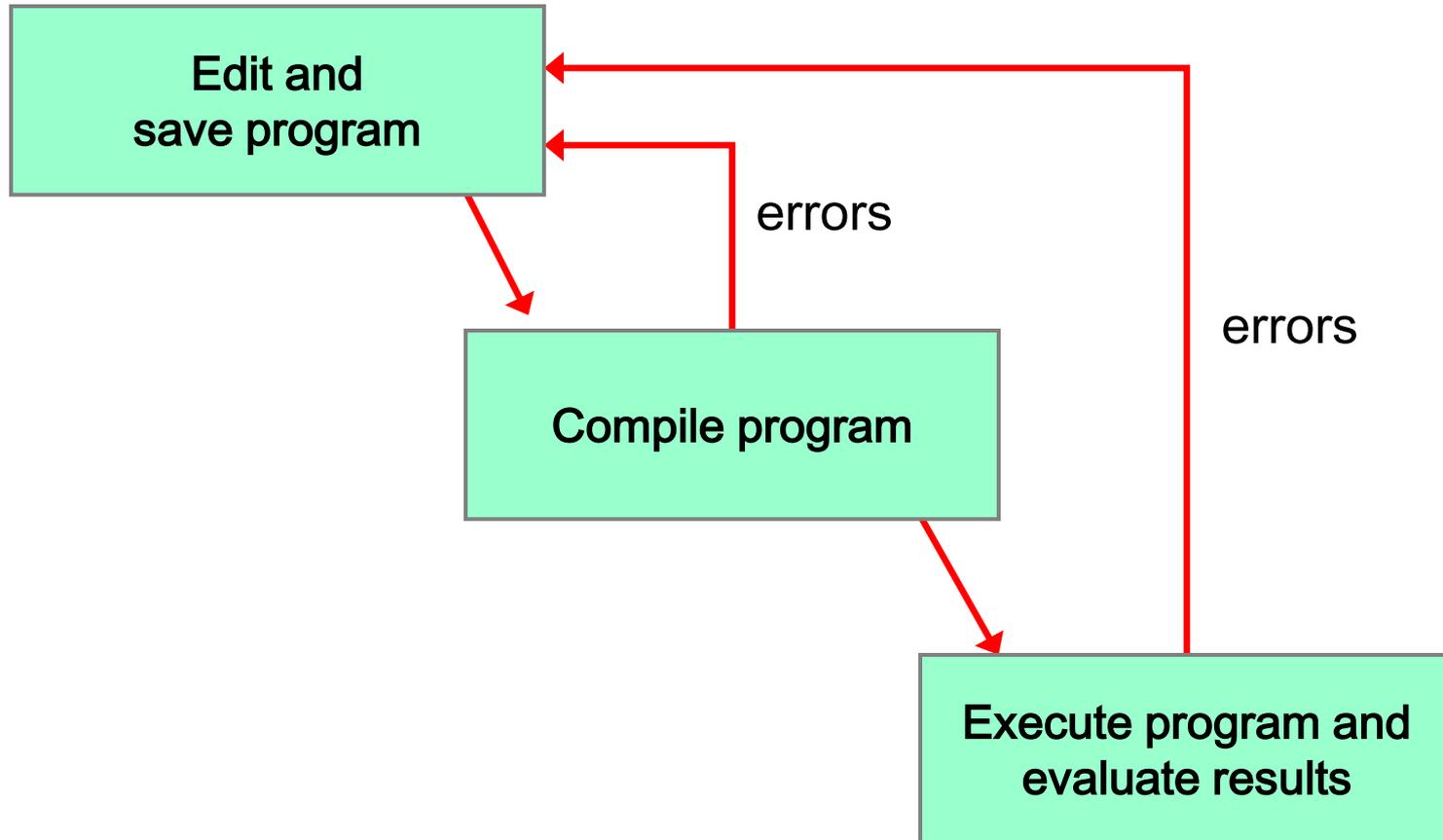
Compilers vs. Interpreters

- Compiler = a program that translates source code into machine code. At the end of the compilation the machine code can be executed.
- Interpreter = a program that translates each line of code into machine language and executes it before translating the next line.
- Differences:
 - Execution of compiled code is faster.
 - Compilers can do optimization.
 - Interpreters used for prototyping, multiplatform.

Translation in Java



Basic Program Development



Testing, Debugging, and Maintenance

- **Testing** = looking for errors (“bugs”) in an algorithm or program by executing it on test data (givens) and checking the correctness of the results. Big programs are usually impossible to test completely.
- **Debugging** = locating and correcting an error in an algorithm or program.
- **Maintenance** = changing an algorithm or program that is in use, updating, fixing errors.

Three Types of Errors

1. **Syntax errors:** These are illegal combinations of symbols that do not obey the rules of the programming language.
 - Symptom: the program will not compile.
2. **Run-time errors:** These are errors which result from the data values used.
 - Symptom: the program crashes while running.
3. **Logic (semantic) errors:** These are errors which result from incorrect reasoning in the program. They probably occur because the algorithm is wrong.
 - Symptom: the program runs, but the results are wrong.

Documentation

- Documentation is all of the materials that make software easy to understand for both software designers, and users of the software.

"If you can't explain it simply, you don't understand it well enough."
- A. Einstein

- **Internal documentation** (such as comments, descriptive variable names) occurs inside the program.
- **External documentation** (such as models, user manuals, etc.) is documentation which is outside of the program.

"The only way to learn a new programming language is by writing programs in it."
- *B. Kernighan & D. Ritchie*

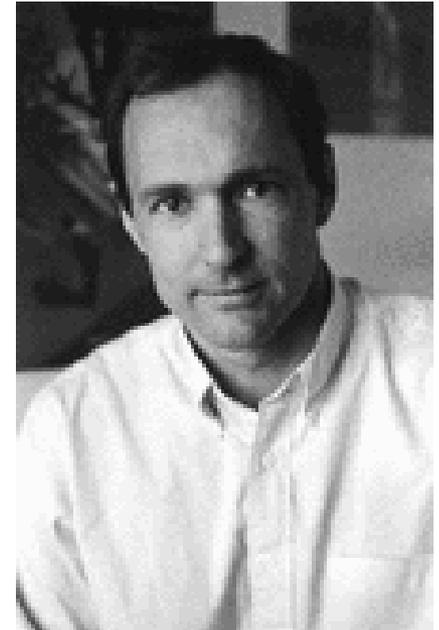
Section 2: Introduction to Java

Objectives:

- Data Types
- Characters and Strings
- Operators and Precedence
- Reading and Displaying Values
- Class Template

Historical note ...

- In 1990, [Tim Berners-Lee](#), researcher at CERN in Geneva, developed the *World Wide Web* technology, in order to facilitate information access on the Internet.
- He initiated many standards, among which the most utilised are HTTP, URL, and the HTML language.
- Now he is working on the [Semantic Web](#)



In 1992, at Sun Microsystems, [James Gosling](#) (born in Canada) and his team invented, the programming language Oak, renamed [Java](#) in 1994.

Conversion to source code

- Our approach to software development consists of developing and testing first an algorithm model and then to **TRANSLATE** (convert or **code**) to source code using a programming language.

“When a new language will allow to program in English, we will discover that programmers cannot speak English.” - Anonymous

- Coding algorithm models is largely a mechanical process, and requires very few decisions.
- For each instruction block of the software model, we shall study an approach to translate into source code. The algorithms can be translate block by block.

Java

- The programming language used in this course is Java.
- A Java program consists of a number of classes. Every class is saved in its own file with the same name and extension **.java**.
- A method (e.g. **main**) can also call other methods, that can be added to the class. Java methods are *sub-programs* designed from problem decomposition.
- At this stage, we will develop software that uses ONE class containing **two** methods. The method **main** will implement the interaction with the user and the second method solves the problem.
 - You will develop two algorithm models, one for each method.
 - The method **main** is the first method executed when the software is started. It will call the problem solving method.

Data in Java

- Data items used in a Java program are represented by **LITERALS** and **VARIABLES**.
 - A literal represents a constant, such as
 - **123** (an integer),
 - **12.3** (a real number),
 - **'a'** (a character),
 - **true** (a Boolean value).
 - A variable uses an identifier (e.g. **x**) to represent a data item.
 - It represents the address of the memory location where the value is stored.
 - Only after it has been initialized (assigned a value), should you consider that its value is "known"

Data Types

- Data items in Java have **TYPES**.
- A data type specifies:
 1. Which values the data item can take.
 - That is, the set of legal literals.
 2. Which operations are available to manipulate this data item, and what those operations do.
 3. How this data item is to be stored in a computer.
 - We will leave details of internal representation format for another course.

Java Primitive data types

- A data item has a **PRIMITIVE** type if it represents a single value, which cannot be decomposed.
- Java has a number of pre-defined primitive data types. In this course, we will use the following types:
 - int** represents integers (e.g. **3**)
 - double** represents "real" numbers (e.g. **3.0**)
 - char** represents single characters (e.g. **'A'**)
 - boolean** represents Boolean values (e.g. **true**)

Type `int`

- A variable of type `int` may take values from `-2147483648` to `2147483647`.
 - Exceeding the range of legal values results in **OVERFLOW**, a run-time error.
- The following operators are available for type `int` :
 - + (addition)
 - (subtraction, negation)
 - * (multiplication)
 - / (integer division, where fraction is lost; result is an `int`)
Example: `7 / 3 gives 2`
 - % (remainder from division)
Example: `7 % 3 gives 1`
 - `==` `!=` `>` `<` `>=` `<=` (comparisons: these take two values of type `int` and produces a **boolean** value)

Short-Hand Notation

- A very common expression is used to increment integer variables:

```
int i;  
i = i+1;
```

- This is so common in computer programs that a short-hand notation is used:

```
i++;
```

- This will be very practical when defining loops (Section)

Type **double** (Literals)

- Type **double** represents “real” numbers approximately from -1.7×10^{308} to 1.7×10^{308} with 15 accurate significant digits.
 - While there are a lot of **double** values, the set of legal double values is still **finite**, and so they are only an **approximation** to real numbers.
 - After a computation, the computer has to choose the closest **double** value to a real result: this can introduce “**round-off**” errors.
- Format of large or small values:
 $123456000000000000.0 = 0.123456 \times 10^{17}$ is **0.123456e17**
 $0.00000123456 = 0.123456 \times 10^{-5}$ is **0.123456e-5**
- If the value of **the exponent** is more negative than -308, the result is **UNDERFLOW**, and the value will be replaced by zero. This happens quietly, and is **not** a run-time error.

Type **double** (Operators)

- The following operators are available for type **double** :
 - + (addition)
 - (subtraction, negation)
 - * (multiplication)
 - / (division in "real" numbers, result is a **double**)
 - > < (comparisons: these take two values of type **double** and return a **boolean** value)
- **WARNING**: Using **==** (or **!=** **>=** **<=**) to compare two values of type **double** is legal, but **NOT** recommended, because of the potential for round-off errors.
 - Instead of **(a == b)**, use **(Math.abs(a - b) < 0.001)**
where **Math.abs(x)** returns $|x|$

Type **boolean**

- Only two literals: **true** and **false**
- The following operators are available for type **boolean**:
 - ! NOT (a **unary** operator, similar to a negative sign)
 - && AND
 - || OR
 - comparison : **==** **!=**

Type `char` (1)

- Characters are individual symbols, enclosed in **single** quotes
- Examples
 - letters of the alphabet (upper and lower case are distinct) `'A'`, `'a'`
 - punctuation symbol (e.g. comma, period, question mark) `','`, `'.'`, `'?'`
 - single blank space `' '`
 - parentheses `'('`, `')'`; brackets `'['`, `']'`; braces `'{'`, `'}'`
 - single digit (`'0'`, `'1'`, ... `'9'`)
 - special characters such as `'@'`, `'$'`, `'*'`, and so on.

Type `char` (2)

- Each character is assigned its own numeric code:
 - **ASCII** character set (ASCII = American Standard Code for Information Interchange)
 - 128 characters
 - most common character set (if you speak American English 😊)
 - used in older languages and computers
 - **UNICODE** character set
 - over 64,000 characters (international)
 - includes ASCII as a subset
 - used in Java

Collating Sequence

- In the computer, a character is stored using a numeric code.
 - The most commonly used characters in Java have codes in the range 0 through 127.
 - For these characters the UNICODE code is the same as the ASCII code.
- Examples:

character	'a'	'A'	' '	'0'	'?'
UNICODE value	97	65	32	48	63

- **Exercise 2-1** - The numerical order of the character codes is called the collating sequence. It determines how the comparison operator works on characters:

'A' < 'a' is
while
'?' < ' ' is



Digit and Letter Codes

- Important features of the ASCII/UNICODE codes:
 - Codes for the digits are consecutive and ordered in the natural way (the codes for '0' to '9' are 48 through 57 respectively). Thus
 - '2' < '7' gives **true**
 - '7' - '0' gives **7**
 - The same is true of the codes for the lower case letters ('a' to 'z' have codes 97 through 122 respectively). Thus
 - 'r' < 't' gives **true**
 - The same is true of the codes for the upper case letters ('A' to 'Z' have codes 65 through 90 respectively).
 - Note they are smaller than the codes for the lower case letters: 'b' - 'A' gives **33**

Exercise 2-2 - Test for Upper case ?

- Suppose the variable **x** contains a value of type **char**.
- Write a Boolean expression that is TRUE if the value of **x** is an upper case letter and is FALSE otherwise.
 - Note that you don't need to know the actual code value of the characters!

Special characters

- Some characters are treated specially because they cannot be typed easily, or have other interpretations in Java.
 - new-line character `'\n'`
 - tab character `'\t'`
 - single quote character `'\''`
 - double quote character `'\"'`
 - backslash character `'\\'`
- All of the above are **single** characters, even though they appear as two characters between the quotes.
- The backslash is used as an **escape** character: it signifies that the next character is not to have its "usual" meaning.

Type conversion

- In general, one *CANNOT* convert a value from one type to another in Java, except in certain special cases.
- When a type conversion is allowed, you can ask for a type conversion as follows (this is called “casting”):

`(double) 3` gives `3.0`

`(int) 3.5` gives `3`

`(int) 'A'` gives `65`

`(char) 65` gives `'A'`

(note loss of precision!)

(this is the UNICODE value)

- **WARNING:** Type conversions with unexpected results have resulted in serious software problems.
 - One such error caused the self-destruction of the Ariane 501 rocket in 1996.
- The best strategy: **DON'T** mix types or values, unless absolutely necessary!

Strings

- A **STRING** is a collection of characters.
 - There is **NO** primitive data type in Java for a string.
 - We will see later how to deal with strings in general.
- String literals (constants) can be used to help make your program output more readable.
 - String literals are enclosed in **double** quotes:
"This is a string"
- **Watch out for:**
 - **"a"** (a string) versus **'a'** (a character)
 - **" "** (a string literal with a blank that has length 1) versus **""** (an **empty** string: a string literal of length 0)
 - **"257"** (a string) versus **257** (an integer)
 - **" "** **<<** **>>** are not valid quotes in Java!

String Concatenation

- Strings can be **CONCATENATED** (joined) using the **+** operator:
 - **"My name is " + "Diana"** gives **"My name is Diana"**
- String values can also be concatenated to values of other types with the **+** operator.
 - **"The speed is " + 15.5** gives **"The speed is 15.5"**
 - Because one of the values for the **+** operator is a **String**, the double is converted to a **String** value **"15.5"** before doing the concatenation.

Precedence of Operators

- Operators are evaluated left-to-right, with the following precedence (all operators on the same line are treated equally):

() (expression) **[]** (array subscript) **.** (object member)

+ **-** (to indicate positive or negative values) **!** (not)

***** **/** **%**

+ **-** (for addition or subtraction of two values, concatenation)

< **>** **>=** **<=**

== **!=**

&&

||

= (assignment to variable)

Exercise 2-3 - Operator Precedence ?

- What is the order of evaluation in the following expressions?

$a + b + c + d + e$

$a + b * c - d / e$

$a / (b + c) - d \% e$

$a / (b * (c + (d - e)))$

Variables in Java

- The givens, results, and intermediates, of an algorithm are represented by variables in a Java program.
- To use a variable, you must **declare** it first.
 - A declaration of a variable specifies its type, its name, and, optionally, its initial value.
 - A declaration reserves memory for the variable.
 - Examples

```
int x = 0;           // An int variable x with initial value 0
double d;           // A double variable not initialized
char c = ' ';       // A char variable initialized as a blank
boolean b1 = false; // A boolean variable initialized to false
```

- Variable declarations end with a semicolon.

Exercise 2-4 - Some Math

- To assign a value to a variable the operator "=" is used, as in **x = 2;**
- Declare the following variables:
 - Three real variables: **cost1**, **cost2**, **cost3**
 - A real variable **avg**
 - An expression that computes the average of the variables **cost1**, **cost2** and **cost3** and assigns the results it to **avg**.
- Declare the integer variables **intVar1**, **intVar2**, **quotient**, and **remainder**
 - Provide the expression that assigns the quotient of **intVar1** divided by **intVar2** to **quotient**.
 - Provide the expression that assigns the remainder of **intVar1** divided by **intVar2** to **remainder**.

Exercise 2-4 - Some Math



```
// Variable declarations
```

```
// Compute the average
```

```
// Variable declarations
```

```
// Compute quotient and remainder
```

Primitive variables

- When a variable is declared as a primitive type,
 - there is a location allocated in the computer memory to hold the value of this variable.
 - The name of the variable (represents the address of the variable) is permanently associated with this memory location.

Reference variables

- If a variable is of an array type, or a user-defined type (an "object", to be introduced later), the variable is a reference variable.
- This variable is used to "reference" an array, or an object.
- When the reference variable is declared, there is no memory allocated to hold the array, or the object. You must use operator **new** to create the array, or the object, and associate the variable to this array or object (*or different ones*).
 - The reference variable is a variable that contains an **address**
 - Associating an array or object to a reference variable means assigning the address of the array/object into the reference variable
 - More on reference variables later!

Comments

- A comment is an explanation of the meaning of your variable or code.
- Java comments have (mainly) two forms:
 - A comment starting from `//` to the end of a line, such as
`int size = 30; // number of students in a class`
 - A comment between `/*` and `*/`, such as
`/* this program calculates the sum of N
* numbers and displays the result.
*/`
- Comments are used to help people to read the program, and are ignored by the computer.

"Being forced to write comments actually improves code,
because it is easier to fix a crock than to explain it."

-- G. Steele

Javadoc

- The Java development kit also includes a tool called javadoc. This tool will take specially formatted comments, and build documentation web pages.
- A javadoc comment:

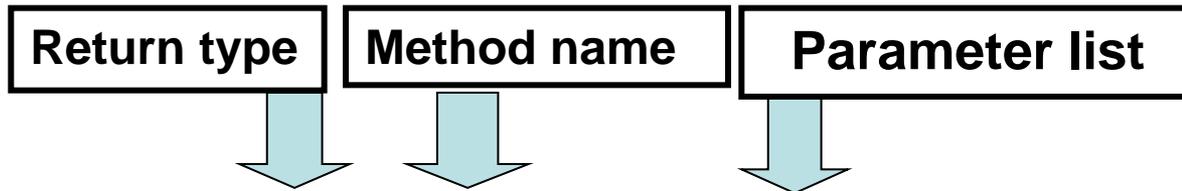
```
/**  
 * This text will appear on a web page.  
 *  
 * @author Diana Inkpen  
 * @version 2009  
 */
```
- More information at <http://java.sun.com/j2se/javadoc>

Java Methods

- As already mentioned, the Java Method is a sub-program
- Every method in Java has
 - **Return type**: It specifies what is the type of the result of the method. If there is no result, this type is **void**.
 - For now always include keywords "public static" before the return type.
 - **Name**: To reference the method (same as the name of an algorithm).
 - **Parameter list**: It specifies the name and the type of the parameters, in order.
 - Note that the parameter list is a list of "variable declarations".
 - **Body**: An instruction bloc consisting of a sequence of Java instructions which is the translation of the algorithm body
 - Notice the form of the instruction bloc: the use of the braces { } and the indentation.

Method Template

```
// METHOD Name: Short description of what the  
// method does, plus a description of  
// variables in the parameter list
```



```
public static double avg3(int a, int b, int c)  
{  
    // DECLARE VARIABLES/DATA DICTIONARY  
    // intermediates and  
    // the result, if you give it a name  
  
    // BODY OF ALGORITHM  
  
    // RETURN RESULT  
    return <returnedValue>;  
}
```



A Java method "returns" a value

- A Java method may return **no** or **one** value.
 - It is not possible to return more than one value in Java, as can be done with our algorithm models. However...
 - This value may be a primitive type or a reference type. For example, a method may return the reference to an array.
 - For now, we shall develop algorithm models that use only one results variable
 - The Java method returns a "value", it does not assign a value to a variable
 - The call to a Java method can be interpreted as "reading the value of a variable" and as such, can be placed anywhere (i.e. expressions) a variable name is used.
 - In this course, to align with the algorithms, we shall assign the return value of a method call to a variable and in methods always declare a result variable
 - More details on method calls in Section 3

A method with return type **void**

- If a method does not return a value, i.e., return type **void**, then a call activates the method to do whatever it is supposed to do.

```
public static void print3(int x, int y, int z)
{
    System.out.println("This method does not return any value.");
    System.out.println( x );
    System.out.println( y );
    System.out.println( z );
}
```

- When the method is called by
print3(3, 5, 7);
it simply prints these arguments to the screen.

"Standard" Java Methods

- Java offers "standard" classes and methods
 - For example methods for doing certain math functions
 - Used `Math.abs(a-b)` to implement $|a-b|$ in slide [#56. Type double \(Operators\)](#).
 - To translate \sqrt{x} use `Math.sqrt(x)`
 - See section 5.10 in Liang for a list of Math methods
 - Other methods exist to read from the keyboard and write to a terminal console
 - Shall explore such methods in just a moment.

Java Output

- Java uses a **console** for basic output; i.e. for communicating to the user.
 - If you run a program from a command line console (terminal), the console (window and keyboard) is used for input and output.
 - If you run a program from a development environment (e.g. DrJava), the console may be a special window.
- Most real-life applications use **dialogs** (a special window) for output.



Java Output

- To output a value to the console, we will use two Java methods from the class **System.out**:

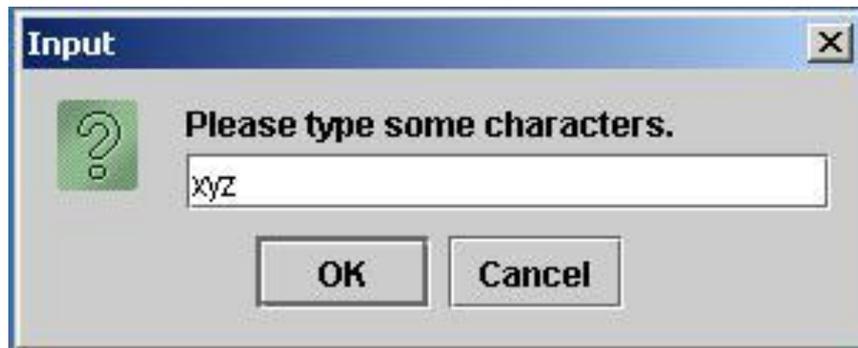
```
System.out.println("aaa/nbbb")
```

```
System.out.print(...)
```

- Method **println()** will append a new-line character to the output, while **print()** does not.
- Calls to these methods are unusual in that they will accept **any type** of argument or **no argument**.
 - Method call **println()** with no argument can be used to print a blank line.

Java Input

- Input is not as simple as output.
- Basic input from the console/keyboard has been difficult until recently.
 - Java 5.0 has made this much easier.
- Many applications use dialogs for input as well:



Reading Input from the Keyboard

- Older versions of Java used a complicated construction to read from the keyboard.
- To keep things simple, we provide the Java class **ITI1120**, available on the course website. It is compatible with Java 1.4, 1.5/5.0, 1.6/6.0 and 1.7/7.0.
- In your assignments, include the file **ITI1120.java**, in the same directory as your program. Then you can invoke the methods of this class in order to read a value (or several values) from the keyboard.

The methods of the class **ITI1120**

ITI1120.readInt() : Returns an integer of type **int**
ITI1120.readDouble() : Returns a real number of type **double**
ITI1120.readChar() : Returns a character of type **char**
ITI1120.readBoolean() : Returns a value of type **boolean**
ITI1120.readDoubleLine() : Returns a array of **double**
ITI1120.readIntLine() : Returns an array of **int**
ITI1120.readCharLine() : Returns an array of **char**
ITI1120.readString() : Returns a string of type **String**

- The value returned by these methods needs to be assigned to a variable of the right type.
- After the invocation of a method, the program will wait for the data to be entered.
- When you input a value from the keyboard and press ENTER, the program stores the value in a variable that you specify, and continues the execution.

Examples of using the `ITI1120` class

```
int x = ITI1120.readInt( );
```

- If you enter `123` and press ENTER, `x` will be assigned the value `123`.
- The method `readDouble` functions in a similar way.

Program Template

(To be Used for Assignments)

```
// Comments identifying you
// Comments describing what the program does
// and how it is used (e.g. what input is required)
import java.io.* ;

class PutYourClassNameHere
{
    public static void main (String args[ ])
    {
        // DECLARE VARIABLES/DATA DICTIONARY
        // PRINT OUT IDENTIFICATION INFORMATION
        System.out.println("ITI1120 Fall 2012, Assignment 0, Question 57");
        System.out.println("Name: Grace Hoper, Student# 12345678");
        System.out.println();

        // READ IN GIVENS use our ITI1120 special class for keyboard input
        // BODY OF ALGORITHM - Call to the method to solve problem
        // PRINT OUT RESULTS AND MODIFIEDS
    }
    // Put the method called by "main".
}
```

More on Reading Input from the Keyboard (an alternative with Java 5.0 and Java 6.0)

- Java now comes with a class called **Scanner** that simplifies input.
- How to use a **Scanner**:
 1. Create a new scanner, and assign it's reference to a reference variable **keyboard**.
 2. Each time you want a value from the keyboard, call a method via the reference variable **keyboard**.
- The method that you call depends on which type of value you want for input (see next page).
 - The scanner will read the characters you type, and convert them - if possible - to a value of the type you requested.

Methods in class **Scanner**

- nextInt()** : Returns an integer of type **int**.
- nextDouble()** : Returns a "real" number of type **double**
- nextBoolean()** : Returns a value of **true** or **false** as a value of type **boolean**
- nextLine()** : Returns a **String** with the entire remaining contents of the line.

- The returned value of these method has to be assigned to a variable of corresponding type.
- When your program reaches a call to one of these methods, the program will suspend and wait for your input.
- When you enter a value from the keyboard and press ENTER, then the program will read the input and store the value you entered into the variable you specified.

Examples of using **Scanner**

- Initialize a scanner:

```
Scanner keyboard = new Scanner( System.in );
```

```
int x = keyboard.nextInt( );
```

- If you enter **123** and press ENTER, **x** will have the value **123**.

```
boolean b = keyboard.nextBoolean( );
```

- If you enter **true** and press ENTER, **b** will have the **boolean** value **true**.

```
String s = keyboard.nextLine( );
```

- Method **nextLine** puts **ALL** characters (including spaces) that you type on a line into a String referenced by **s**.

Alternative Program Template

```
// Comments identifying you
// Comments describing what the program does
// and how it is used (e.g. what input is required)
import java.util.Scanner;
Import java.io.*;

class PutYourClassNameHere //your own algorithm name.
{
    public static void main (String[] args)
    {
        // SET UP KEYBOARD INPUT
        Scanner keyboard = new Scanner( System.in );

        // DECLARE VARIABLES/DATA DICTIONARY
        // PRINT OUT IDENTIFICATION INFORMATION
        System.out.println();
        System.out.println("ITI1120 Fall 2012, Assignment 0, Question 57");
        System.out.println("Name: Grace Hoper, Student# 12345678");
        System.out.println();

        // READ IN GIVENS
        // BODY OF ALGORITHM - Call to algorithm to solve problems
        // PRINT OUT RESULTS AND MODIFIEDS
    }
    // Place the method called by "main".
}
```

"Don't ask what it means, but how it is used."
- L. Wittgenstein

Section 3: Algorithms and Translating them to Java

Objectives:

- Calling Algorithms
- Information Passing
- Modified arguments
- Translating Algorithms to Java

Historical note ...

- [Ada Byron](#), countess of Lovelace, mathematician and collaborator of C. Babbage, defined the principle of successive iterations in the execution of an operation (1840).
- She created the first computer algorithm; she is considered the first programmer.
- There is a programming language named after her: [Ada](#)

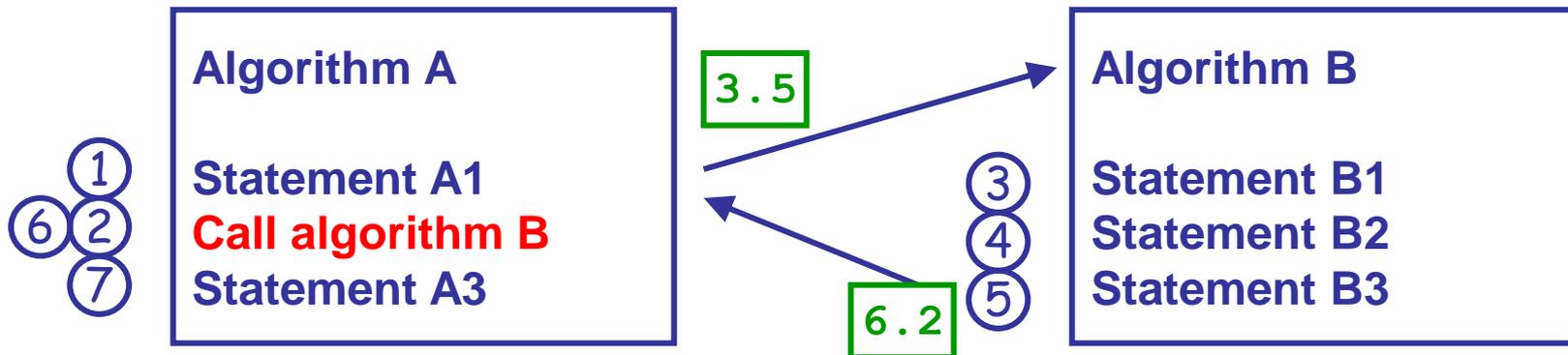


Using Algorithms

- When developing an algorithm, it is a good idea to make as much use as possible of existing algorithms (ones you have written or that are available in a library).
- You can put a **CALL** statement to any existing algorithm wherever it is needed in the algorithm you are developing.
- Be sure you get the **ORDER** of the givens, modifieds, and results correct.
- To call an algorithm you need to know its header but not how it works - you must just trust that it works correctly.

Algorithm model calls

- When one algorithm model calls another:
 - The algorithm model that is currently executing stops and waits at the point of the call.
 - Values may be “passed” to the called algorithm model.
 - The called algorithm model executes.
 - When the called algorithm model finishes, result values may be passed back to the calling algorithm model .
 - The calling algorithm model restarts and continues



Invoking an Algorithm

- “Calling” is the execution of an algorithm with specific data values.
- A “call” statement is identical to the header except for the names of the givens and results, which are replaced with values.
`theAvg ← markResult(10, 7, 4)`
invokes our algorithm with givens `score1=10`, `score2=7`, `score3=4` and returns the results in `avgPct` (average).
 - Note: If a variable name is used as an argument, then it is the value contained in the variable that is passed to the called algorithm (pass by value)
 - Note: the result value is stored in the variable specified in the call (`theAvg`).
- **ATTENTION:** Information is passed between the call statement and the algorithm based on the **ORDER** of the givens and results, not their names.
- The values passed to an algorithm are called the arguments (or actual parameters). The arguments (**actual parameters**) match the parameters (**formal parameters**) according to their orders.

Variable scope and duration

- **Scope:** Where you can use a variable's name.
 - General rule: Variables can be only be accessed inside their own algorithm.
 - If you use the name **x** in two algorithms, the names represent two **different locations** in memory in each algorithm and can contain two **different** values.
- **Duration:** The "lifetime" of a variable's value.
 - When an algorithm finishes execution, the values of all variables are forgotten.
 - The values of the RESULTS will be passed back to the calling algorithm.
 - When you call an algorithm again, new values are used for all variables.
 - How the computer works: whenever a subprogram is executed (algorithm), a piece of Working memory is assigned for the duration of its execution.

Algorithm Refinement

- If an algorithm with a complex instruction block nested inside another block, make the inner block as a separate algorithm.
 - It is then possible to take a complex task and divide it up into simpler tasks, for example,
 - Task 1 - interact with the user (main)
 - Task 2 - solve the problem
- Then the former algorithm "calls" the latter algorithm.
- In this way, we can keep algorithms simpler, shorter and clearer.

Problem Decomposition

- The best way to develop algorithms for all but the simplest problems is by Problem Decomposition (also called Top Down Design).
- An algorithm for problem P is developed by these steps:
 1. Identify subproblems (P_1, P_2, \dots, P_n), simpler than P , whose results would be useful in solving P .
 2. Finalize the header information (givens, results, etc., header) for $P_1 \dots P_n$ but do not design their algorithms yet.
 3. Write the algorithm for P assuming algorithms exist for $P_1 \dots P_n$.
 4. Develop algorithms for $P_1 \dots P_n$.

Program with Two Algorithm Models

- Initially we develop programs with two algorithm models (i.e. two sub-programs)
 - The first, call it `main()`, is used to interact with the user
 - The second (with an appropriate name) is a problem solving algorithm
- Consider developing a program for the following problem: calculate the average out of 100 for 3 scores out of 25.
- Have already developed the problem solving algorithm
- For the `main()` algorithm, use a number of “pre-defined” algorithms for interacting with the user.

MarkResult Algorithm (problem solving)

Givens: score1, score2, score3 (scores out of 25)

Results: avgPct (average of scores, out of 100)

Intermediates: sum (sum of scores)

avgOutOf25 (average of scores, out of 25)

Header: avgPct \leftarrow markResult(score1, score2, score3)

Body: 1. sum \leftarrow score1 + score2 + score3

2. avgOutOf25 \leftarrow sum / 3

3. avgPct \leftarrow avgOutOf25 * 4

Exercise 3-1 - The main() Algorithm

- Our program execution always starts with this algorithm model (hence translated to the Java `main()` method)
- No *GIVENS* or *RESULTS* - i.e. no variables are defined to receive given values and no variable to return a result value.
- Use of intermediate variables only.
- Use the `println(...)` and `print(...)` algorithm models to print messages to the console
- Use the `readReal()`, `readInteger()`, `readLine()`, and `readBoolean()` to read values from the console.
- Basic logic:
 - Get input values from the user,
 - Call the problem solving algorithm to produce results,
 - Print results

Exercise 3-1 - Main Algorithm (interface to the user)



Givens:

Results:

Intermediates:

Header:

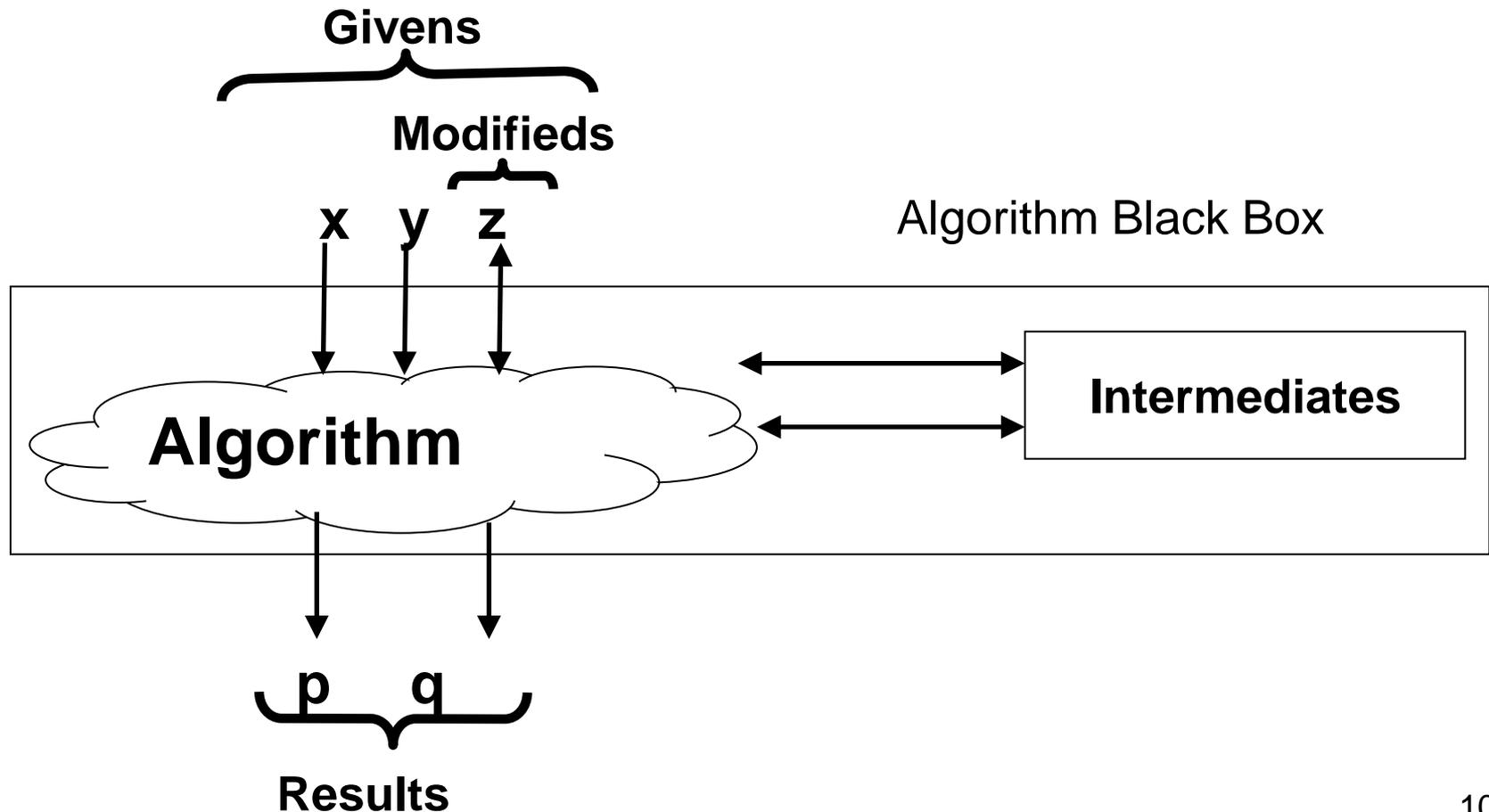
Body:

Modifying Arguments

- Some problems require the value of a variable in the argument list of a called algorithm to be changed.
 - The corresponding *GIVEN* variable is called a **MODIFIED** parameter and is listed among the **GIVENS** and put in the header like a *GIVEN*, but it is also described in a list of **MODIFIEDS** separate from normal **RESULTS**.
- Example: Suppose we want to write an algorithm that swaps the values of two variables .
 - e.g. given $x=7$ and $y=3$, the algorithm would exchange the values, resulting in $x=3$ and $y=7$.
- Shall come back to the concept later
 - Modifying parameters of simple types are not possible in Java
 - This can be done with Arrays and Objects using reference variable

Modified Parameters

$(p,q) \leftarrow \text{algorithm}(x,y,z)$



Augmented Algorithm Template

GIVENS:

RESULTS:

MODIFIEDS:

ASSUMPTIONS:

CONSTRAINTS:

INTERMEDIATES:

HEADER

(List of Results) ← Algorithm Name (List of Givens)

BODY

statement

statement

:

statement

} **Instruction block**

(comments added for clarification!)

(instructions can be numbered for tracing)

Exercise 3-2: Swap two values



GIVENS:

RESULTS:

MODIFIEDS:

INTERMEDIATES:

HEADER:

BODY:

Variables, a summary...

- The **GIVENS** are variables that receive values when at the start of the algorithm execution. Their values can differ from one call to another, i.e. for each instance of the problem resolution.
- The **RESULTS** are variables that contain the answers produced by the algorithm and used to pass back values to the calling algorithm.
- The **INTERMEDIATES** are other variables used during the execution of the algorithm.
- The **MODIFIEDS** are a sub-set (possibly empty) of the **GIVENS** whose corresponding arguments are modified by the executed algorithm.
- NOTE: For **GIVENS**, **RESULTS**, **INTERMEDIATES**, the variables exist in working memory only during the execution of the algorithm and do not exist between calls to the algorithm.

"When a programming language is created that allows programmers to program in simple English, it will be discovered that programmers cannot speak English." -- Anonymous

Translating Algorithms to Java

Translating Algorithm Models to a Java Program

- Our approach to programming is to first develop and test algorithms using a model and then **TRANSLATE** them into code in a programming language.
- Translating algorithms into code is very much a mechanical process, involving only a few decisions.
- Each algorithm model is translated to a Java Method.
- For the first half of the course, we shall develop programs that consists of a single Java class with two methods:
 - **main** that will interact with the user
 - Problem solving method to implement our problem solving algorithm model.

Translated to a Program (2 methods)

```
import java.util.Scanner;
class AverageScores
{
    /**
     * Method: main
     * Description: Interface with the user
     */
    public static void main (String[] args)
    {
        // Instructions } Instruction block
    }
    /**
     * Method: markResult
     * Description: Computes the average (percent)
     *               for three scores out of 25
     * Parameters (givens):
     *   score1, score2, score3   The three scores
     */
    public static double markResult(double score1,
                                    double score2,
                                    double score3 )
    {
        // instructions } Instruction block
    }
}
```

Translating Main Algorithm Model to a Java Method

- Intermediates all get translated to Java **VARIABLES**.
 - They all must be declared and given a type. Their descriptions are put in the program as comments (called the **DATA DICTIONARY**).
- Calls to `println(...)` and `print(...)` algorithm models are translated to appropriate calls to "**`System.out.println(...)`**" and "**`System.out.print(...)`**" Java methods.
- Calls to the `readReal()`, `readInteger()`, `readLine()`, and `readBoolean()` to translated to appropriate calls to methods in the **ITI1120** Class or **Scanner** Class.
- Basic logic:
 - Get input values from the user,
 - Call the problem solving algorithm to produce results,
 - Print results

Translating Statements to Java

- Assignment statement

- Model: $x \leftarrow \text{expression}$
- Java: **`x = expression;`**

- Expressions

$\text{sum} \leftarrow \text{score1} + \text{score2} + \text{score3}$
`sum = score1 + score2 + score3;`

$\text{hypotenuse} \leftarrow \sqrt{x^2 + y^2}$
`hypotenuse = Math.sqrt(Math.pow(x,2) + Math.pow(y,2));`

- Call statement

$\text{average} \leftarrow \text{markResult}(\text{first}, \text{second}, \text{third})$
`average = markResult(first, second, third);`

$\text{first} \leftarrow \text{readReal}()$
`first = keyboard.nextDouble();`

The CALL statement

Model: $x \leftarrow \text{avg3}(10, j, k)$

Java: `x = avg3(10, j, k);`

- Here, `avg3(10, j, k)` is a method call statement. The method call statement evaluates to the value returned from the method which is assigned to a variable
- The method call can also be used in any expression (of the right type); values are used to evaluate the expression.
 - Example:
`y = 10.2 * avg3(a, b, c) + avg3(p, q, -11)`
 - The use of multiple calls in a single expression is difficult to trace! To be avoided in this course.

When a CALL is made ...

1. Execution of the calling method is suspended.
 2. Memory is allocated for parameters and local variables of primitive types (**int, double, char, boolean**) in the called method.
 3. Initial values for parameters of primitive types are **COPIED** from the corresponding arguments of the call.
 4. Parameters of reference types are associated to the arrays or objects that the corresponding arguments refer to.
 5. Execution of method body begins.
- When the method body finishes, the return value is **COPIED** back to the calling method and the calling method resumes execution. All other values in the called method are forgotten.

Exercise 3-3 - Translating to Java main Algorithm to a Java Method



```
// PROGRAM Average--reads 3 scores and computes their average.
import java.util.Scanner;
class AverageScores
{
    public static void main (String[] args)
    {
        // SET UP KEYBOARD INPUT
        Scanner keyboard = new Scanner( System.in );
        // DECLARE VARIABLES/DATA DICTIONARY

        // READ IN Values from the user

        // Call to markResults

        // PRINT OUT RESULTS

    }
    // markResult method goes here
}
```

Translating Problem Solving Algorithm to a Java Method

- Givens, Results, and Intermediates all get translated to Java **VARIABLES**.
 - They all must be declared and given a type. Their descriptions are put in the program as comments (called the **DATA DICTIONARY**).
 - Initial values for Givens will be received from **main** via a call to the method.
- The result value will be returned to the main method for printing.
 - **IMPORTANT:**
 - In Java, the value of a variable is returned using the instruction "**return (aVariable) ;**".
 - This instruction terminates the execution of the method, and thus must be placed only at the end of the method (last instruction).
 - You are not allowed to place this instruction anywhere else in the method.
 - Although algorithm models allow for multiple results, the Java method can only return a single value
 - For now, shall develop algorithms that return only a single result

Exercise 3-3 - Translating to Java

Problem Solving Algorithm to a Java Method

?

```
// PROGRAM Average--reads 3 real numbers and computes their average.
```

```
class AverageScores
{
    // main method goes here
    // GIVENS:  score1, score2, score3  scores out of 25
    public static double markResult(double score1, double score2,
                                    double score3)
    {
        // DECLARE VARIABLES/DATA DICTIONARY
        // Intermediate variables

        // Result variable

        // BODY OF ALGORITHM

        // RETURN RESULTS
    }
}
```

"To understand a program,
you must become both the machine and the program."
- A. Perlis

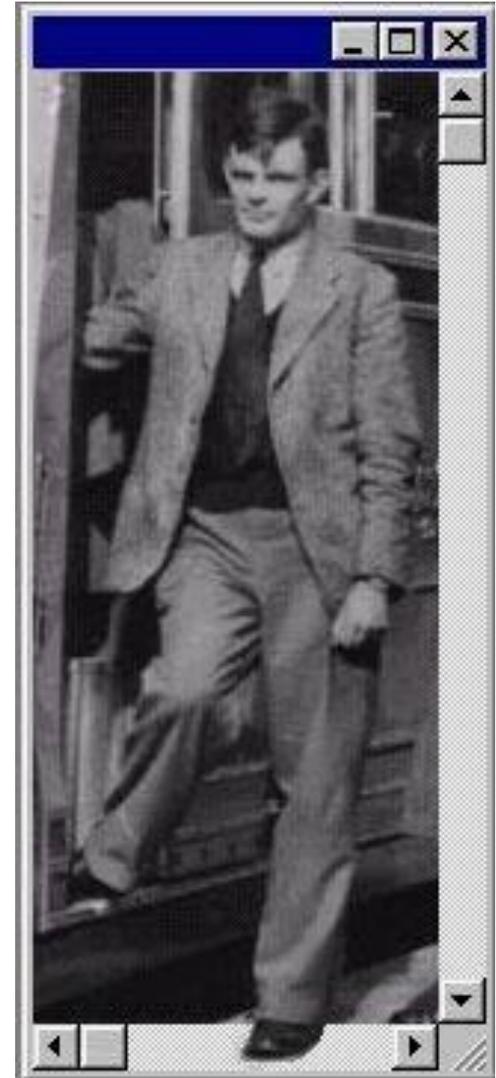
Section 4: Tracing and Debugging

Objectives:

- Tracing algorithms
- Debugging tools

Historical note ...

- [Alan M. Turing](#), British mathematician and father of modern computer science, designed, in 1936, a logic machine able to solve all the problems that can be formulated in algorithms for the modern computers: the [Turing machine](#).
- He also proposed the [Turing test](#), for artificial intelligence.
- The highest distinction in computer science now, awarded by ACM, bears his name: the [Turing Award](#).



Tracing an Algorithm

- To **TRACE** an algorithm is to execute it by hand, one statement at a time, keeping track of the value of each variable. The aim is either to see what results the algorithm produces or to locate “bugs” in the algorithm.
- Tracing always involves a single problem instance. You may need to do several traces (with **different** givens) of the same algorithm to find bugs.
 - Define test cases for your tracing, that is, expected results for a set of given values
 - You can re-use the test cases for testing your software

Tracing Steps

1. Number every statement in the algorithm model.
2. Prepare a programming model (use as many pages as required) so that you may "execute" the algorithm model.
3. Make a table. The first column will say which statement is being executed. The other columns each correspond to a variable. There should be one column for each given, result, and intermediate.
4. In the programming model, "create" the variables and insert initial values into the given variables and ? in all other variables. Fill in the first row of the table with the variables' initial values.
 - The givens will have the values supplied by the calling statement; all other variables will have a "?".
 - Write "initial values" in column 1 of row 1.

Tracing Steps (continued)

5. The second row is for the first statement executed. Put its number in column 1, and update the programming model by changing the variable values affected by the instruction. For every variable whose value is changed in the programming model put the new value in the variable's column. Leave the other columns blank.
 6. Proceed to the next statement, make a row for it, and continue until you reach the end of the algorithm.
- Note: The values across a row represent the "state" of the system at a particular point in time.

Exercise 4-1 - Tracing Example



avgPct ← markResult(18, 23, 19)

Tracing a Call

- When tracing an algorithm, every time it calls another algorithm whose body is known you must produce a trace of the called algorithm on the givens provided by the call. The trace for each call of each algorithm should be done on a separate page. Each trace should say where it was invoked ("called from page X").
- When the statement being executed is a call you put more in column 1 of the trace table than just the statement number - you say where to find the trace of the called algorithm ("see page Y"), and you put in a complete picture of the information passing. Just as on the previous slide, write the call statement directly above the header of the algorithm being called and draw arrows showing how the information is passed.
- Values (results, modifieds) that are passed back when the called algorithm terminates should be written in the columns for the variables in the results part of the call.

Exercise 4-2 -Tracing a Call

- Redo the “marks out of 100” problem, but trace the main algorithm model for the following test case (shows interaction with the user):

Please enter three scores out of 25

23 16 21

The average is 80 percent

Exercise 4-2 - Tracing Main (page 1) ?

- Trace: `main`

Exercise 4-2 - Tracing MarkResult (page 2)

Exercise 4-3 - Marks out of 100, again



-
- Redo the “marks out of 100” problem, but this time use the algorithm developed for the “average” problem: $avg \leftarrow \text{average}(\text{num1}, \text{num2}, \text{num3})$

GIVENS:

RESULTS:

INTERMEDIATES:

HEADER:

BODY:

Exercise 4-4 - Tracing Example (page 1) ?

- Trace: `avgPct ← markResult(23, 16, 21)`

Exercise 4-4 - Tracing Example (page 2)

Exercise 4-5 - Reverse Digits



- Given a 2-digit positive number, N , reverse its digits to obtain a new number, $reverseN$.
- Assume there is available an algorithm with the header

$(high, low) \leftarrow digits(x)$

which returns the left (high) and right (low) digits of a given 2-digit number x .

GIVENS:

RESULTS:

INTERMEDIATES:

HEADER

BODY

Exercise 4-6 - Trace Reverse Digits

Exercise 4-7 - Join Four Numbers



- Write an algorithm that takes 4 positive integers and joins them into one,
 - e.g., given 11, 35, 200, and 7 it should produce 11352007.
- Trace your algorithm on these givens.
- You may assume there is available an algorithm:

$c \leftarrow \text{join}(a, b)$

Givens: a, b, two positive integers
Result: c is the number having the digits
 in a followed by the digits in b.

- Example: $\text{join}(120, 43)$ produces 12043

Tracing Java Programs

- When you are tracing an actual program, a useful tool is a “debugger”. This tool can perform a trace on a running program.
 - Often this requires telling the compiler to add extra information to help the debugger.
- Two modes of operations:
 - Run up to a **breakpoint**: the program will run at full speed up to a pre-defined point in the source code, and then stop.
 - “Single step” mode: the program will execute one statement and stop.
- When the program stops, you can inspect the current values of variables
 - You can check if the variables’ values correspond to the equivalent row of a manual trace.

Using a debugger

- Many debuggers come with four options for execution of a stopped program.
 - **Step Into**: If the next statement involves a call to another algorithm, execution will go to the first statement of the algorithm body and stop.
 - **Step Over**: If the next statement involves a call to another algorithm, the other algorithm will be completely executed, and debugger will stop at the next line in the current algorithm.
 - **Step Out**: Execute all statements up to the end of the current algorithm.
 - **Resume**: Begin normal execution from the next statement.

"When you come to a fork in the road, take it."
- Y. Berra

Section 5: Branching

Objectives:

- Structure Charts and Flow Charts
- Branching Instructions and Tracing
- Translating Branches to Java
- Complex Boolean Expressions

Historical note ...

- 1945: An insect in the circuits blocked the computer Mark I. The computer scientist Grace Murray Hopper decided to call any program malfunction « bug »!
- 1951: Invented the first compiler (AO) for generating machine code from a program source code.
- She was one of the main creators of one of the first programming languages: COBOL.



9/2 Ph 5

9/9

0800 Action 847 025
1000 " " 1057 846 895 check

1300 (032) MP-MC 1.98230000
2.130476415 (03) 4.615925059(-2)
(033) PRO 2 2.130476415
check 2.130676415

Relays 6-2 in 033 failed special speed test
in relay 11,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

1630/1630 Antenna started.
1700 closed down.

First actual case of bug being found.

Relay 3145
Relay 3370

Branching Control Instruction

- So far in the bodies of our algorithms, we have used:
 - a simple statement
 - a straight sequence of simple statements
- Sometimes, we need more than a straight sequence in our solutions, as we sometimes need to do different computations depending on certain conditions.
- **Branching** instruction (condition)!

Problem: Larger of Two Numbers

- Write an algorithm to compute the larger of two given numbers.

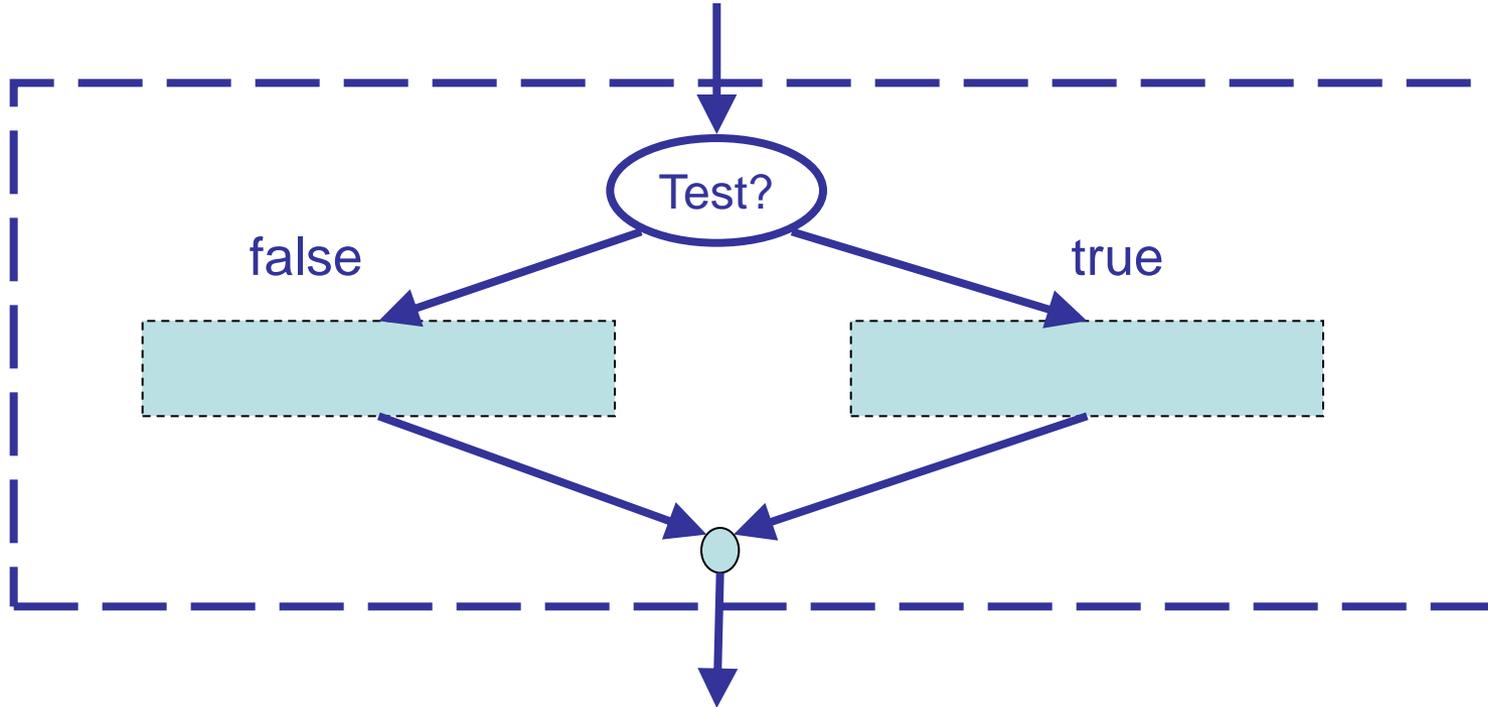
GIVENS: x, y (two numbers)

RESULT: m (the larger of x and y)

HEADER: $m \leftarrow \max2(x, y)$

- We will represent this with a Branching Control Instruction

Branching Control Instruction



- The shaded boxes are **INSTRUCTION BLOCKS**.
- Note that the branch instruction is complex and can contain many other instructions within its instruction blocks.
 - Graphical representation in our software models!

Algorithm Model Diagrams

- Provides visual description of algorithms.
- Composed of nodes connected with arrows.
- Test node:
 - Represents the testing of a condition (Boolean expression with question mark)



- Instruction block :
 - Indicates where another instruction block can be inserted. The instruction block can contain simple or complex instructions (and even no instructions at all)



Contents of Instruction Blocks

- Simple instruction (call, assignment)
- Empty statement (\emptyset = "do nothing")
- Branching control instruction
- Loop control instruction (coming soon...)
- **Important:** Each block has exactly one entrance (one arrow in) and one exit (one arrow out).

Exercise 5-1 - Back to the Larger of Two Numbers

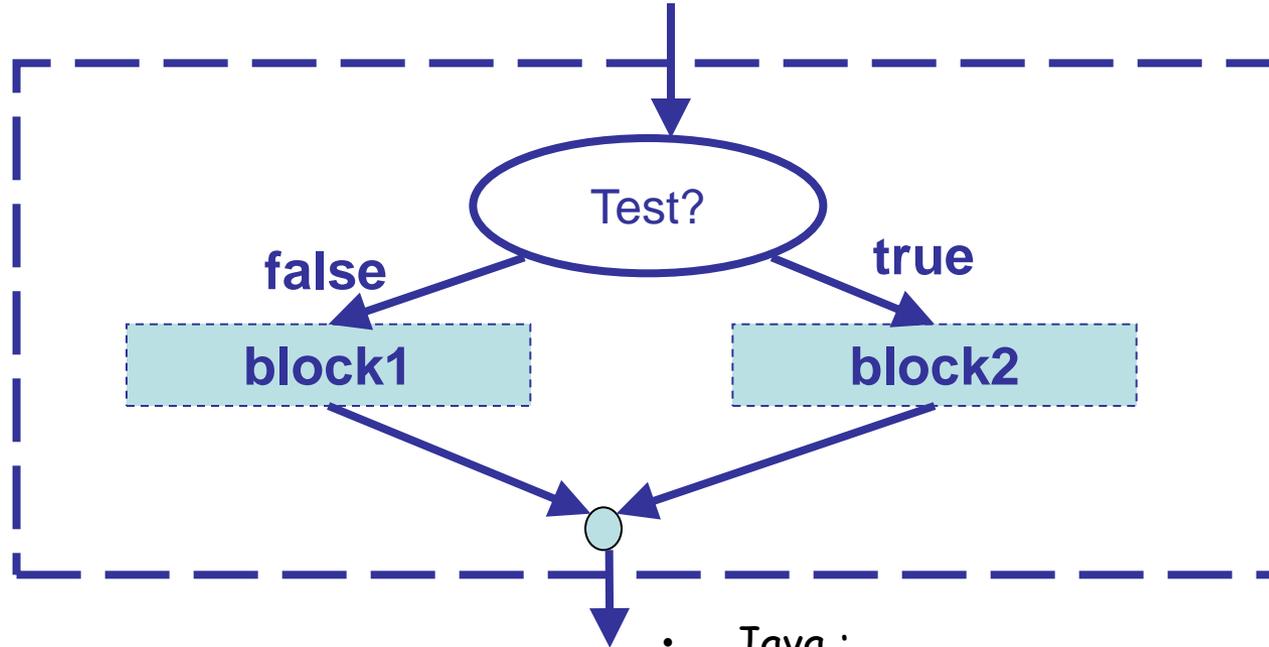


-
- How would the algorithm for finding the larger (`max`) of two values `x` and `y` be written in a model diagram ?

Exercise 5-2 - Maximum of 3 numbers ?

- Given three numbers x , y , and z , find the maximum of the three values.
 - Version 1: nested tests
 - Version 2: sequence of tests

Translating Branches to Java



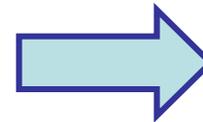
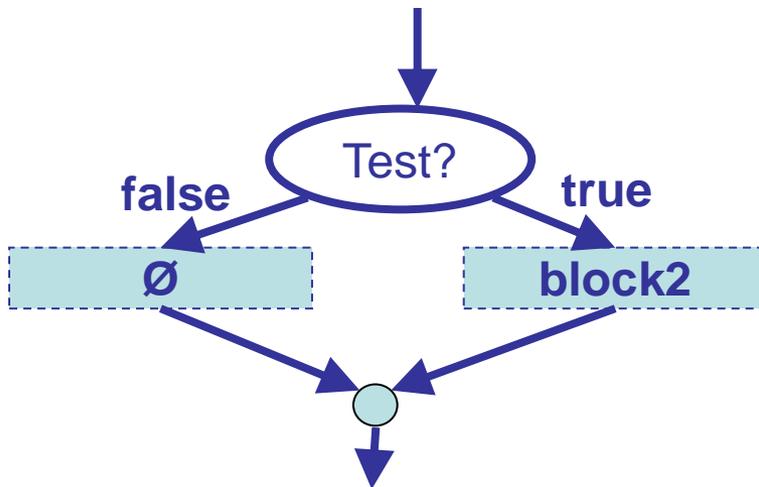
• Java :

```
Instruction block 2 { if (test)
                      { // Instructions
                      }
Instruction block 1 { else
                      { // Instructions
                      }
```

Java if Instruction - Option

The else part of the instruction is optional. Thus in algorithms use only the empty statement block in the false part of the branch statement.

```
if (test)
{
    // Instructions
}
else
{
    // Instructions
}
```



```
if (test)
{
    // Instructions
}
```

Exercise 5-3 - Translating Branches



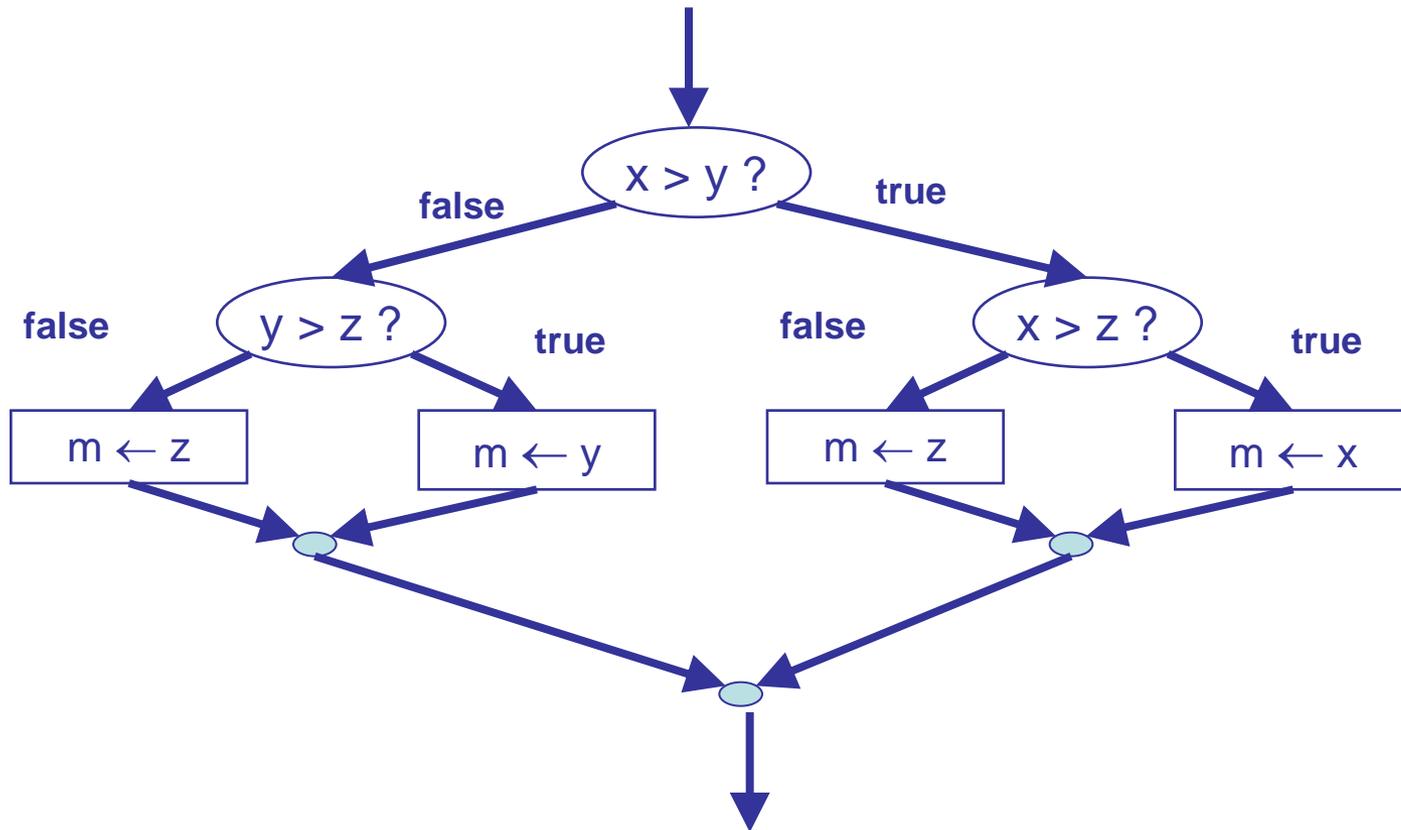
Givens: x, y, z (three numbers)

Result: m (the largest given value)

Header: $m \leftarrow \max3(x, y, z)$

- Two solutions:
 - sequence of branch instructions
 - nested branch instructions
- Translate the latter into Java:

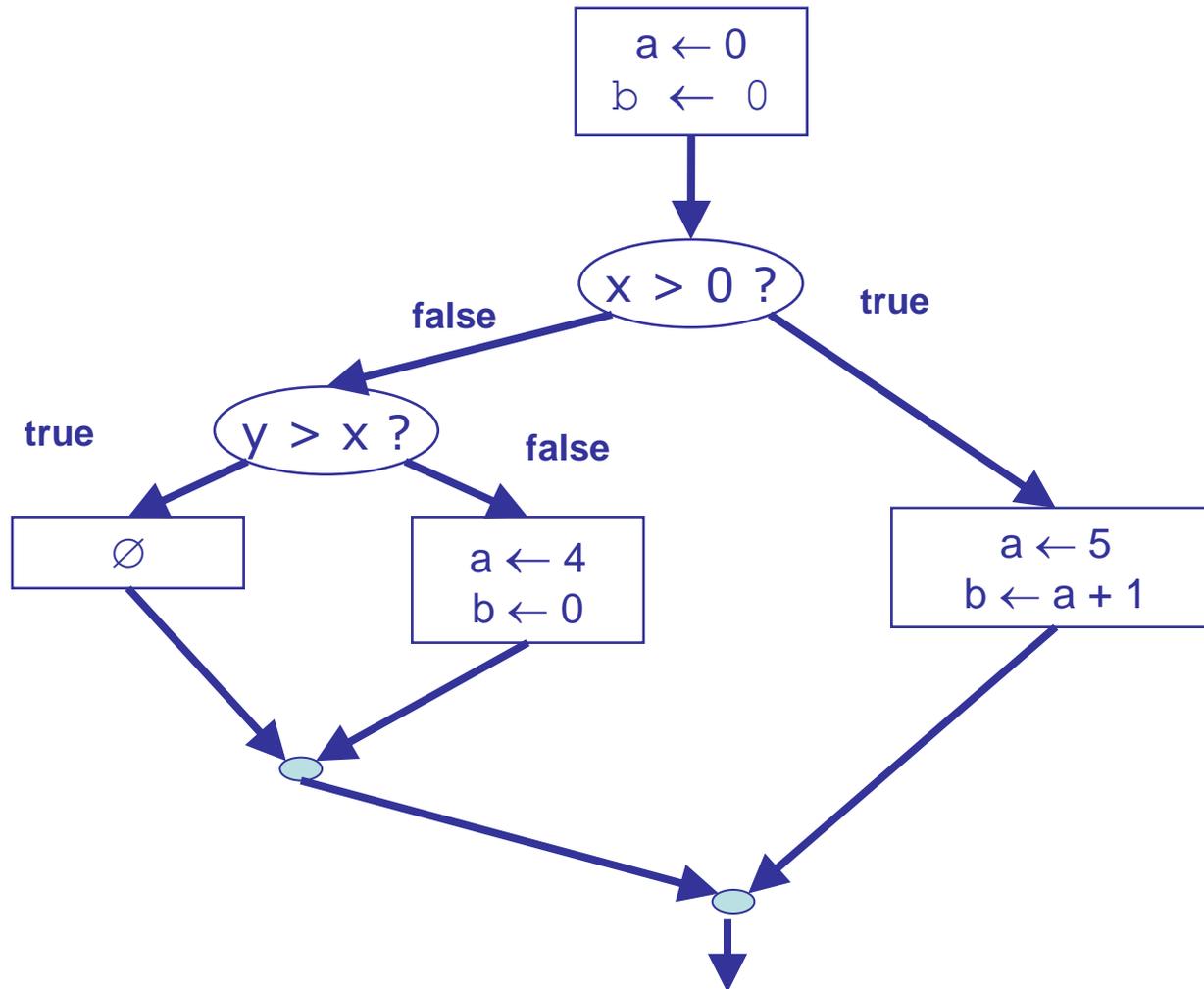
Nested Branches



Exercise 5-4 - Translation of Nested Branches



Example of an instruction block with a branch instruction



Exercise 5-5 - Translation of an Instruction Block



Tracing algorithms with branching

- When tracing an algorithm with tests (branches or loops)
 - Number the tests as well as the statements
 - Include a row in the trace indicating which test is being done and whether it is true or false.
 - Indicate only the instructions executed in the trace.

Exercise 5-6 - Trace of Maximum of 3 numbers



Trace: $\text{max3}(5, 11, 8)$

Exercise 5-7 - Movie Tickets



-
- Calculate the amount to charge for a person's movie ticket given that the charge is \$7 for a person 16 or under, \$5 for a person 65 or older, and \$10 for anyone else.
 - Version 1: nested tests
 - Version 2: sequence of tests

Boolean Variables

- A **Boolean variable** is one which can have only 2 possible values: **TRUE** or **FALSE**.
 - In reality represented by two values (e.g. 0 and 1), but in high level language use only these key words are allowed!
- An assignment statement is used to put a value into a Boolean variable, e.g.,
x ← TRUE
y ← FALSE
- The outcome of a test (Boolean expression) can be assigned to a Boolean variable:
x ← (a < 0)

Exercise 5-8 - Positive Value



-
- Write an algorithm which checks if a given number x is positive.

Compound Boolean Expressions

- A compound Boolean expression consists of two or more Boolean expressions connected by operators AND and/or OR.
- **Exercise 5-9** - Write a compound Boolean expression that is true if a given age is between 16 and 65 (not including 16 or 65) and false otherwise.



Truth Tables



- A **TRUTH TABLE** for a compound Boolean expression shows the results for all possible combinations of the simple expressions:

x	y	x AND y	x OR y
TRUE	TRUE		
TRUE	FALSE		
FALSE	TRUE		
FALSE	FALSE		

Operator NOT

x	NOT x
TRUE	FALSE
FALSE	TRUE

- NOT is an operator to negate the value of a simple or compound Boolean expression:
- Example. Suppose age = 15. Then:
 - Expression age > 16 has a value FALSE, and NOT (age > 16) has a value TRUE.
 - Expression age < 65 has a value TRUE, and NOT (age < 65) has a value FALSE.

Exercise 5-10

More Compound Boolean Expressions



Suppose $x = 5$ and $y = 10$.

Expression	Value
$(x > 0) \text{ AND } (\text{NOT } (y = 0))$	
$(x > 0) \text{ AND } ((x < y) \text{ OR } (y = 0))$	
$(\text{NOT } (x > 0)) \text{ OR } ((x < y) \text{ AND } (y = 0))$	
$\text{NOT } ((x > 0) \text{ OR } ((x < y) \text{ AND } (y = 0)))$	

Expressions in Tests

- The TEST in a Branch or Loop may be any Boolean expression:
 - Boolean variable
 - Negation of a Boolean expression
 - NOT (Java: **!**)
 - Comparison between two values
 - Java operators: **==** **!=** **<** **>** **<=** **>=**
 - The **data** being compared may not necessarily be **boolean**, but the **result** of the comparison is **boolean**
 - Join two Boolean expressions
 - AND (Java: **&&**)
 - OR (Java: **||**)
- Watch out for
 - confusing **=** with **==**
 - confusing AND with OR
- e.g. test if x is in the range 12..20:
(x >= 12) && (x <= 20)



"A program without a loop ... isn't worth writing."
-- A. Perlis

Section 6: Loops and Arrays

Objectives:

- Loops
- Arrays
- Translating to Java
- Tracing Arrays and Loops
- Strings in Java
- Many Examples!

Historical note ...

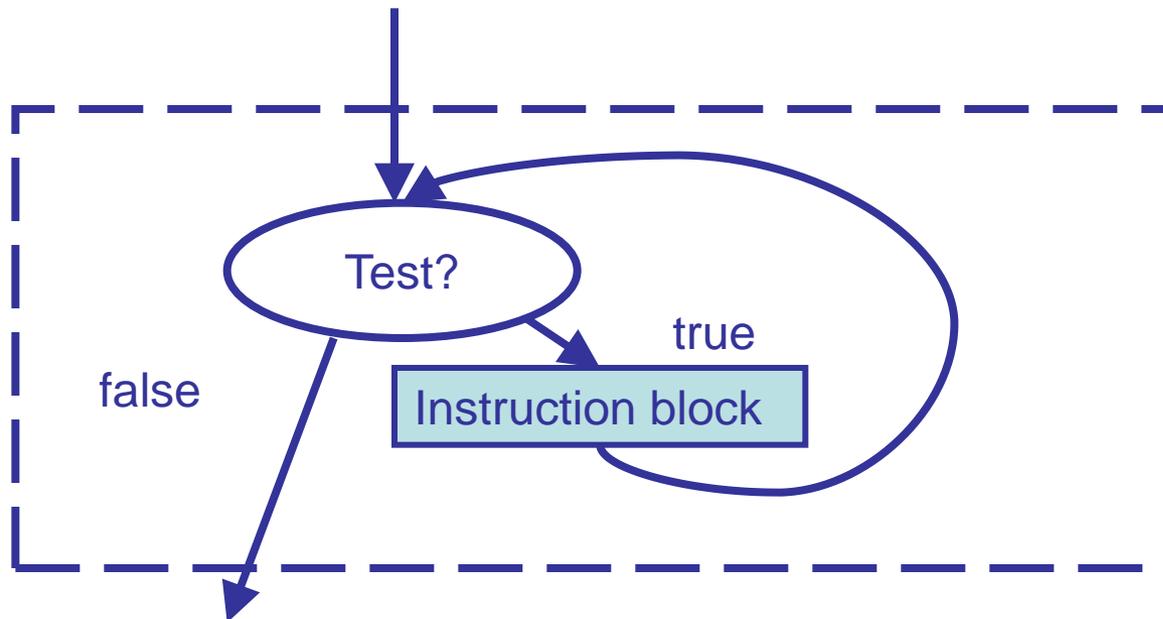
- Donald Knuth, American computer scientist, a pioneer of the domain of algorithm analysis.
- He is the author of the very respected book *The Art of Computer Programming* and of the scientific text editor TeX.



- Edsger Dijkstra, computer scientist from Netherlands, developed the algorithm of the shortest path (that bears his name) and the concept of sentinel for programming and for parallel processing.
- His 1968 article, "Go To Statement Considered Harmful" revolutionized the utilization of the instruction GOTO to the profit of the control structures such as the while loop.

The Loop Instruction

- Sometimes we need to repeat an instruction block. In our algorithm diagrams, we use a **loop instruction**:



- The instruction block inside the loop is repeated over and over until the test becomes false.

Designing a loop instruction

1. Initialization
 - Are there any variables to initialize?
 - These variables will be updated in the loop.
2. Test condition
 - A condition to determine whether or not to repeat the loop instruction block
3. Loop instruction block
 - What are the steps to repeat?
 - Definite loop: know a-priori how many times the loop is repeated - usually involves a counter variable
 - Indefinite loop: the number of times the loop will be repeated is unknown - usually involves a flag variable

Exercise 6-1: Sum from 1 to N



GIVENS:

INTEMEDIATES:

RESULTS:

HEADER:

BODY:

Exercise 6-1: Trace of `sum1toN(3)`

Instructions	N	Count	Sum

Exercise 6-2: A "definite" loop



-
- Write an algorithm to find the factorial of a number N , denoted as $N!$
 - Definition of factorial (product of 1, 2, 3 until N):

$$N! = 1 \times 2 \times \dots \times N$$

Exercise 6-2: A "definite" loop

GIVENS:

RESULTS:

INTERMEDIATES:

ASSUMPTIONS:

HEADER:

BODY:

Exercise 6-3: An "indefinite" loop



-
- Write an algorithm to determine how many times an integer `aNumber` can be divided by another integer `divisor`, until the result is less than `divisor`.
 - This is the integer part of the logarithm function.

Exercise 6-3: An "indefinite" loop

GIVENS:

RESULTS:

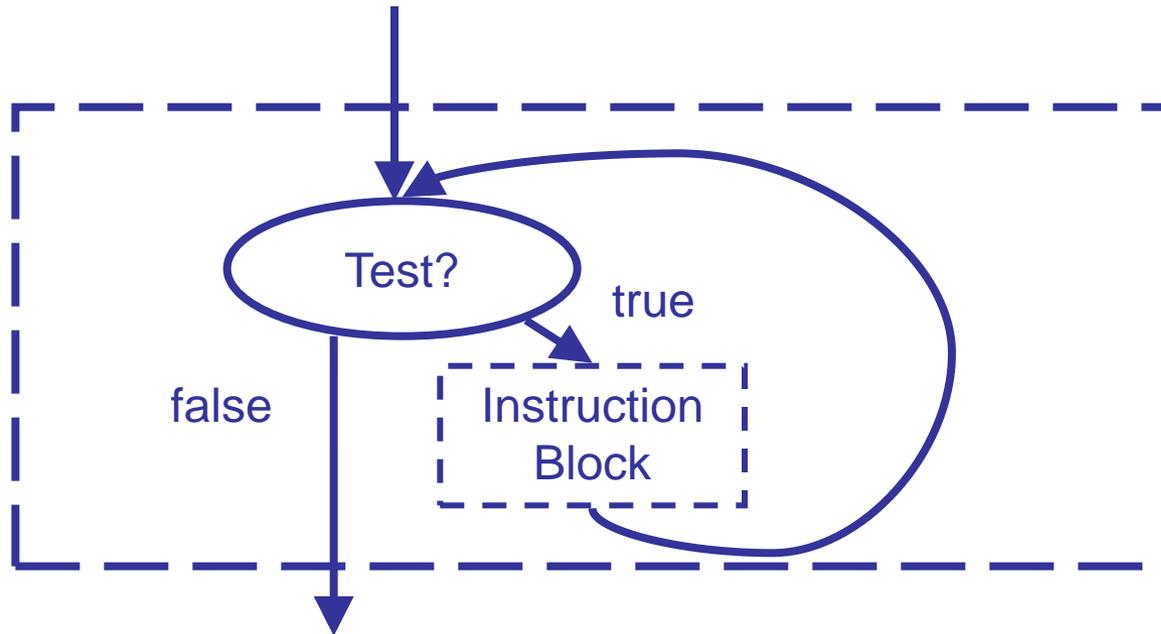
INTERMEDIATES:

ASSUMPTIONS:

HEADER:

BODY:

Translating Loops to Java



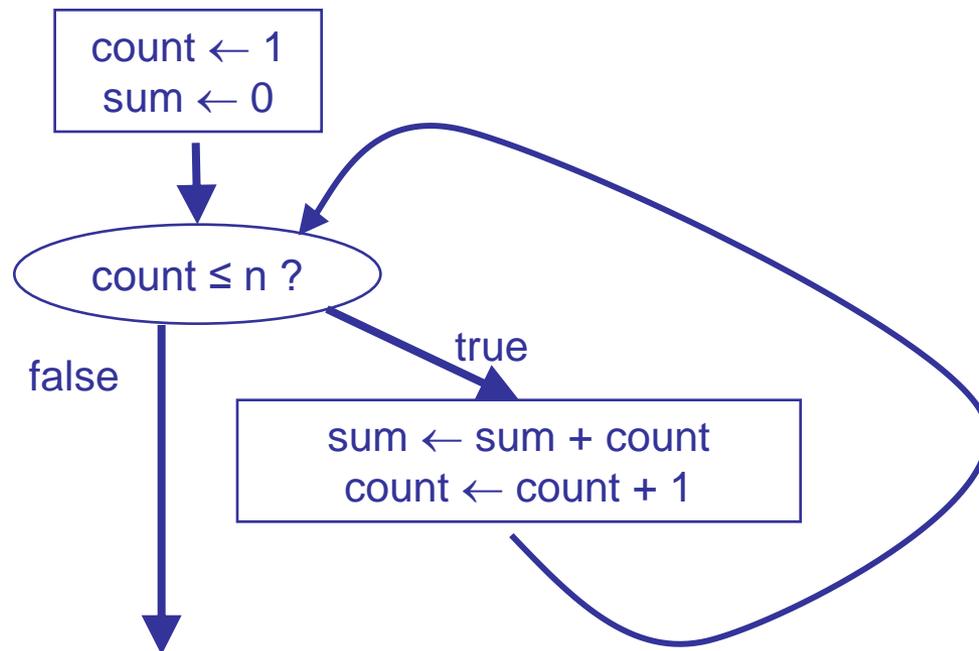
- Java

```
while (Test)
{
    // Instructions
}
```

Instruction block {

Algorithm: Sum from 1 to N

GIVEN: n (a positive integer)
INTERMEDIATE: count (index going from 1 to n)
RESULT: sum (sum of integers 1 to n)
HEADER: sum \leftarrow sum1ToN(n)
BODY:



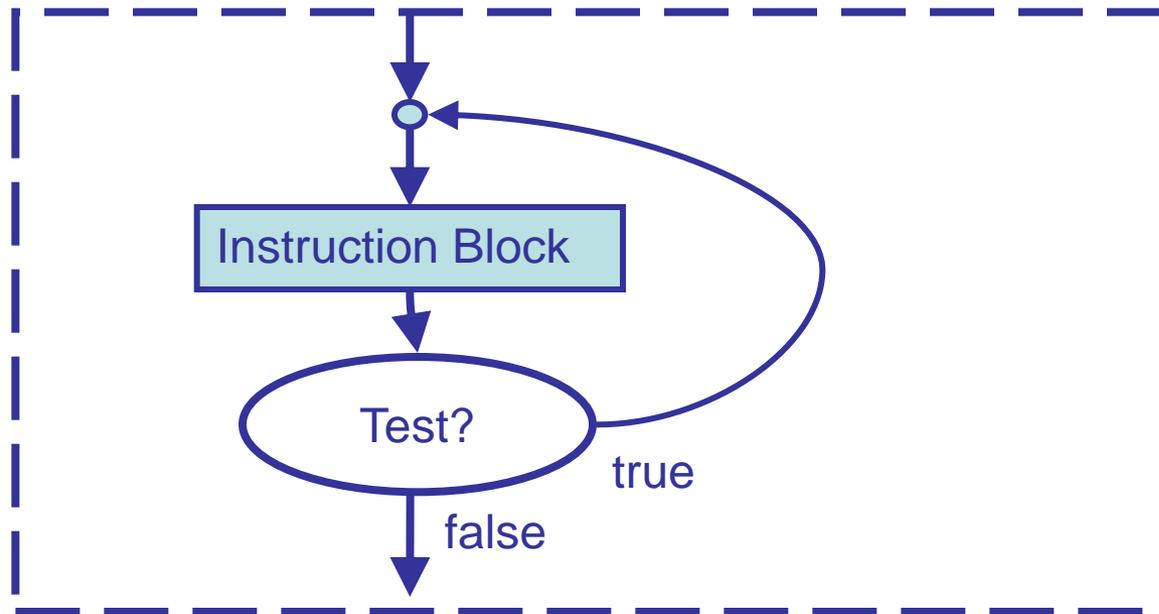
Exercise 6-4: Translate Loop to Java ?

```
import java.io.*;

class Sum1ToN
{
    public static int sum1ToN (int n)
    {
        
    }
}
```

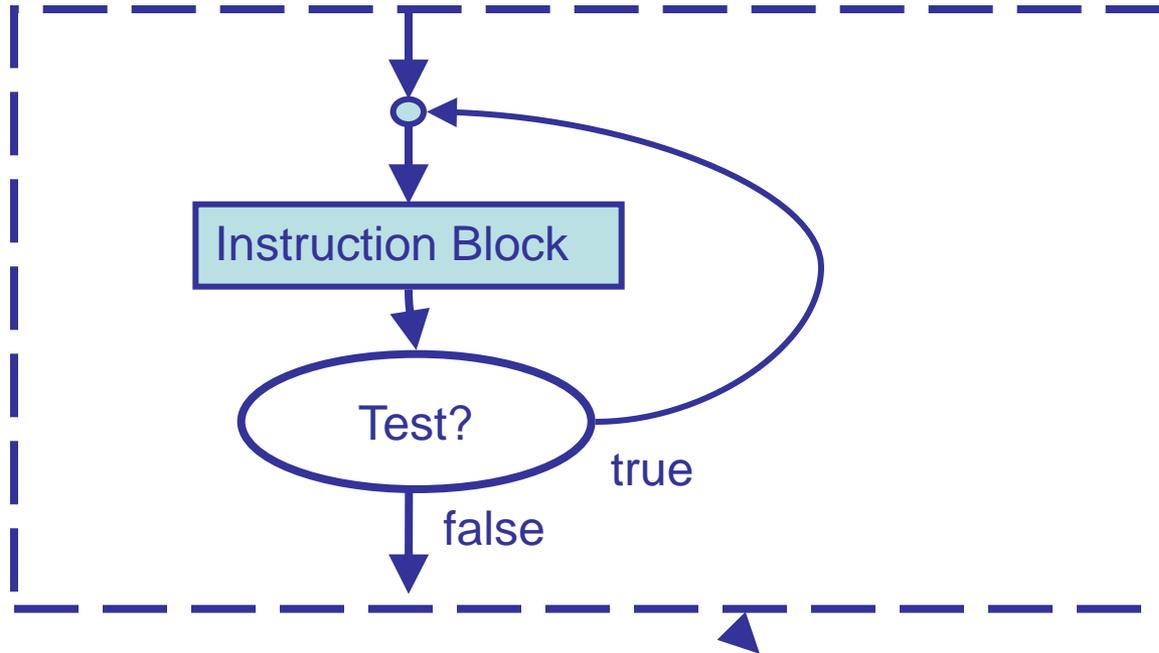
The Post Test Loop Instruction

- Test after the execution of the instruction block gives the **post test loop instruction**:



- The loop instruction block is executed at least once and is repeated over and over until the test becomes false.

The Post Test Loop Instruction - Translating to Java



• Java:

```
Instruction block {  
    do  
    {  
        // Instructions  
    }  
    while(test);
```

Exercise 6-5: Example of Post-Test Loop

- Use a post-test loop to develop a « main » algorithm for computing factorial and translate to Java

GIVENS:

RESULTS:

INTERMEDIATES:

CONSTRAINTS:

HEADER:

BODY:

Exercise 6-5: Translation to Java

```
public static void main(String args)
{
    // Variables

    // Body

}
```

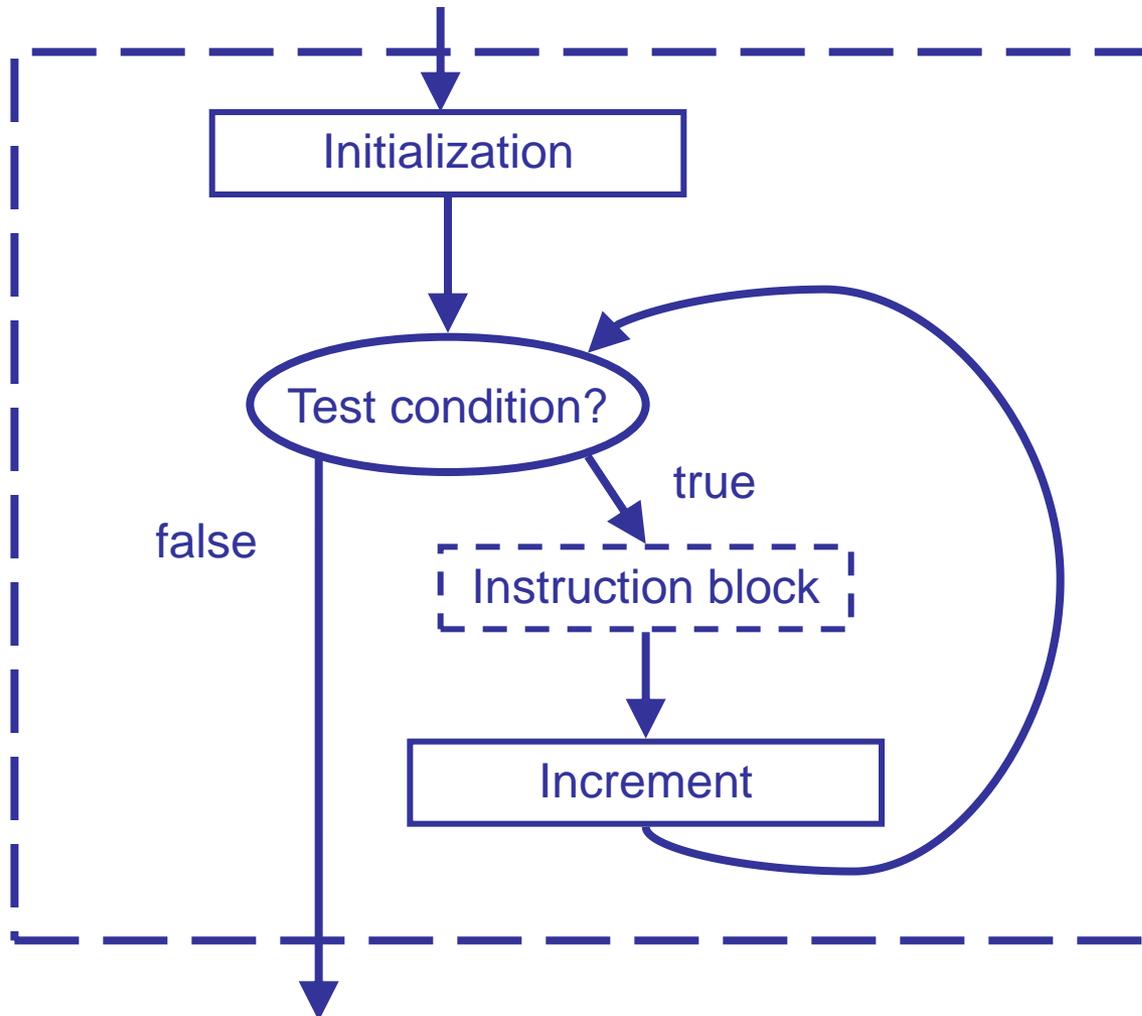
The FOR Loop

- Java provides another format of a loop, which is usually used when we know how many times the loop body is to be executed (definite loop).
- The FOR loop has the following format:

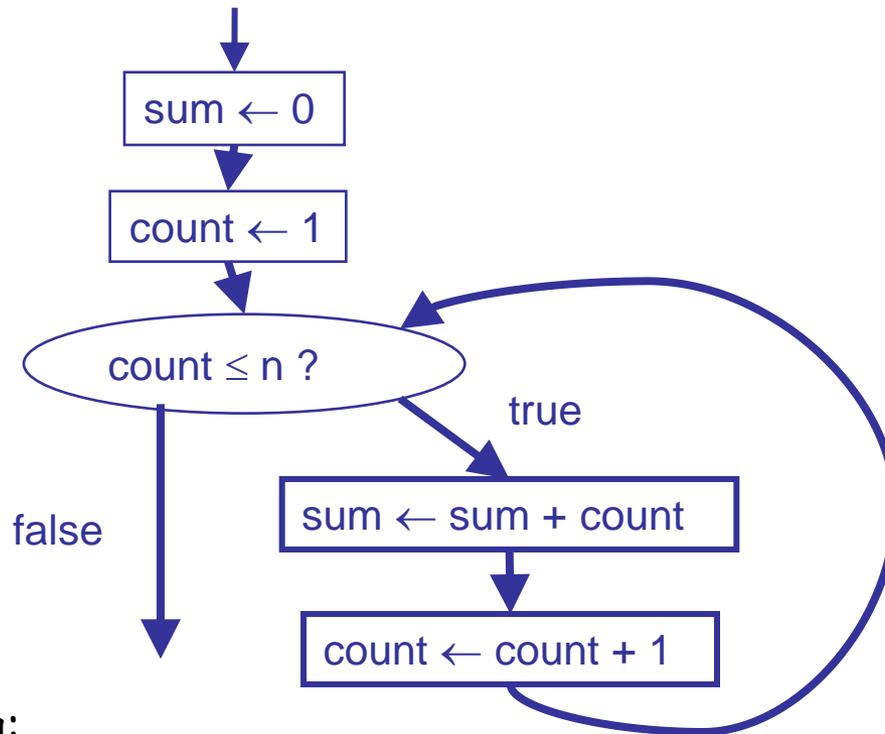
```
for (<initialization>; <test_condition>; <increment>)  
{  
    // instructions  
}
```

- In most cases, the initialization part initializes a counter, the test condition tests if the counter is within the limit, and the increment part modifies the counter.
- Any FOR loop can always be formed as a WHILE loop
 - It does not give us any extra capability.
 - However, the notation is often more convenient.

The FOR loop diagram



Exercise 6-6: FOR loop to add 1 to N



- Translate to Java:

A Problem with Simple Variables...

- Suppose that an algorithm reads 5 integers and displays them in reverse order:

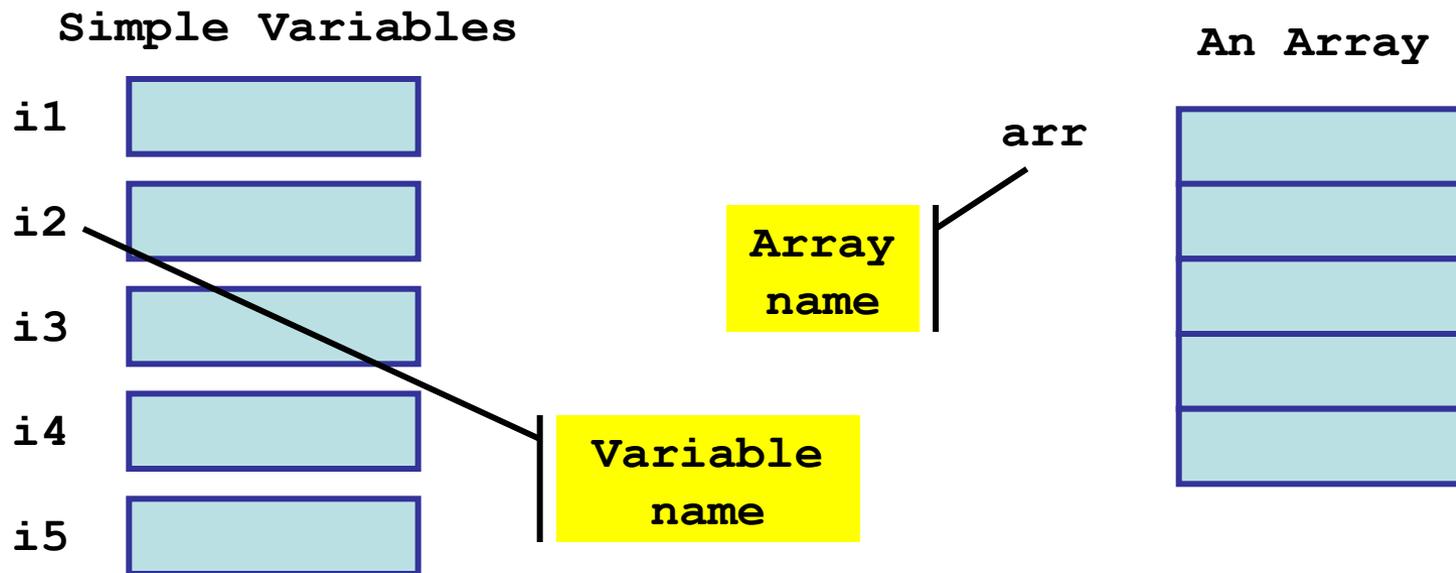
Body:

```
i1 ← readInteger()
i2 ← readInteger()
i3 ← readInteger()
i4 ← readInteger()
i5 ← readInteger()
printLine(i5)
printLine(i4)
printLine(i3)
printLine(i2)
printLine(i1)
```

- What happens with 1000 integers? X integers?

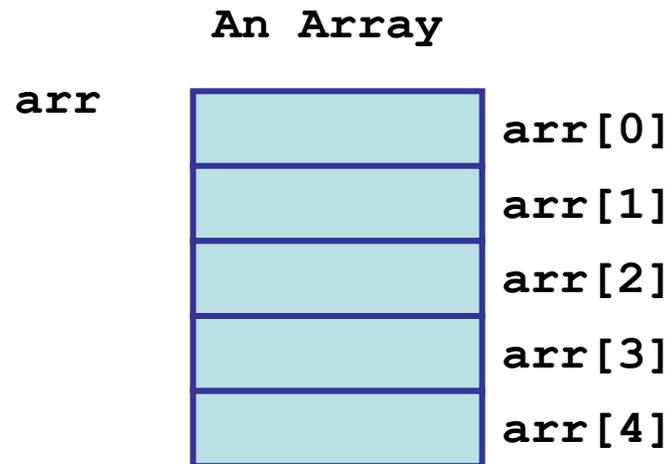
Computer Arrays

- "Simple" variables contain one value.
- An array has many locations, each able to contain one value.
- An array is essentially a collection of variables of the same type



Computer Arrays (continued)

- If array `arr` has 5 positions, we refer to them using the integers 0-4, called indices or subscripts.
 - e.g. `arr[2]` is the **THIRD** position with index 2.
 - Note that `arr[2]` is equivalent to a variable name and can be used anywhere a variable name is used, e.g. in expressions.

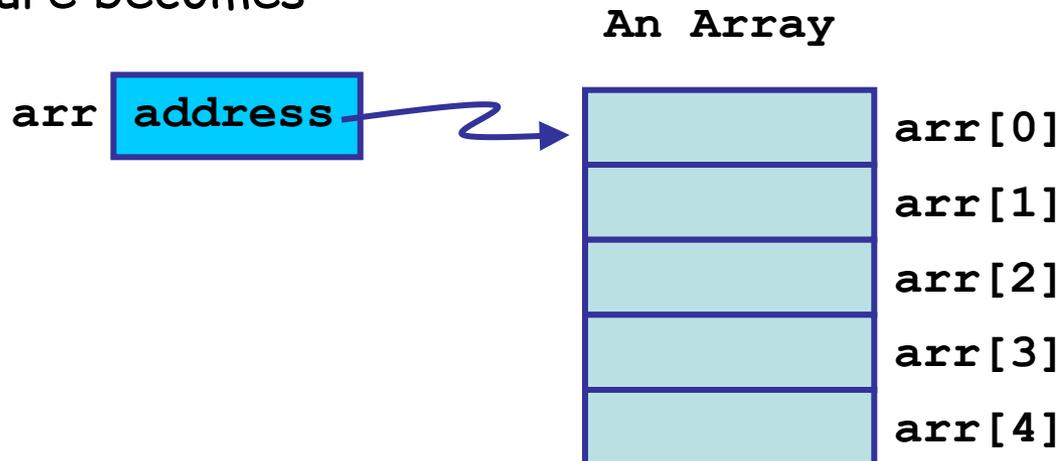


Computer Arrays (continued)

- Two pieces of information may be required when dealing with arrays
 - The first is the size of the array, that is the number of positions available in the array
 - The second is the number of positions that contains "values", that is the number of positions that have been initialized.
 - When we pass arrays to algorithms, we may pass one or both these values
 - E.g. if values are in the first 3 positions (of 5 positions available), we might pass 3 to a GIVEN **aLength** in an algorithm that wishes to process the known values.
 - E.g. if we want an algorithm to see all 5 positions then the GIVEN **aLength** would receive 5.
 - What would be valid indexes that can be used with the array?

Array Name

- What does the array name represent?
- It can represent the address where the array is located in memory
 - This is similar to a variable name, but is often treated differently to a variable name
 - Approach used in languages like C
 - Pictures in previous slides show this representation
- The array name can also be the name of a reference variable - the picture becomes

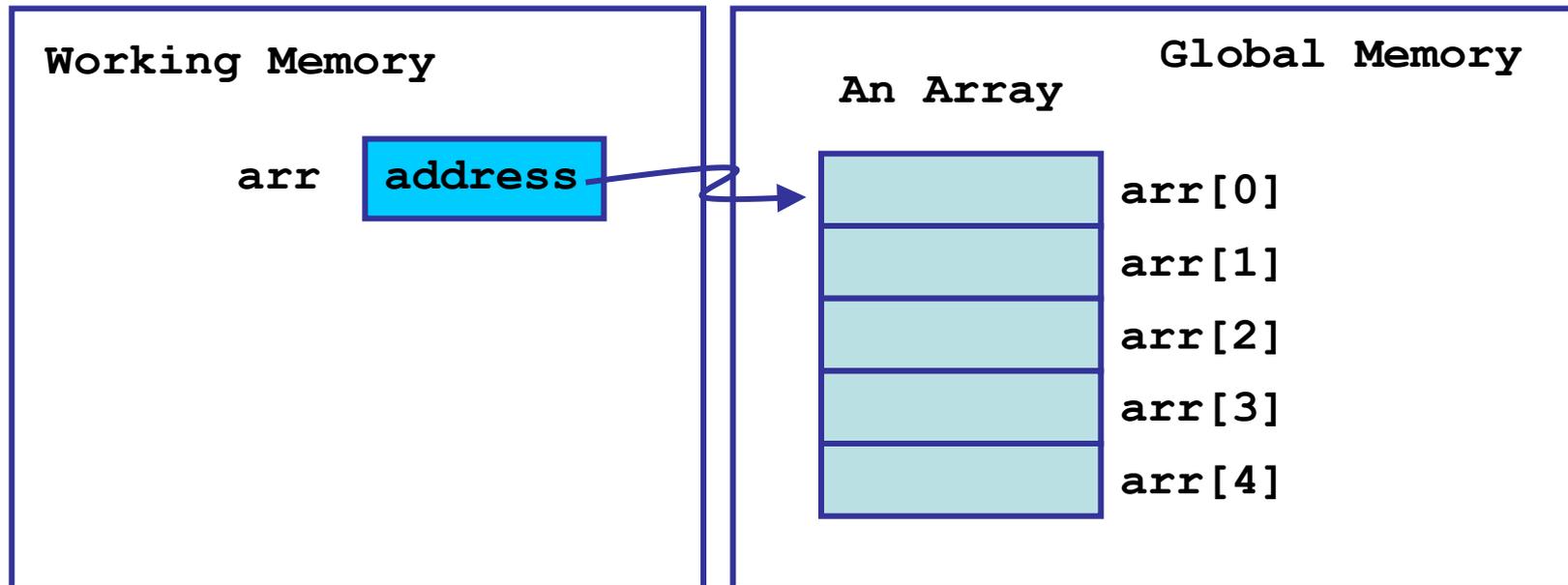


- This is how we shall model arrays.

Creating Arrays

- The following “standard” algorithm is used to create arrays
`anArrayRefVar ← makeNewArray(l)`
 - creates an array of `l` positions with unknown values in them.
 - `anArrayRefVar` is a reference variable to which is assigned the address to the array.

E.g. `arr ← makeNewArray(5)`



Exercise 6-7: Array Indexing



- The index (subscript) of an array of length l may be any integer expression that returns a value in the range $0 \dots (l-1)$.

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."
-- Stan Kelly-Bootle

- Suppose $k = 2$, and a references



$a[2] =$

$a[k] =$

$a[2*k-1] =$

$a[a[0]+1] =$

- $a[\text{expression}]$ is just like any ordinary variable and can be used anywhere an ordinary variable can be used.
- Remember a is a reference variable

Exercise 6-8: Value in Middle of an Array

- Write an algorithm that returns the value in the middle of an array A containing N numbers, where N is odd.
- Note that that the *GIVEN* a receives the reference (address) to the array.

Exercise 6-9: Swap Values in an Array ?

- Write an algorithm that swaps the values in positions I and J of array A.
- This possible because the address is passed to the algorithm (subprogram)

Exercise 6-10: Creating an Array



-
- Create an array containing the integers 1 to N in reverse order.
 - Remember the standard algorithm
 `anArrayRefVar ← makeNewArray(L)`
that creates an array referenced by `anArrayRefVar`; the array has L positions with **unknown** values in them.

Exercise 6-10: Trace for N=3



statements	n	index	a
initial values			

Loop and Array Exercises

6-1 Find the sum of the numbers $1\dots n$ ($1+2+\dots+n$).

? 6-11 Find the sum of the values in an array containing N values.

6-12 Given a value t and an array a containing n values, check if the sum of a 's values exceeds t .

? a) Use algorithm from Exercise 6-11.

? b) Efficient version which exits as soon as the sum exceeds t .

6-13 Count how many times j occurs in an array containing n values.

?

6-14 Given an array a of n values and a number k , see if k occurs in a or not.

?

a) Use algorithm from Example 6-13.

b) Efficient version which exits as soon as k is found.

?

More Loop and Array Exercises

? 6-15 Given an array a of n values and a number k , find the position of the first occurrence of k . (If k does not occur, return -1 as the position.)

? 6-16 Find the maximum value in an array containing n values.

6-17 Find the position of the first occurrence of the maximum value in an array containing n values.

? a) Use algorithm from Exercise 6-16.

? b) Use algorithms from any examples.

? c) Version using one loop and no other algorithms.

6-18 Check if an array of n values contains any duplicates.

? - Strategies?

Arrays in Java

- An array reference variable is declared with the type of the members.
 - For instance, the following is a declaration of a variable of an array with members of the type double:
`double[] anArray;`
- When an array reference variable is declared, the array is **NOT** created. What we have is a reference variable that can point to an array.
 - `anArray` will contain the special value `null` until it is assigned a valid reference.

Creating an array

- How do we translate the makeNewArray algorithm?
- To create the array in Java, operator **new** is used.
- We must provide the number of members in the array, and the type of the members: e.g. **new double[5]**
 - The number of members cannot be changed later.
 - The **new** operator returns a reference (address) that can be assigned to a reference variable, for example:

```
double[] anArray;  
anArray = new double[5];
```
- Note: Creating an array initializes all elements to zeros (which translates to **0**, **null**, **'\0'** according to the type of the array)
- When an array is created, the number of positions available in the array can be accessed using a field called **length** with the dot operator. For instance, **anArray.length** has a value 5.
- Arrays are created in "global memory"
- Reference variables are created in working memory.

Memory for Arrays

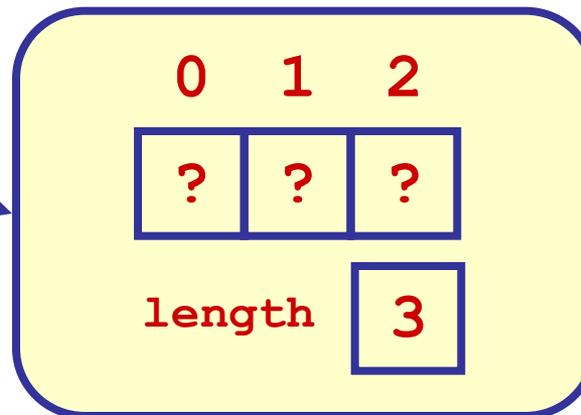
```
double[] anArray ;
```

anArray: null

"Thou shalt not follow
the NULL pointer,
for chaos and madness
await thee at its end."
- H. Spencer

```
anArray = new double[3] ;
```

anArray: address



Accessing array members

- Array members are accessed by indices using the subscript operator `[]`. The indices are integers starting from 0.
- For instance, if `anArray` is an array of three integers, then:
 - the first member is `anArray[0]`
 - the second member is `anArray[1]`,
 - and the third member is `anArray[2]`.
- The indices can be any expression that has an integer value.

If an index is out of range, i.e., less than 0 or greater than `length-1`, a run-time error occurs. 200

Initializing array members

- Array members can be initialized individually using the indices and the subscript operator.

```
int [] intArray = new int[3];  
intArray[0] = 3;  
intArray[1] = 5;  
intArray[2] = 4;
```

- Array members may also be initialized when the array is created:

```
int [] intArray;  
intArray = new int [] { 3, 5, 4 };
```

Partial initialization of an Array

- An array may be partially initialized.

```
int [] intArray;
```

```
intArray = new int [5];
```

```
intArray[0] = 3;
```

```
intArray[1] = 5;
```

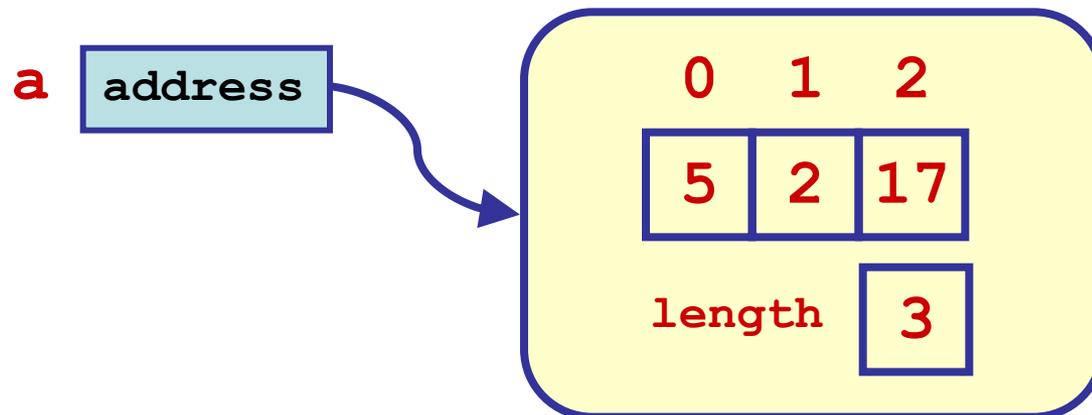
```
intArray[2] = 4;
```

- In this case, `intArray[3]` and `intArray[4]` are undefined.

- When an array is processed, we may need another variable (or variables) to keep track of the indices for which we have assigned values.

Reference Types

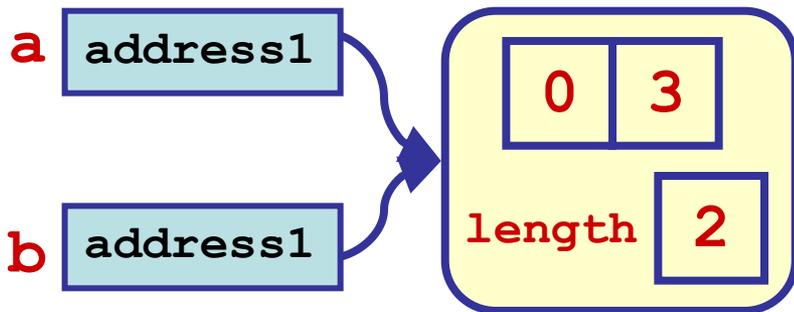
- An array type is a **reference type**, because of the “pointer” to the array.
- It is important to distinguish the reference (pointer) from the “item being pointed to”.
 - In the diagram below, **a** is the reference, and the array is what is being pointed to.
 - Java does not allow us to peek inside **a** to see what is in the pointer.



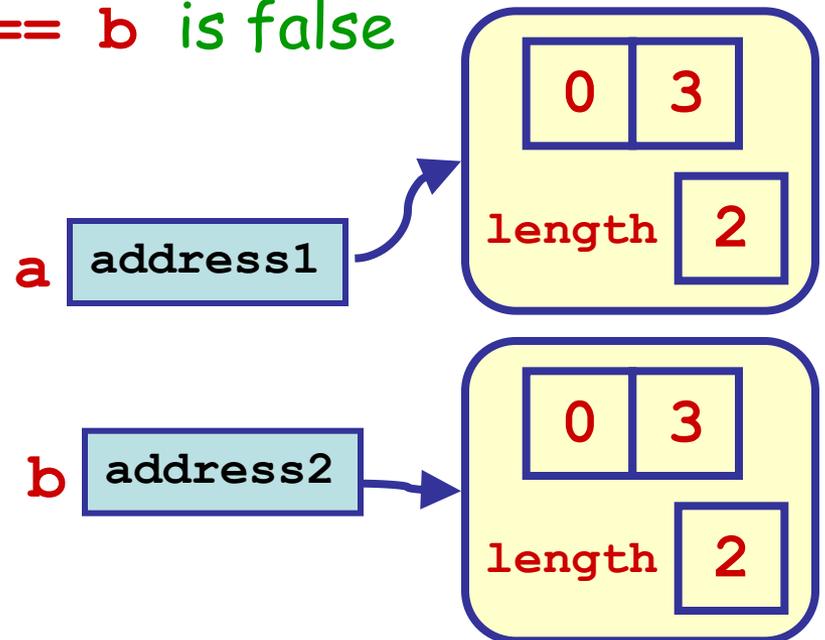
Reference Types

- What happens with assignment and comparison of reference types?
 - It is the **references** that are compared or assigned, not the arrays.

a == b is true

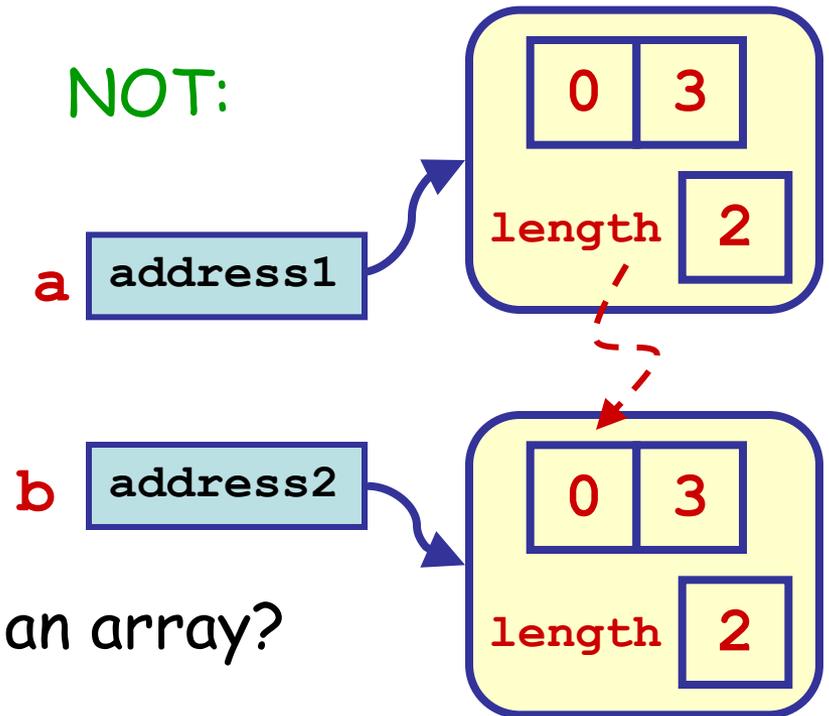
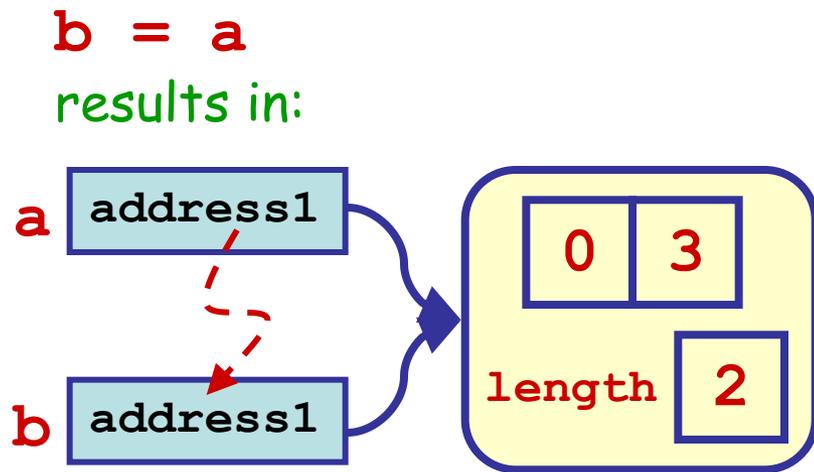


a == b is false



Reference Types

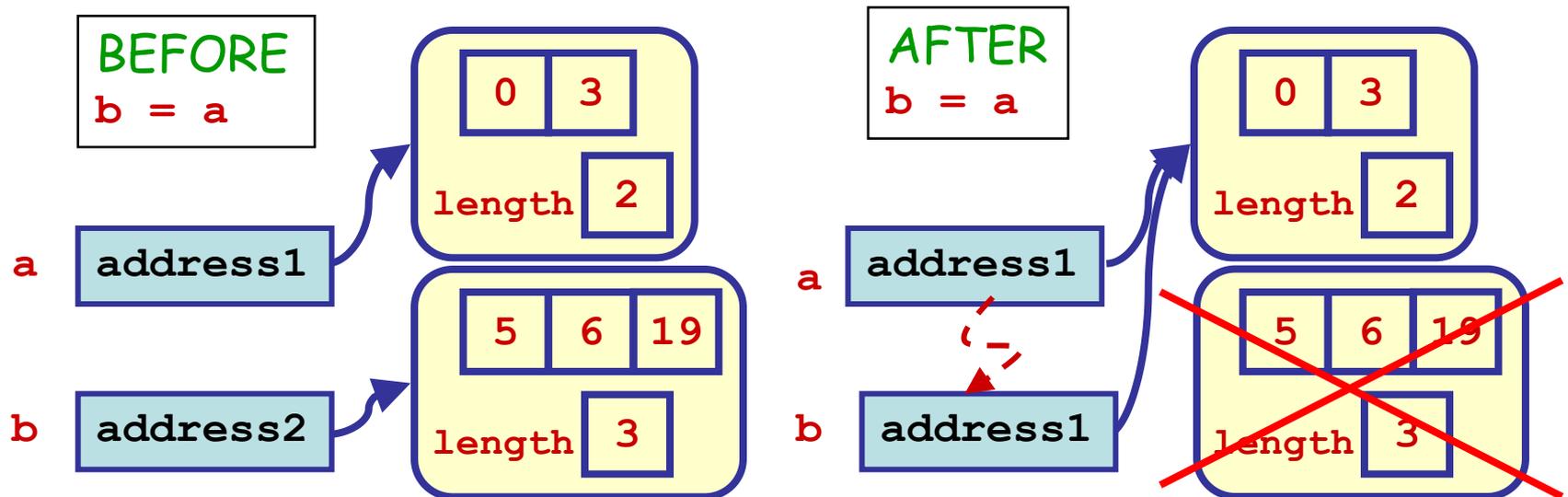
- Assignment only copies a reference, not the object to which it points.



- How can we make a copy of an array?

Lost references

- With reference types, be careful that you don't "lose" an object to which a reference points.



- After the assignment, there is no reference to the second array. The second array will be forgotten by Java and **cannot** be recovered.

"Objects can be classified scientifically into three major categories: those that don't work, those that break down and those that get lost." - R. Baker

Finding the maximum member in an array

- Problem (recall exercise 6-16):
 - Suppose an array of numbers is given, we want to find the maximum value of a member in this array.
- Idea:
 - Use the idea of scan and update. First, set the first member as the initial candidate to be the maximum. Then look at the other members one by one and keep track of the maximum value seen so far.
- We use a loop to look at the members in the given array.

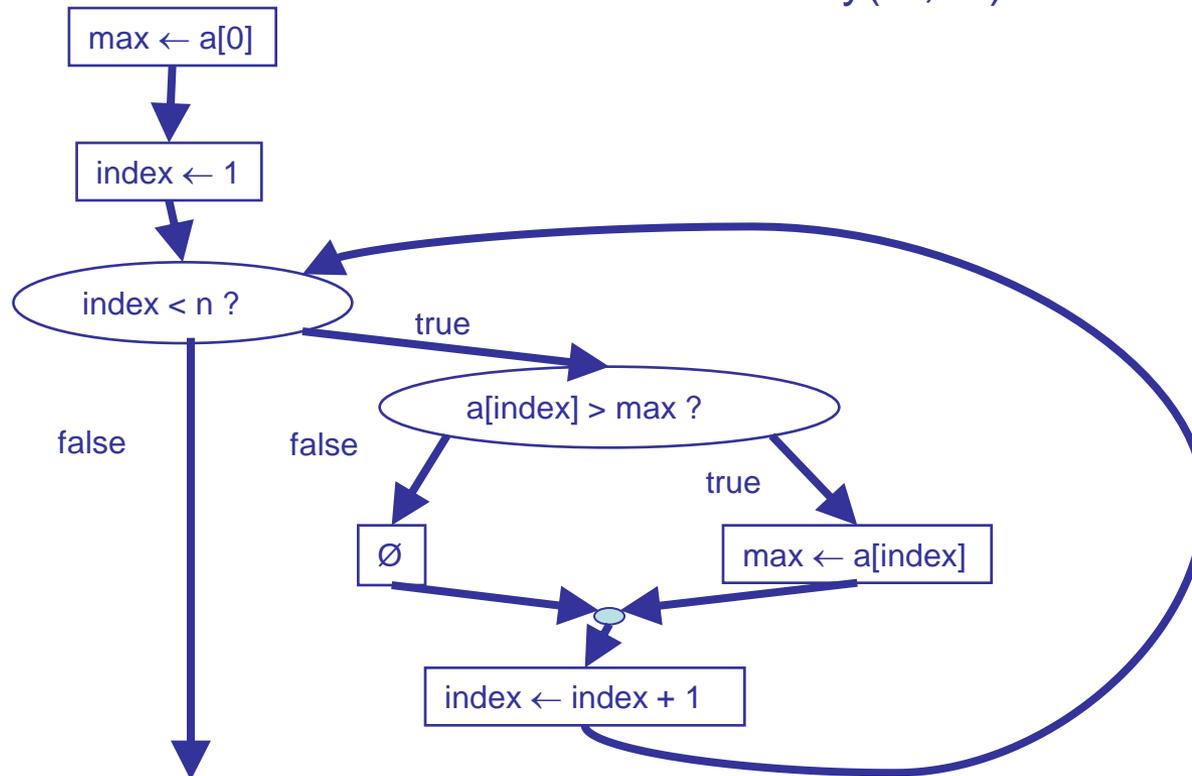
The Algorithm

GIVENS: n (a positive integer)
 a (array containing N values)

INTERMEDIATE: $index$ (indices for a)

RESULT: max (maximum member of a)

HEADER
BODY $max \leftarrow \text{maxInArray}(a, n)$



Translating 6-16 to Java



String Variables

- String variables have always presented a challenge for programming languages.
 - They have varying sizes, and for internal storage purposes, the computer would prefer to predict in advance the amount of storage needed for the value of a variable.
- As a result, strings have often been a "special case" in a programming language.

Strings in Java

- Strings in Java are also accessed using reference variables.
 - They are similar to an array of characters.
 - **EXCEPT:**
 - You don't need to use **new** to create a string
 - You **don't** use **[]** to access the characters in the string.
- Example:

```
String message = "Hello World!";  
System.out.println( message );
```
- There is a class (data type) **String** that provides many useful methods.
 - This means that Strings are objects (more on objects in the second half of the course).

Useful **String** methods

- Suppose we have

```
String message = "Hello World!";
```

Then:

- To find the length of a string:

```
int theStringLength = message.length();
```

- To find the character at position *i* (numbered from 0):

```
int i = 4;
```

```
char theChar = message.charAt( i );
```

- To change any primitive data type to a **String** :

```
int anInteger = 17;
```

```
String aString = String.valueOf( anInteger );
```

```
// works for int, double, boolean, char
```

- To append one string after another (concatenation):

```
String joinedString = string1 + string2;
```

Comparing Strings

- A **String** is a reference type and so they are **NOT** compared with **==**.
- The **String** class has a method **compareTo()** to compare 2 strings.
 - The characters in each string are compared one at a time from left to right, using the collating sequence.
 - The comparison stops after a character comparison results in a mismatch, or one string ends before the other.
 - If $str1 < str2$, then **compareTo()** returns an **int** < 0
 - If $str1 > str2$, then **compareTo()** returns an **int** > 0
 - If the character at every index matches, and the strings are the same length, the method returns **0**

Exercise 6-19: Comparing Strings



- What is the value of **result** for these examples?

- Example 1:

```
String str1 = "abcde";  
String str2 = "abcfg";  
int result = str1.compareTo(str2);
```

- Example 2:

```
String str1 = "abcde";  
String str2 = "ab";  
int result = str1.compareTo(str2);
```

"Everything should be made as simple as possible,
but not one bit simpler ."
-- A. Einstein

Section 7: Program Structure

Objectives:

- Translating Algorithms to Methods
- Arrays as Parameters
- Multiple Class Program
- Problem Decomposition (*top-down*)

Historical note ...

- 1976: [Steve Jobs](#) and [Steve Wozniak](#) created the first personal computer called [Apple I](#).
- The computer sold for 666.66 \$; it had 256 bytes of ROM, 4 K bytes of RAM and video output on the television set.



- In June 1975, [Bill Gates](#) and [Paul Allen](#) renamed their company Traf-O-Data into **Microsoft**.
- Produced MS-DOS, **Windows**, Basic-Microsoft, and later **Visual Basic**.

Using Multiple Methods

- We have been using programs containing 2 sub-programs (2 algorithms translated to 2 Java methods)
 - If an algorithm contains a complex instruction bloc nested within another bloc, the nested bloc can be placed in a separate algorithm.
 - Thus an algorithm can in turn call other algorithms. In this fashion, algorithms can be kept simple, short and clear.
 - And thus it is possible to divide a complex problem into multiple tasks which can in turn be subdivided (top-down design).

Using Multiple Methods (continued)

- The program starts with a **main** method (translated to a **main** method), which may read in some values and output the result. The **main** algorithm calls one or more other algorithms, each of which are translated to Java methods.
 - In this way, the main algorithm (method **main**) acts as a dispatcher.
 - Try to keep **main** as simple as possible - you should be able to tell what the overall program by examining **main**

Method Accessibility

- In Java,
 - Methods are collected inside a "class"
 - A Java program can be made up of many classes
- If a method is **public**, it can be called from anywhere in a program.
- If a method is **private**, it can only be called from inside the class where it is defined.
- There are two other levels of access: **protected** and "package", that are between public and private. We will not use them in this course.

Arrays as Parameters

- An array is a reference type; i.e. is accessed using a reference variable.
- Arrays are not passed from one method to another method, it is the **reference** (i.e. the content of the reference variable) that is passed to a method (or can be returned by a method).
- The result is that there are (temporarily) two references to the same array.
- While we cannot modify the original reference variable, a called method **can** modify the contents of the array. These changes to the array contents will remain after the called method returns.
 - The copy of a variable of a primitive type is trashed when the method returns.
 - For an array, it is the **copy of the reference variable** that is trashed on return.

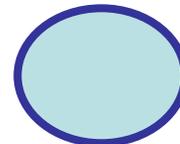
Passing primitive and reference types to a method

At caller:

`m(anInt, anArray) :`

`anInt`

`anArray`



4

5

3

2

length

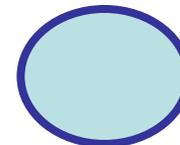
3

copy

copy

At called method:

`m(int x, int[] y)`



4

`x`

`y`

Exercise 7-1: Trace this Program



```
class SwapTilYouDrop
{
    public static void main (String args[ ])
    {
        int i = 0;
        int[ ] a = { 2, 4, 6, 8, 10, 12 } ;
        while ( i <= 2 )
        {
            arraySwap(a, i, 5 - i ) ;
            i = i + 1;
        }
        for ( i = 0 ; i <= 5 ; i = i + 1 )
        { System.out.println( "a[" + i + "] is " + a[i] ); }
    }
    // arraySwap : swaps values of x at positions i,j
    // Givens: x, an array, i,j, 2 indices in x
    public static void arraySwap(int[ ] x,int i,int j)
    {
        // DECLARE VARIABLES/DATA DICTIONARY
        int temp ; // Intermediate, holds x[i]
        // BODY OF ALGORITHM
        temp = x[i] ;
        x[i] = x[j] ;
        x[j] = temp;
    }
}
```


Exercise 7-1: Trace (Table 2)

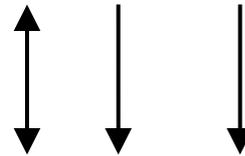
`arraySwap(a, i, 5-i)`
 ↑ ↓ ↓
`arraySwap(x, i, j)`

Statement	x	i	j	temp	Array in Table 1
Initial values					
1. <code>temp = x[i]</code>					
2. <code>x[i] = x[j]</code>					
3. <code>x[j] = temp</code>					



Exercise 7-1: Trace (Table 3)

`arraySwap(a, i, 5-i)`

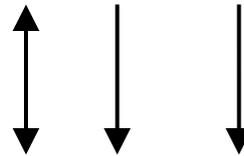


`arraySwap(x, i, j)`

Statement	x	i	j	temp	Array in Table 1
Initial values					
1. <code>temp = x[i]</code>					
2. <code>x[i] = x[j]</code>					
3. <code>x[j] = temp</code>					

Exercise 7-1: Trace (Table 4)

`arraySwap(a, i, 5-i)`



`arraySwap(x, i, j)`

Statement	x	i	j	temp	Array in Table 1
Initial values					
1. <code>temp = x[i]</code>					
2. <code>x[i] = x[j]</code>					
3. <code>x[j] = temp</code>					

Programs with more than one Class

- A program may have more than one class. If you save all classes in a program in one directory, any class may call a **public** method in any other class in the same directory.
- When a (**static**) method is called from another class, use the name of the class with the dot operator.
 - For example, if we include class **Library** in the same directory in an assignment program, you may call a method such as **aMethod()** by
Library.aMethod() ;

Library Classes

- Instead of putting all our methods in the same class as **main** (the class that contains our program) it is better to separate them into coherent groups and put each group in a class of its own.
- These classes will not be programs - they have no **main** method. Each will be a small library of methods that can be used by other methods.
- Such classes can be compiled on their own but cannot be run as standalone programs. They must be compiled before attempting to compile any other class that uses them.

Exercise 7-2: Validating numbers

- Some credit cards use the following method to determine the validity of a card number: the number is valid if its last digit is equal to the last digit of the sum of the other digits.
- For example:
 - 5792 is invalid ($5+7+9 = 21$)
 - 4231628 is valid ($4+2+3+1+6+2 = 18$)
- **Problem:** Write a program that checks if a card number given by the user is valid. Use a loop to check more than one card, until the user enters the number zero.
- **Note:** The credit card numbers usually have 16 digits; the type `int` is not sufficient for representing such numbers (can only represent values up to 2 147 483 647).
- **Assumption:** The first 4 digits of the credit card number are not all zero.

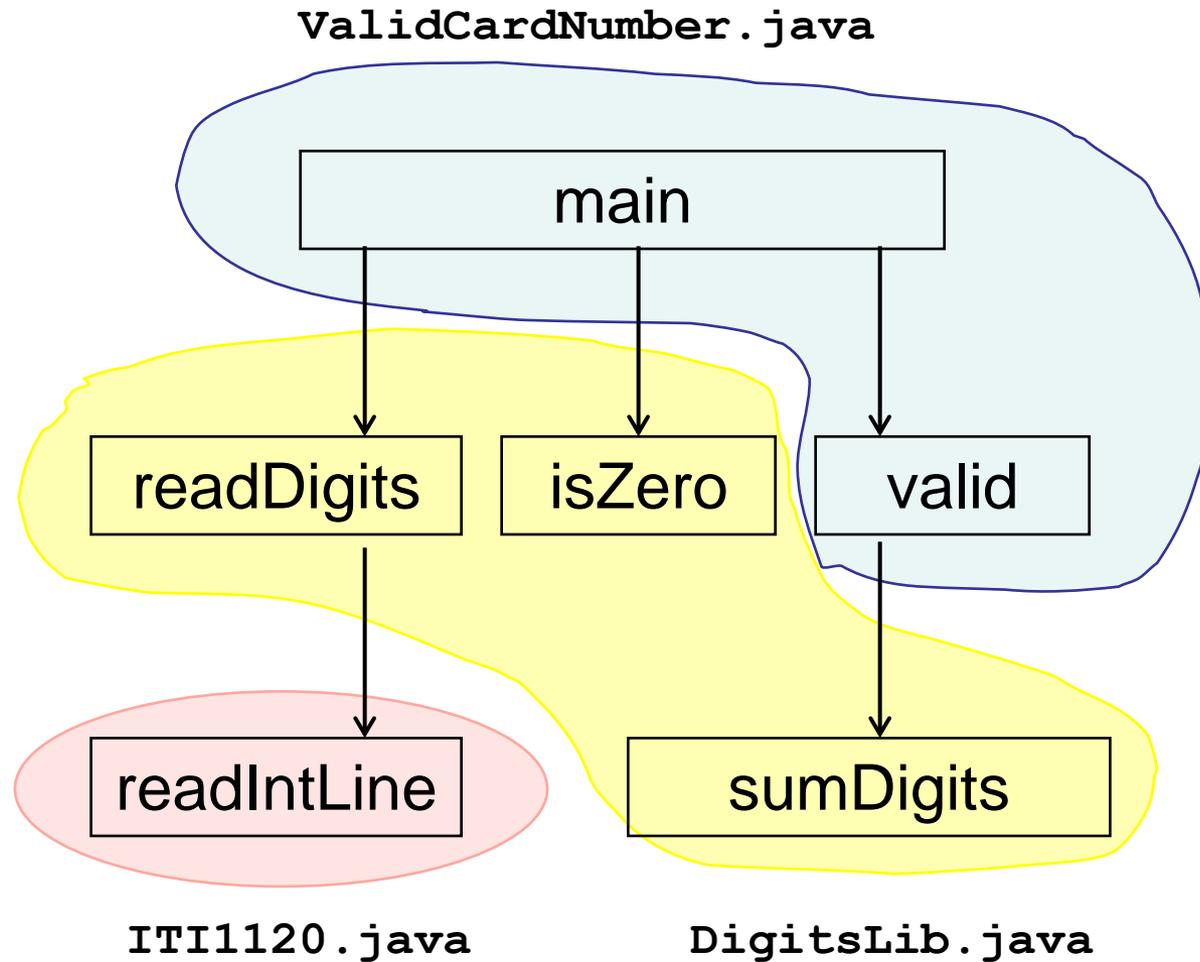
Exercise 7-2: Data structure for the numbers

- A **data structure** is used to organize the data used in a program.
- In this problem, we use an array of 4 integers (*int*) to represent the credit card number.
 - The real numbers don't have the desired precision.
 - Each integer in the array is used to represent 4 digits in the credit card.

Exercise 7-2: Designing the program

- A possible algorithm needs to do the following (not all the details are provided here):
 1. Read the card number input by the user (in a separate method).
 2. Check if the number is 0 (the first 4 digits).
 3. If the number is not 0, check if the number is valid or not (in a separate method).
 - In order to check the validity of the number, call a method (sub-algorithm) that computes the sum of its first 15 digits.
 - Display the result.
 - Read another number input by the user.

Exercise 7-2: Structure diagram



Exercise 7-2: main method



```
/* the main method calls the other methods in order to solve  
particular tasks. */
```

```
public static void main (String [ ] args)
{
    int [ ] digits; // reference variable to array of digits
    boolean testValid; // indicates if card is valid
    // body
    digits = DigitsLib.readDigits( ); // call to read the data
    while ((digits.length == 4) && (!DigitsLib.isZero(digits)))
    {
        // sends the number to the valid() method
        testValid = valid(digits);
        // print the result
        if (testValid)
            { System.out.println("This number is valid."); }
        else
            { System.out.println("This number is invalid."); }
    }
}
```

Exercise 7-2: valid() method



```
/* This method validates the credit card number */
private static boolean valid(int [ ] digits)
{
    int firstThree; // first three digits of last group
    int lastDigit; // last digit of credit card number
    int sum; // sum of first 15 digits
    boolean isValid; // result: true if number is valid
    // find the first 3 digits of the last group
    firstThree = 
    // find the last digit of the number
    lastDigit = 
    // find the sum of the first 15 digits
    sum = 
    // determines the validity
    isValid 
    return isValid;
}
```

Exercise 7-2: isZero() method



//first version: only the first digits need to be 0

```
public static boolean isZero(int [ ] digits)
{
    boolean flag; // result
    flag = digits[0] == 0;
    return(flag);
}
```

//second version: all 16 digits need to be 0

```
public static boolean isZero(int [ ] digits)
{
    
}
}
```

Exercise 7-2: readDigits() method



```
/* This method asks the user to input a credit card  
number as 4 integers, that will be placed in an  
array. This method calls readIntLine( ) from the  
class ITI1120 to read the array in integers. */
```

```
public static int [ ] readDigits( )  
{  
    int [ ] intArray; // reference to array  
    System.out.println  
        ("Please input the credit card number as four ");  
    System.out.println  
        ("numbers of four digits, separated by spaces;");  
    System.out.println("or press 0 to finish.");  
    intArray = ITI1120.readIntLine( );  
  
}
```

Exercise 7-2: sumDigits()



// Returns the sum of the digits of a number x

```
public static int sumDigits(int x)
{
    int sum; // result - sum of digits
    // Body
    sum = 0;
    while (x != 0)
    {
        
    }
    return sum;
}
```

Exercise 7-2: Combine all of them !

```
import java.io.* ;
class ValidCardNumber
{
    public static void main (String [] args) { ... }
    private static boolean valid (int [ ] digits) { ... }
}
```

```
import java.io.* ;
class DigitsLib
{
    public static boolean isZero (int [ ] digits) { ... }
    public static int [ ] readDigits( ) { ... }
    public static int sumDigits (int x) { ... }
}
```

- Replace the { ... } by the modules from the previous pages, and save them in two files (**ValidCardNumber.java** and **DigitsLib.java**)
- Also put the **ITI1120.class** in the same directory.
- Now you can validate credit cards 😊

"Inside every large problem is
a small problem struggling to get out."
-- C.A.R. Hoare

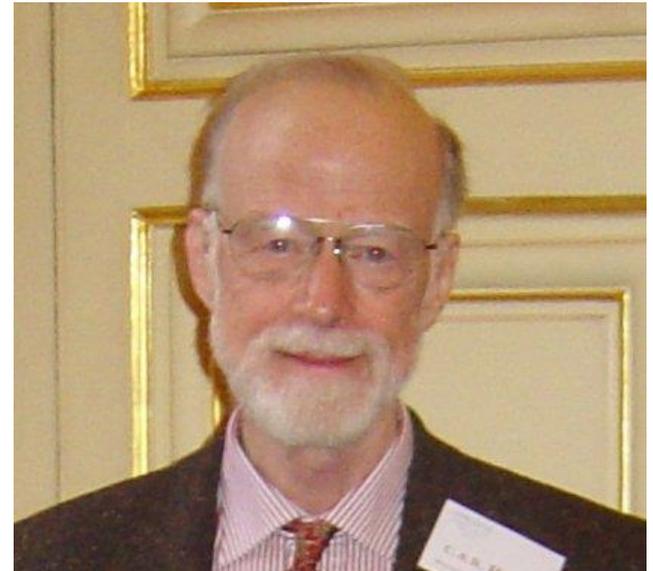
Section 8: Recursion

Objectives:

- Defining and Illustrating Recursion
- Template for Recursion
- Examples

Historical note ...

- Charles Antony Richard (Tony) Hoare, British computer scientist, developed, in 1960, the sorting algorithm (recursive) the most used: Quicksort.
- He also developed the Hoare logic used in software engineering for program verification and for programming by contracts.
- He is at the origin of the concurrent programming language CSP (Communicating Sequential Processes).

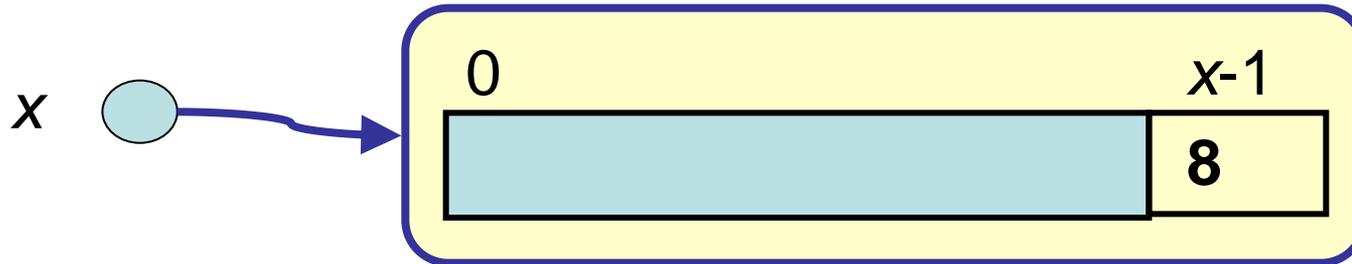


Recursion

- **Recursion** is a problem-solving technique which uses smaller sub-problems of problem P; the parameter of the sub-problem is "smaller" (typically an integer).
- In recursion the sub-problems are similar to problem P, but are simpler versions of it.
- When the parameter is small enough (the **base case**), the sub-problem is solved directly.
- If the parameter is large, the **problem** is reformulated in terms of a sub-problem with a smaller parameter.
 - The solution to the larger problem is found using the solution of the smaller sub-problem.
 - If the parameter of the sub-problem is still too large, then it must be reduced until we reach the base case.
- The problem and sub-problems are solved using multiple executions of a single subprogram (algorithm/method) which calls itself
 - Each execution of the subprogram can be viewed as an instance of its execution.

Recursion Example 1: basic idea

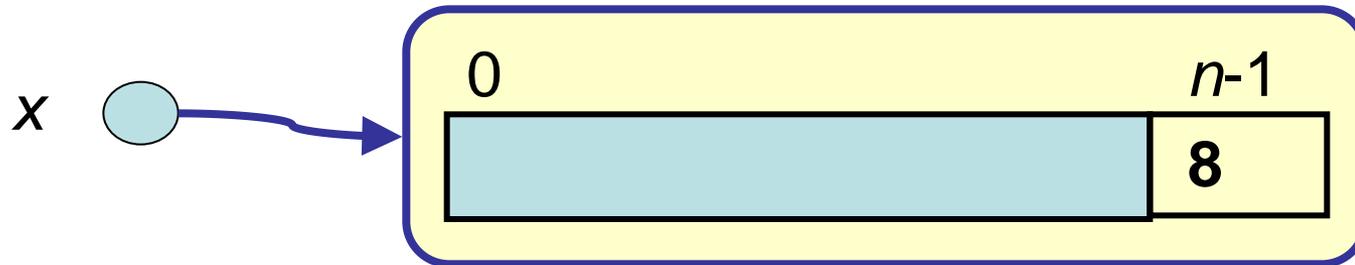
- What is the maximum value in positions $0 \dots (n-1)$ of array x ?



- (the maximum value in the shaded area is m)
- Answer: the maximum of m and position $n-1$

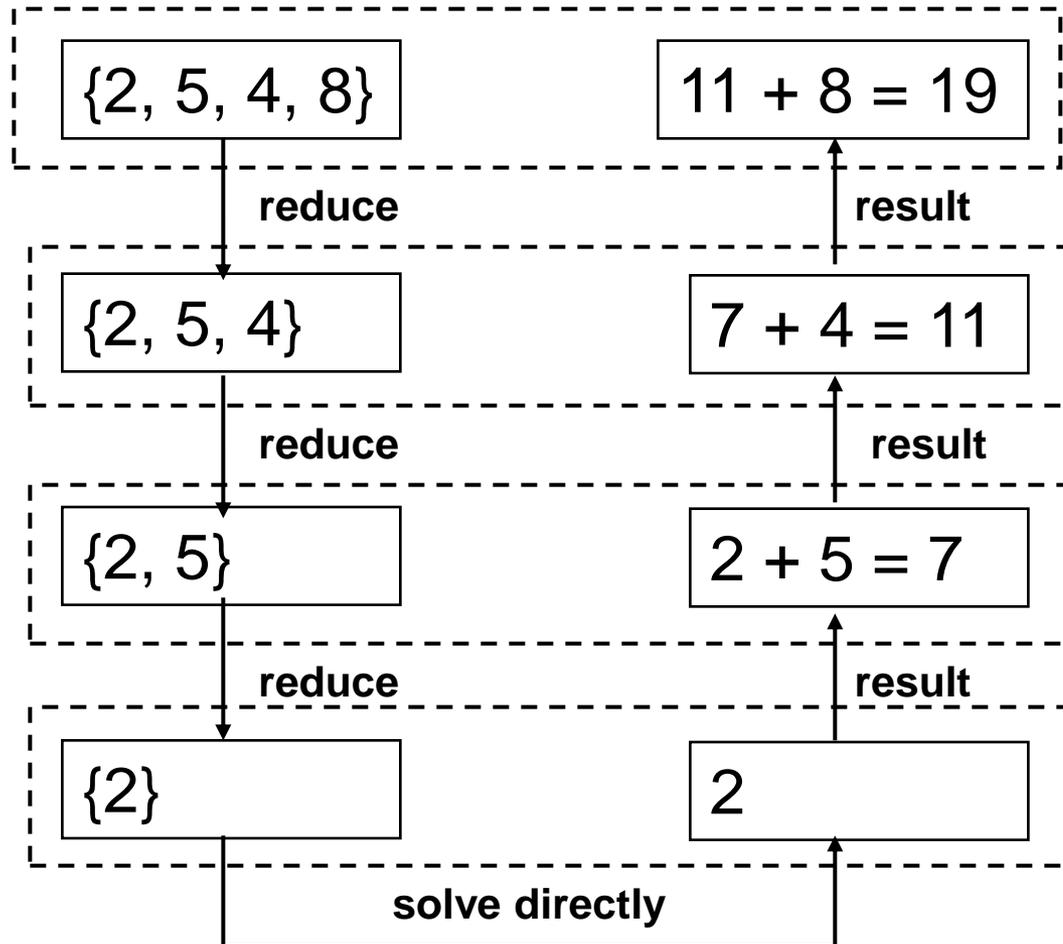
Recursion: Example 2 (basic idea)

- What is the sum of the numbers in positions $0 \dots (n-1)$ of array x ?

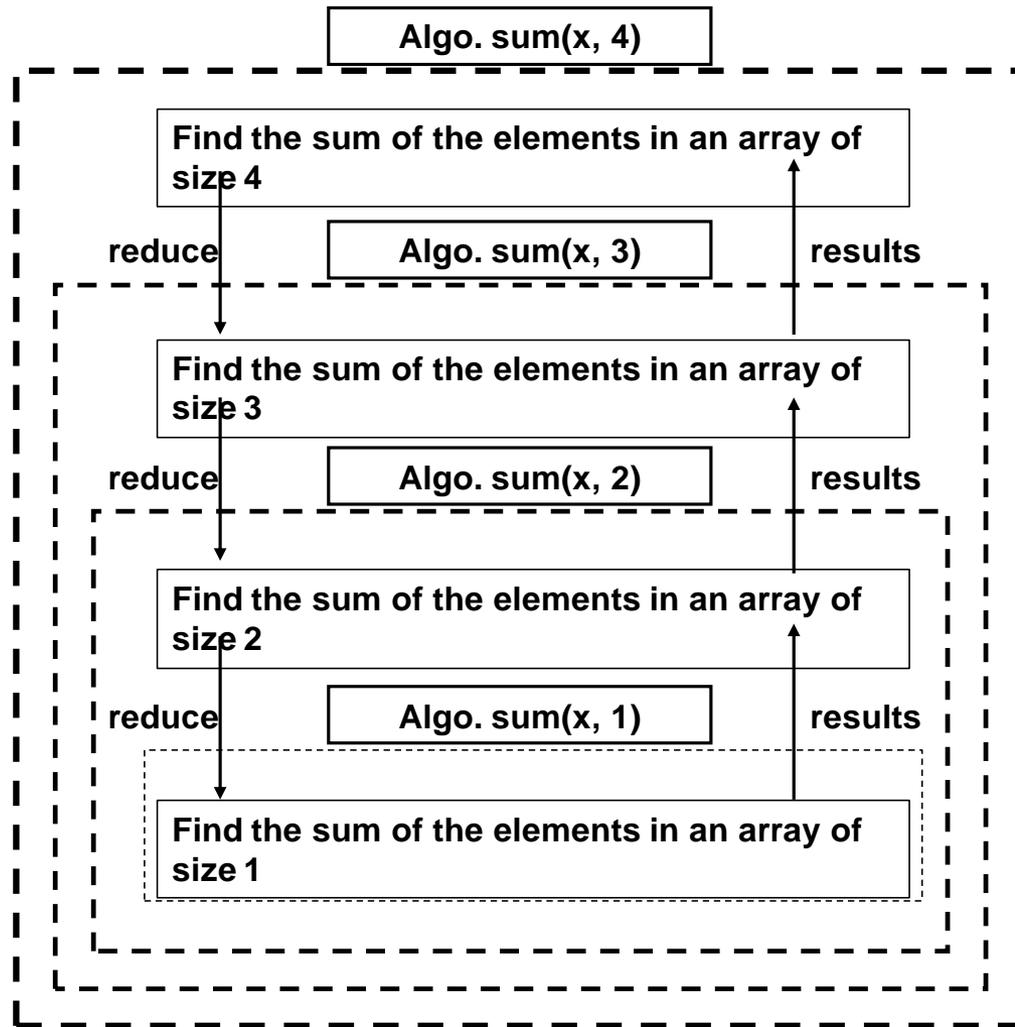


- (the sum of all values in the shaded area is s)
- Answer:
 - The sum is $s +$ the value at index $n-1$

Example for $x = \{2, 5, 4, 8\}$



Recursive Calls

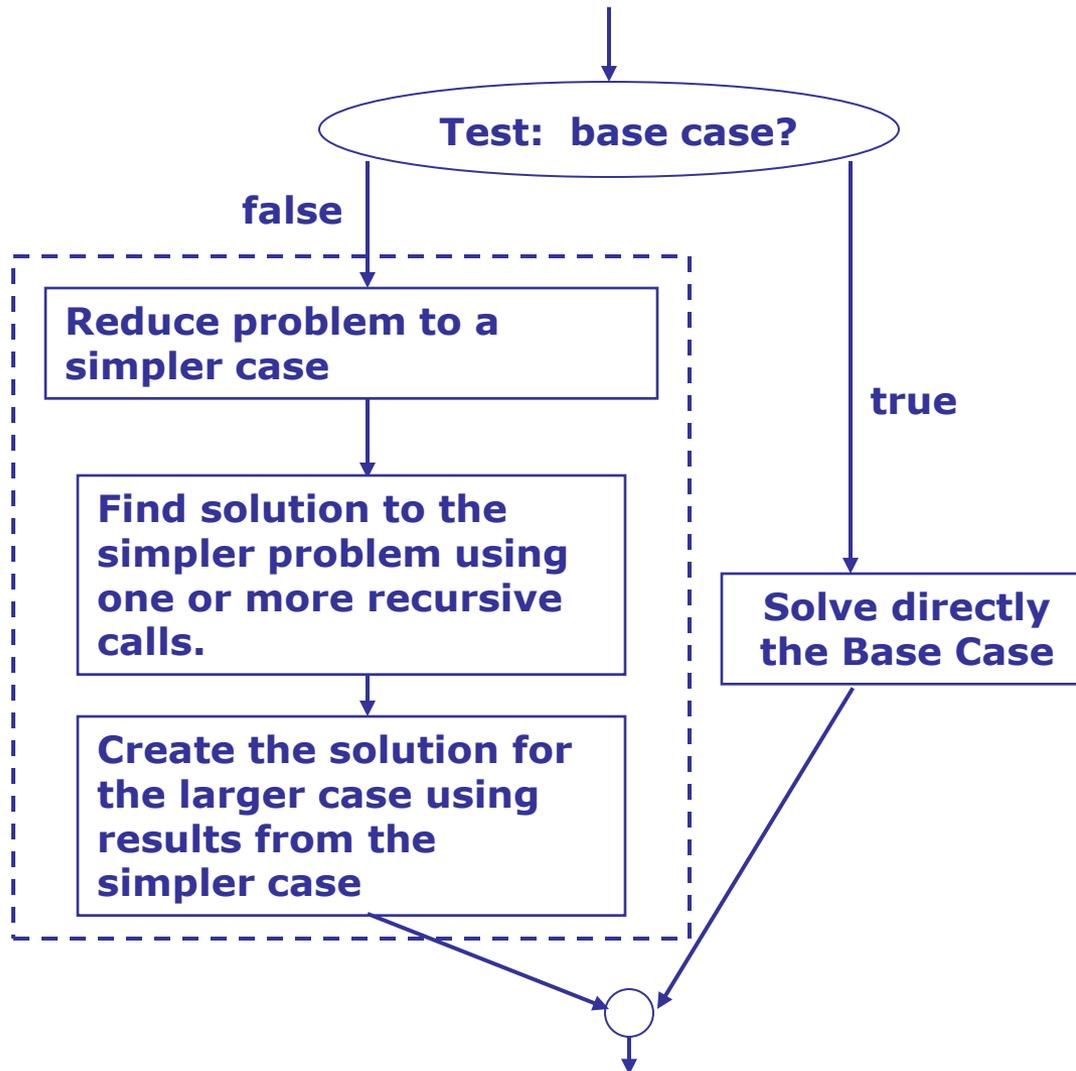


Components of Recursion

There are 3 components to recursion:

1. A test to see if the problem is simple enough to solve directly (i.e. non-recursively): the "base case"
2. The solution for the base case.
3. A solution to the problem which involves solving one (or more) smaller versions of the same problem.

Template for recursive algorithms



Exercise 8-1: Recursive Sum of Array (II) ?

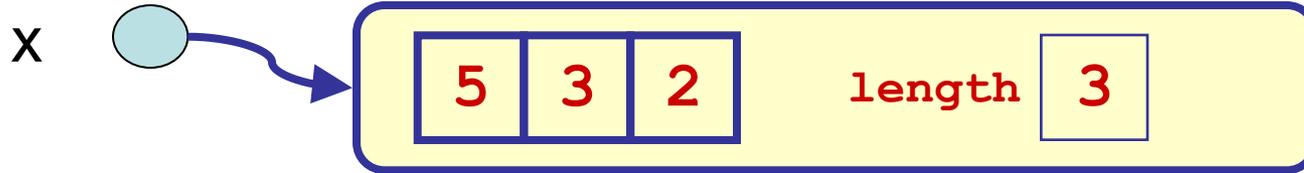
BODY:

Exercise 8-1: Simplified Version



-
- m can be substituted directly in the call

Exercise 8-1: Trace for this value of x: ?



Exercise 8-1: Trace Table 2



Exercise 8-1: Trace, Table 3



Recursive Algorithms and Recursive Methods

- An algorithm that calls itself, with different **GIVENS**, is called a **recursive algorithm**.
- A Java method that calls itself, with different arguments, is called a **recursive method**.
- We can also have circular recursion, which is more difficult to detect and manage
 - Ex: a calls b, b calls c and c calls a.
 - Not studied in this course.

Template for Recursive Java Method

```
public static typeOfReturn recursiveMethod
    (int size, <otherParameters>)
{
    typeOfReturn result;
    typeOfReturn partialResult;

    if (baseCase(size))
        { <Find the result directly > }
    else
    {
        { < Reduce to an instance with smaller >; }
        < partialResult > =
            recursiveMethod(smaller, < otherParameters >);
        { < Calculate result, using partialResult > }
    }
    return result;
}
```

Comment: There are NO loops!

Exercise 8-2: Translate the Recursive recSum to Java



Variation

- Having the base case do nothing is variation that is fairly common.
- For example, in the algorithm `recSum`, we can use the base case `n=0` and that the result is 0.
- In this case the method `recSum` becomes:

```
public static int recSum(int [] x, int n)
{
    int s = 0; // RESULT
    if (n >= 1)
    {
        s = recSum(x, n - 1);
        s = s + x[n - 1];
    }
    return s;
} // Is this easier to read or understand..?
```

Recursion Examples

Example 1 - What is the maximum value in positions 0...(n-1) of array x?

8-1 What is the sum of the numbers in positions 0...(n-1) of array x?
(Exercise 8-2 - Translation to Java)

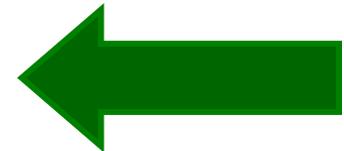
8-3 Find x^n where x and n are integers and $n \geq 0$, $x \geq 1$.

(a) Direct algorithm

(b) Alternative algorithm based on fact:

$$x^n = x^i * x^{n-i}$$

- Choose i to get most efficient version.



8-4 Given an array a of more than n numbers, return TRUE if all the numbers in positions 0... n of a are equal, and false otherwise.

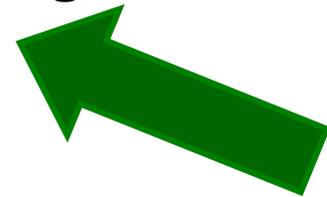
More Recursion Examples

8-5 Calculate $n!$

8-6 Find the sum of $1+2+\dots+n$.

8-7 Given an array a of n characters, reverse the values stored in positions Start to Finish.

8-8 Sort an array of numbers in increasing order.



Exercise 8-3: Find x^n (version 1)



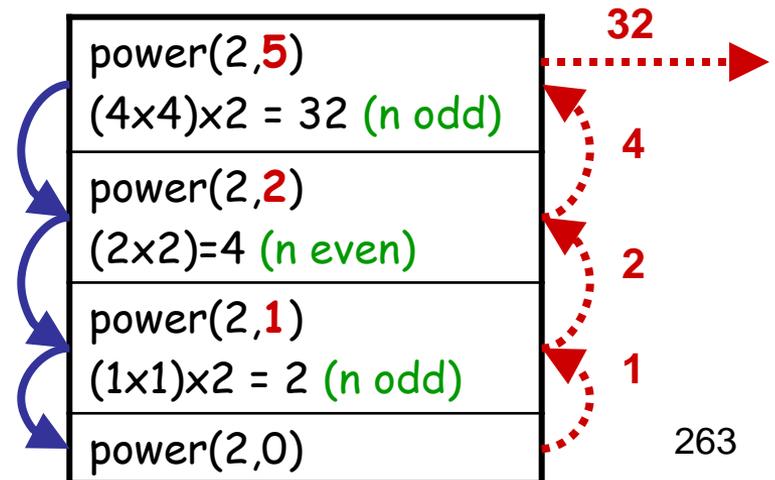
Exercise 8-3: Find x^n (version 2: more efficient)

Exercise 8-3: Reducing the problem size

- The second version of x^n is more efficient because we cut the problem size in half, instead of reducing it by one.
- This leads to fewer recursive calls:

→ recursive call

⋯ return value



Exercise 8-8: Sort an array of numbers, version 1

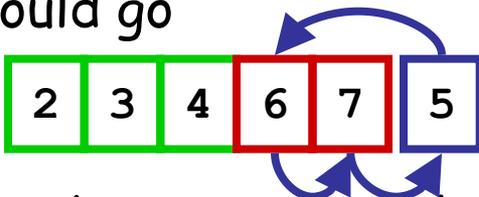
- General idea: "insertion sort": insert last value into previously sorted array.



- First, sort array of size $n-1$ using recursive call



- Second, determine where the value in last array position should go



- Third, move over values, and insert last value



Exercise 8-8: Sort an array of numbers, version 2

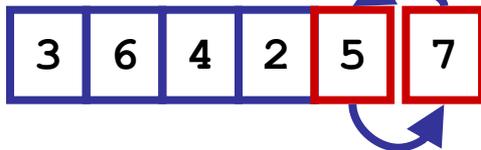
- General idea: "selection sort": select largest element, and put in correct position.



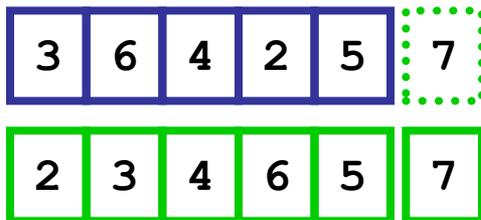
- First, find position  with maximum value in array



- Second, swap this position with last position



- Third, use recursion to sort "smaller" array



Exercise 8-8: Recursive selection sort ?

Exercise 8-8: Recursive location of largest value



Exercise 8-8: Recursive location of largest value

`locateLargestValue(x, 6)`



`locateLargestValue(x, 5)`



`locateLargestValue(x, 4)`



`locateLargestValue(x, 3)`



`locateLargestValue(x, 2)`



`locateLargestValue(x, 1)`



(base case)

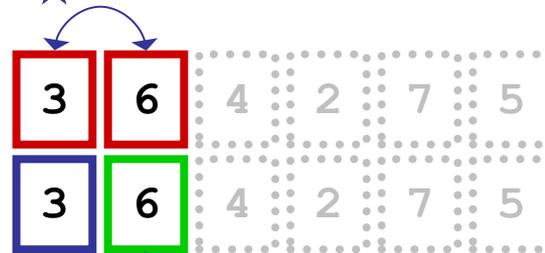
Exercise 8-8: Recursive location of largest value

repeated from previous page
`locateLargestValue(x, 1)`



(base case)

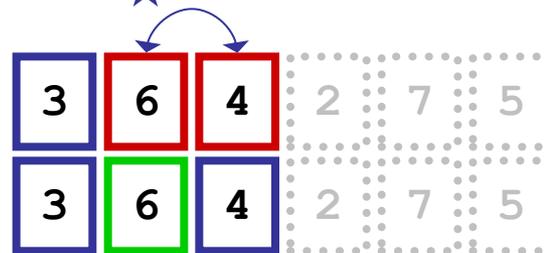
return to
`locateLargestValue(x, 2)`



(compare)

(choose)

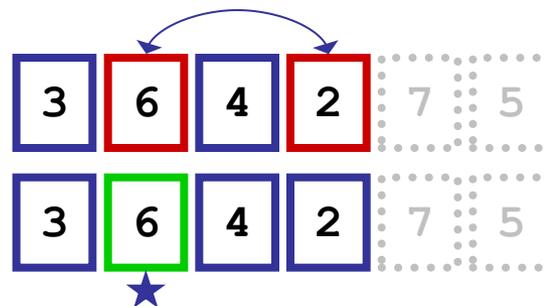
return to
`locateLargestValue(x, 3)`



(compare)

(choose)

return to
`locateLargestValue(x, 4)`

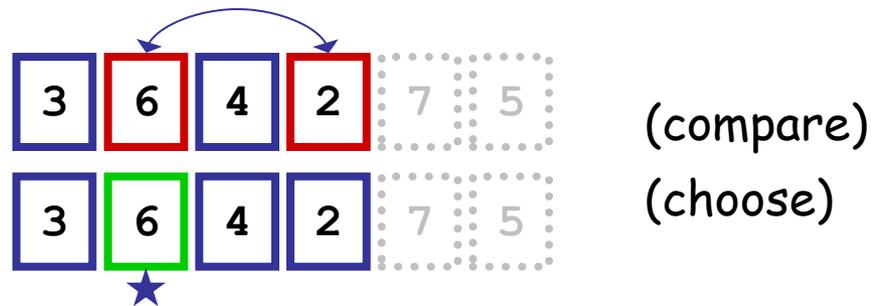


(compare)

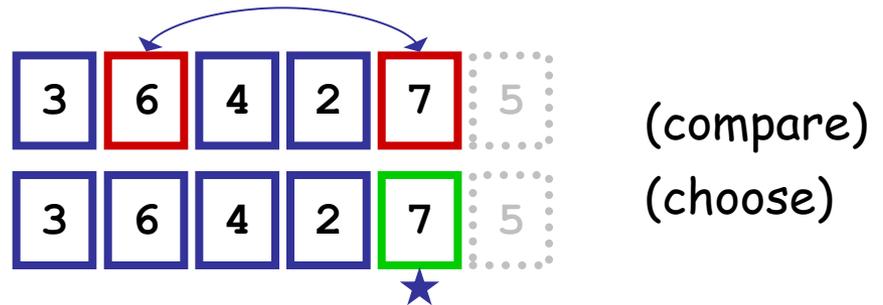
(choose)

Exercise 8-8: Recursive location of largest value

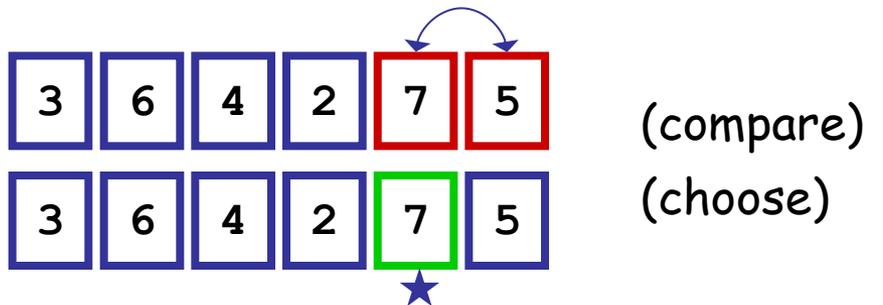
repeated from previous page
`locateLargestValue(x, 4)`



return to
`locateLargestValue(x, 5)`



return to
`locateLargestValue(x, 6)`

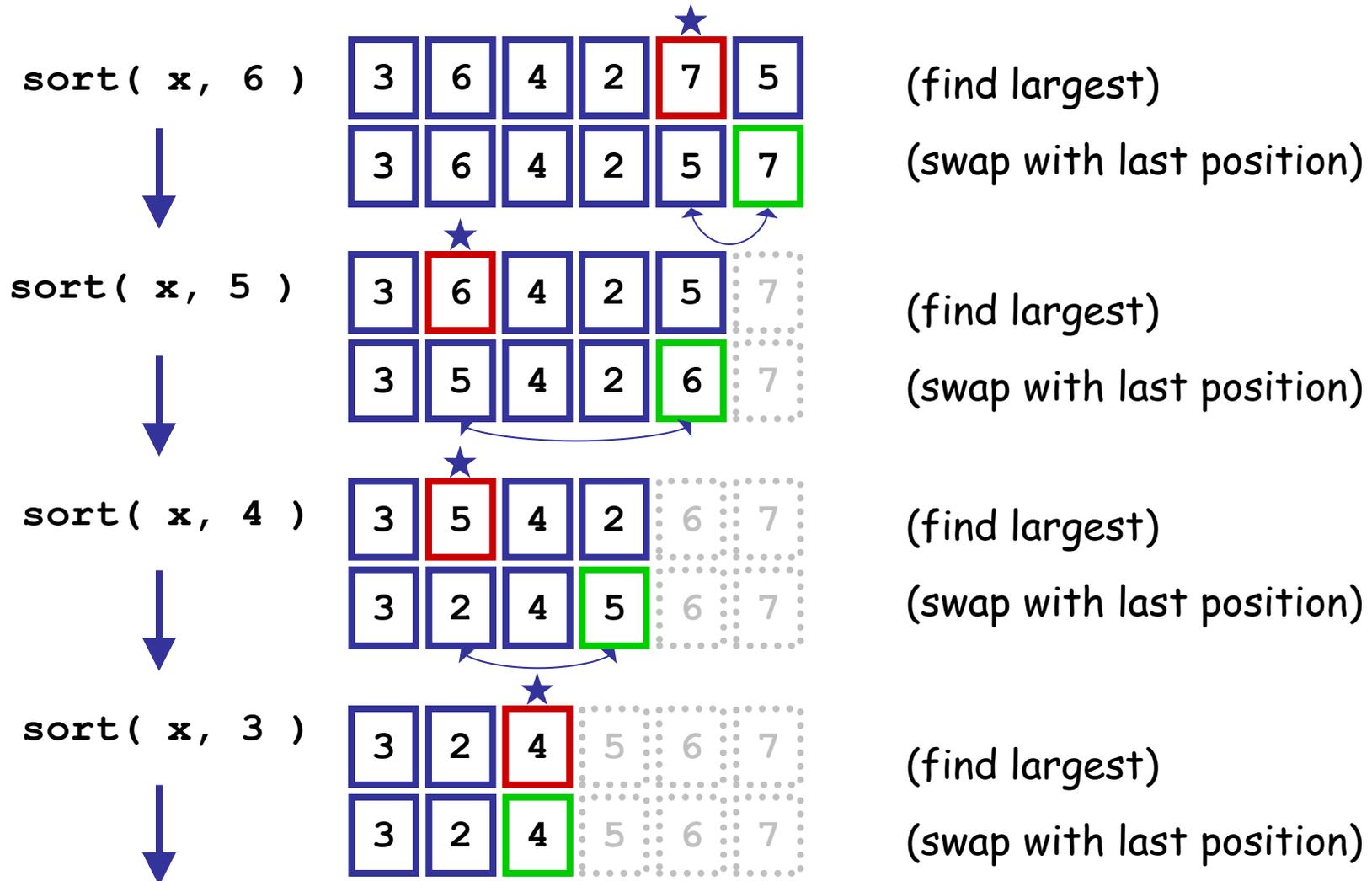


done!

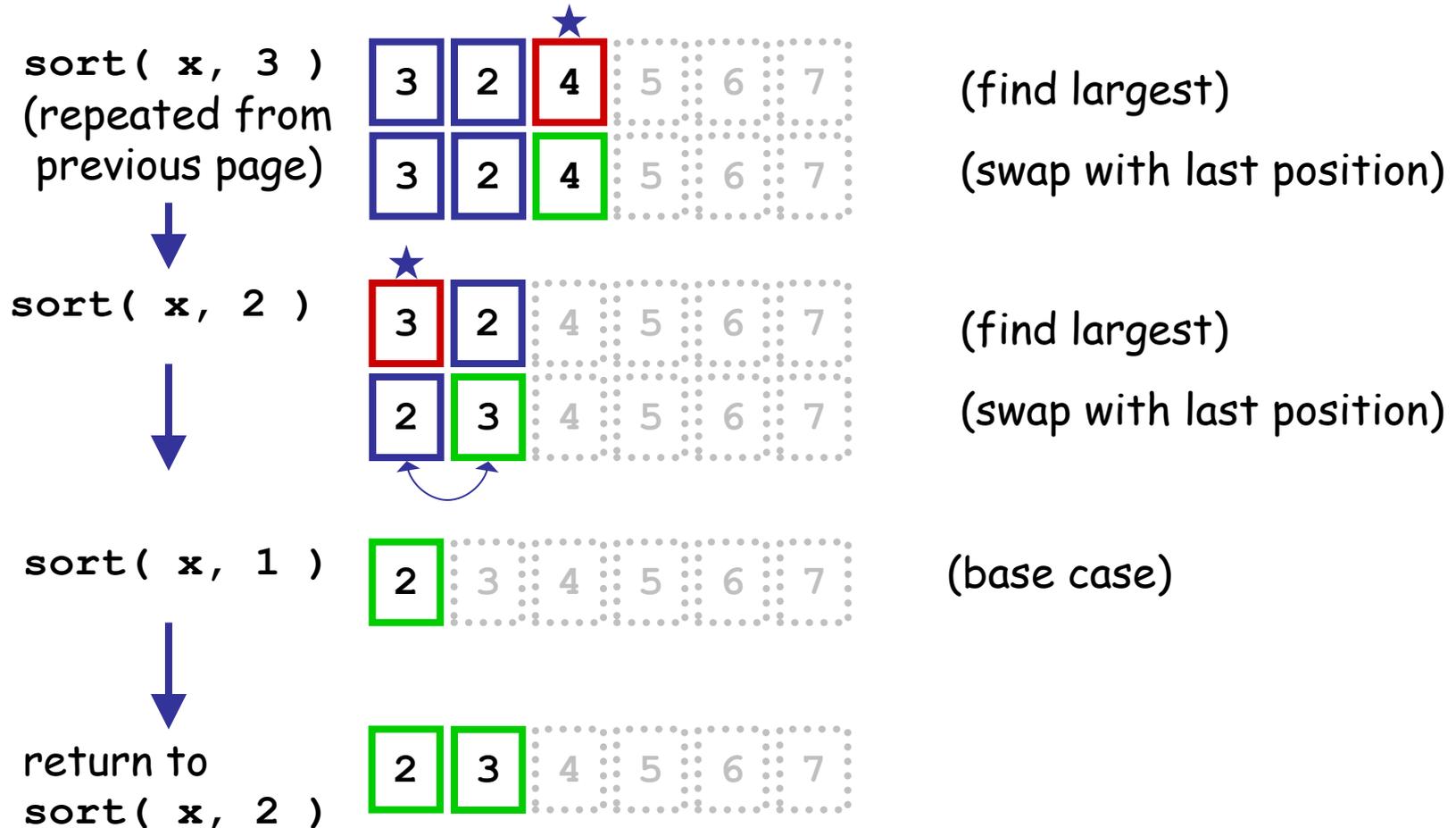
When does the problem actually get solved?

- In the previous example, we needed the result of a recursive call first.
 - Therefore, recursive calls were made without doing any “work” until the base case was reached.
 - It was after the recursive calls started returning that the actual comparisons were done, and the problem was solved.
- This is not always the case. The following illustration of the selection sort method illustrates the situation where the “work” is done before the recursive call, and when we reach the base case, the problem is solved.
 - However, we still have to return from all the recursive calls!

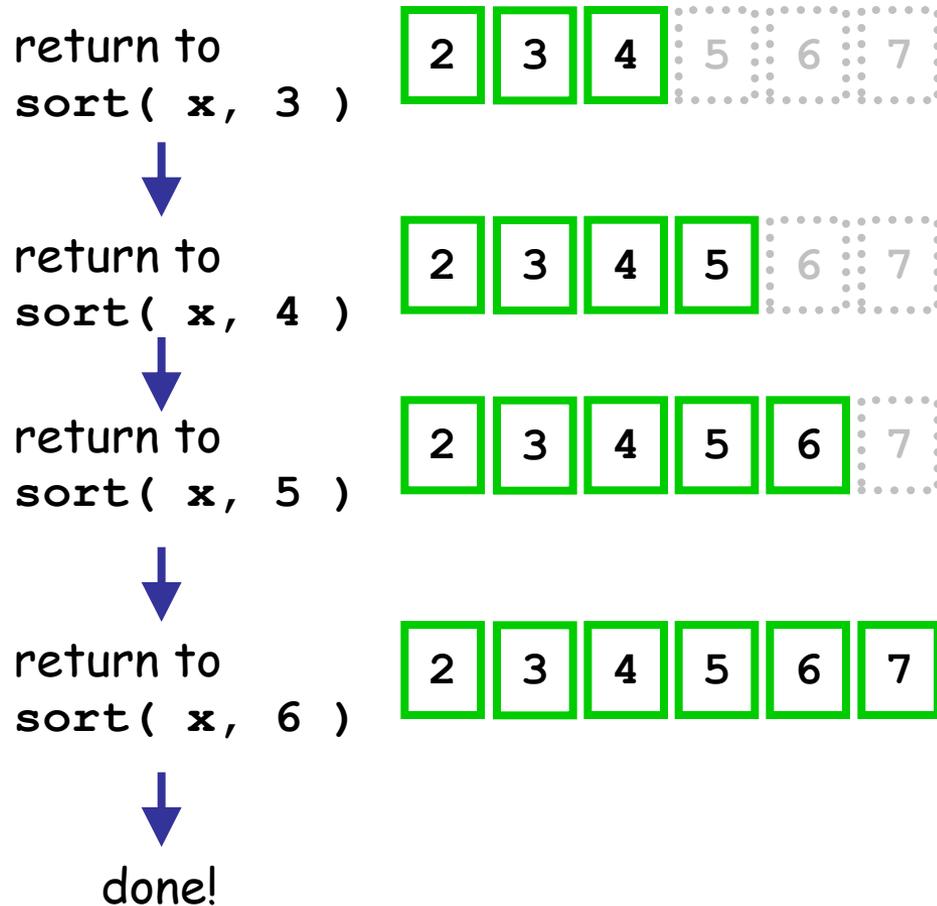
Exercise 8-8: Recursive selection sort



Exercise 8-8: Recursive selection sort



Exercise 8-8: Recursive selection sort



"The Matrix is a system, Neo"
-- from *The Matrix*

Section 9: Matrices

Objectives:

- Matrices = arrays of arrays
- Declaration, access, and modification
- Adjacency Matrices

Historical note ...

- 1998: Larry Page and Sergey Brin, founded a company that revolutionized the world of search engines and the Internet: **Google!**
- Many applications, such GoogleMaps.
- 450 000 servers.
- More than a billion queries per day!



Matrices

- An $r \times c$ matrix has r rows and c columns.
- Example. A 4×6 matrix of integers

$$m = \begin{bmatrix} 71 & 62 & 33 & 89 & 85 & 74 \\ 68 & 65 & 75 & 84 & 70 & 72 \\ 87 & 0 & 1 & 90 & 92 & 88 \\ 58 & 72 & 66 & 57 & 76 & 73 \end{bmatrix}$$

$m[\text{rix}][\text{cix}]$ is the entry at row rix and column cix . (Indices start from 0).

Matrices and 2-dimensional Arrays

$$m = \begin{bmatrix} 71 & 62 & 33 & 89 & 85 & 74 \\ 68 & 65 & 75 & 84 & 70 & 72 \\ 87 & 0 & 1 & 90 & 92 & 88 \\ 58 & 72 & 66 & 57 & 76 & 73 \end{bmatrix}$$

- A matrix is represented in algorithms by a 2-dimensional array, i.e., an array of arrays.
- **Exercise 9-1:** The matrix `m` is an array of 4 arrays, each with 6 members. If `m` is regarded as a **reference** to a 2-dimensional array, then

`m[1][2]` is

`m[2][5]` is

`m[4][1]` is

`m[3]` is



Exercise 9-2: Max value in a matrix (p. 1) ?

- Write an algorithm to find the maximum value in a matrix.

Max value in a matrix (p. 2)



BODY:

Exercise 9-2: Alternative Algorithm

BODY:

Diagonal Matrices

- A square matrix has the same number of rows and columns. If all its "off-diagonal" values are 0 it is a diagonal matrix.
- For example, in the following matrices,

$$m1 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad m2 = \begin{bmatrix} 2 & 4 & 0 \\ 3 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- $m1$ is a diagonal matrix and $m2$ is not a diagonal matrix.
- Write an algorithm that checks if a given square matrix is diagonal.

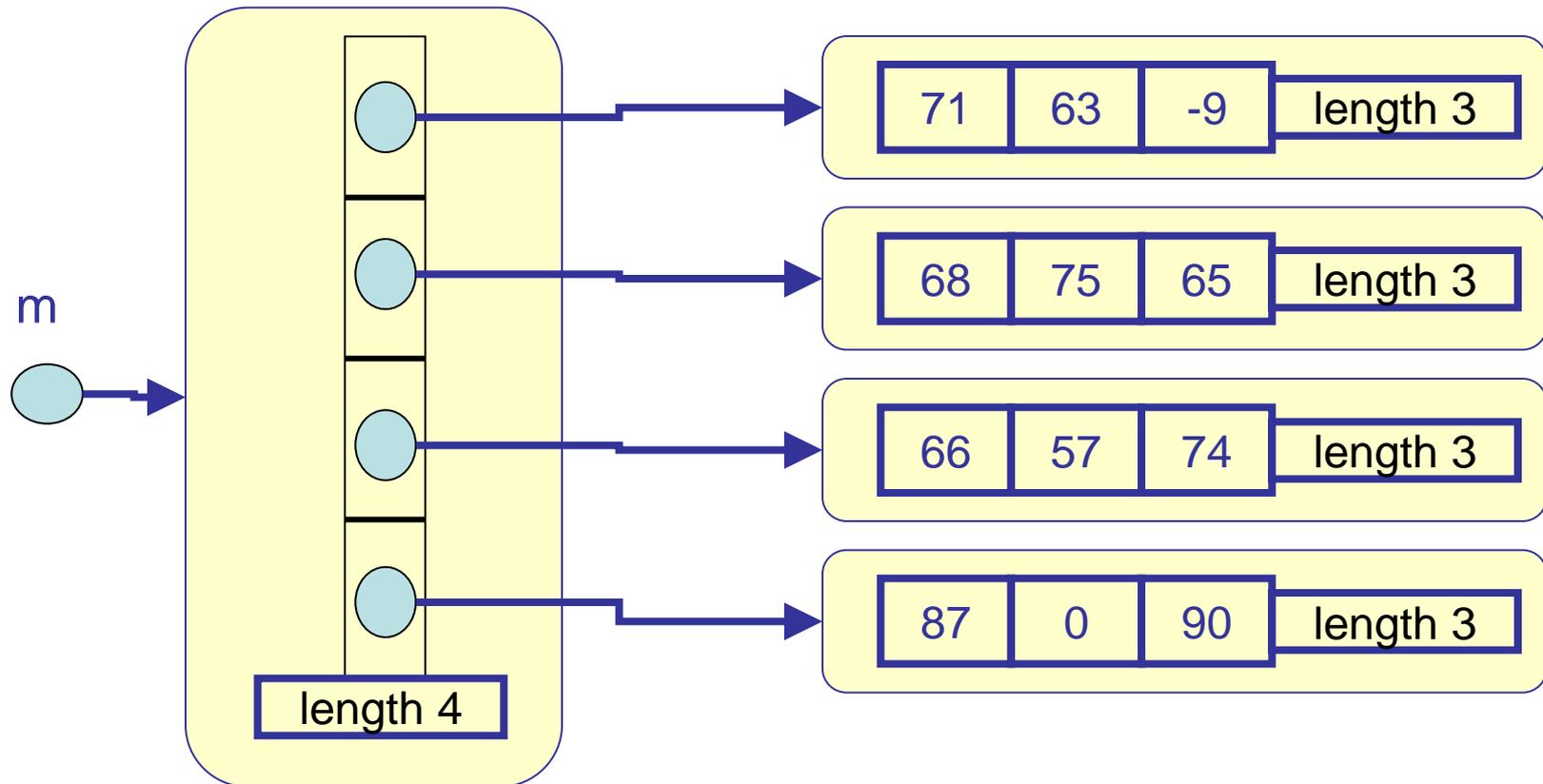
Exercise 9-3: Diagonal-check algorithm ?

Exercise 9-3: Efficient Version



2D Arrays in Java

- A 2D array in Java is literally an array of arrays (each entry in the array is a reference to an array).



Declaring a 2D Array

- To declare `m` to be a reference variable to a 2-dimensional array of integers:

```
int [][] m;
```

- To create a 2x3 instance of a 2D array (i.e. allocate memory for the array and all the sub-arrays) and assign its reference to `m` :

```
m = new int[2][3];
```

- To create an initialized 2 x 3 2-dimensional array :

```
int [][] m;
```

```
m = new int[][] { {1, 2, 3}, {4, 5, 6} };
```

```
int [][][] c = new int [][][] { {{1,2}, {3,4} },  
                                 {{5,6}, {7,8} } }
```

- You may use `length` to find the dimension of a 2-dimensional array, or any sub-array:

`m.length` is

`m[0].length` is

What about `m[0][0].length`?

Exercise 9-3: Max value in a matrix in Java ?

- Translate the algorithm for the maximum value in a matrix to Java:
 - Note: **Integer.MIN_VALUE** is the most negative allowable integer for a Java **int**, and can be used for $-\infty$.

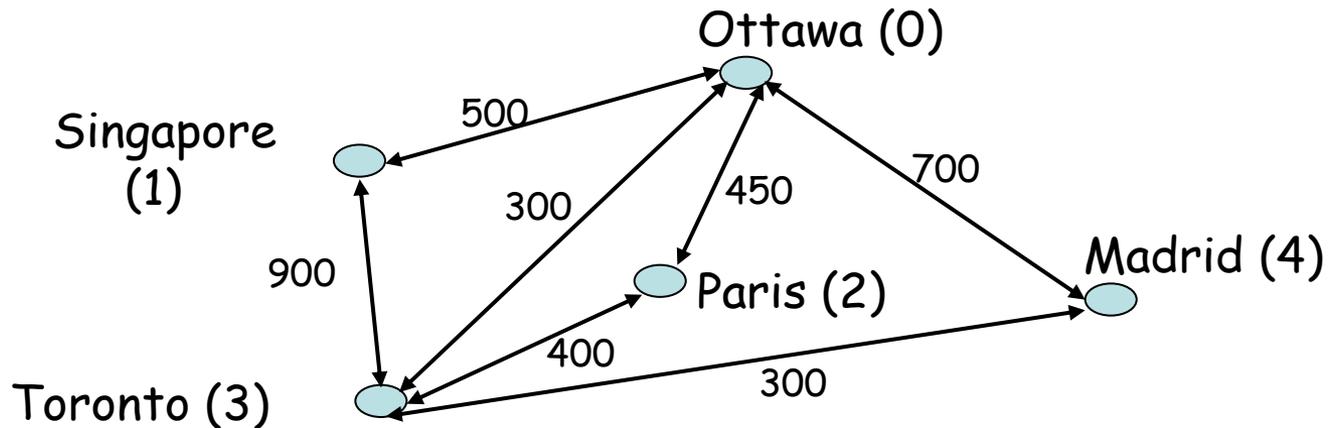
Exercise 9-4: Reading in a Matrix



- Write Java code to read in a matrix row by row (first it reads in the number of rows and columns, then it asks for the values in row 0, then it asks for the values in row 1, etc.). All values are read one per line using `ITI1120.readInt()`.

Adjacency Matrix

- Escape Airlines has flights between certain cities. The flights and their costs can be represented as a graph in which an edge between city x and city y with a weight (label) of w means Escape Airlines has a flight between x and y costing w dollars.



Matrix Representation

- This graph can be represented with an adjacency matrix. There is a row and a column for each city, and $\text{cost}[x][y]$ is the cost of a flight from x to y if one exists and is infinity (∞) if there is no such flight.

$$\text{cost} = \begin{bmatrix} 0 & 500 & 450 & 300 & 700 \\ 500 & 0 & \infty & 900 & \infty \\ 450 & \infty & 0 & 400 & \infty \\ 300 & 900 & 400 & 0 & 300 \\ 700 & \infty & \infty & 300 & 0 \end{bmatrix}$$

- Here, "infinity" is actually a very large number, greater than any number.
 - In Java: a predefined constant is available for the largest possible integer: **Integer.MAX_VALUE**

Finding the Cheapest Direct Flight

- Suppose you live in one of Escape's cities and have \$ d to spend. Write an algorithm that returns an array of the cities you can afford to fly to **directly**.
- What you know (Givens)
 1. The city where you live.
 2. The cost of flight between two cities.
 3. The total number of cities.
 4. The amount you can spend.
- What you want (Result)
 - An **array** of cities that can be visited.
- Idea:
 - First, find the number of cities that can be visited.
 - Then, create an array of the right size.
 - Finally, place cities that can be visited in the array.

Exercise 9-5: Find Cheap Direct Flights (p. 1) ?

Find Cheap Direct Flights (p. 2)



Exercise 9-5: Translate to Java (p. 1) ?

Exercise 9-5: Translate to Java (p. 2)

Alternative Solution

- To simplify the problem, we could use separate algorithms to find first the number of cities that can be visited and then fill in the array.
 - Algorithm `findNumber`
 - Find number of cities that can be visited
 - Algorithm `findCities`
 - Find the cities that can be visited
- Algorithms only require one row of the cost matrix; the one corresponding to the city where you live.
- New main body:

```
numCities ← findNumber(cost[home], d, n)
cities ← makeNewArray(numCities)
findCities(cost[home], d, n, cities)
```

Deleting rows and columns

- Escape Airlines has decided to stop flying to and from city x (e.g. Paris, $x=2$), and the city numbers greater than x have all been reduced by 1 (e.g. Madrid is now city 3).
 - This problem can be solved by deleting a row and a column that correspond to the city.
- Write two algorithms, one to delete a row from a matrix, the other to delete a column.
- We are not going to change the size of the matrix, but we are going to shift the rows and columns up and put zeros in the last row and column.

Delete a row

- **Idea:**

- To delete row i of a **square** matrix, we shift rows from bottom up and put zeros in the last row.
- Shall develop a separate algorithm to copy elements of one row to the row above it, overwriting its values.
 - Thus our first algorithm only needs a a loop that copies row $i + 1$ into row i , row $i + 2$ into row $i + 1$, ..., row $n-1$ into row $n - 2$.
- We use a separate algorithm to put zeros in the elements of the last row.

Exercise 9-6: Algorithm deleteRow (1) ?

Exercise 9-6: Algorithm deleteRow (2) ?

Exercice 9-6: Algorithme moveUp



Exercice 9-6: Algorithm putRowToZero ?

Exercise 9-6: Translation to Java



Delete a column

- Delete a column of a matrix. Use a similar approach. To do, as an exercise!
- Another possible exercise is to generalise the problem: Delete a row or column from a matrix which is not necessarily square.

"Politics is the skilled use of blunt objects."
-- L.B. Pearson

Section 10: Introduction to Objects

Objectives:

- Records
- Classes and Objects
- Information Hiding
- Accessors and Modifiers

Historical note ...

- [Barbara Liskov](#) was the first woman to have obtained her Ph.D. in computer science in US (in 1968, from Stanford University).
- She was at the origin of [CLU](#), the first language that supported abstract data types (1975), that influenced many object-oriented languages, including Java.
- In 1993, she and [Janette Wing](#) have developed a specific definition of sub-types, [the Liskov principle of substitution](#), used in object-oriented programming.



Student Information

- How can we store all the information about each student in a course?
 - ID (student number) (integer)
 - midterm mark (real)
 - final exam mark (real)
 - is taking this course for credit (Boolean)
- **Exercise 10-1:** What is the problem with the following solutions:
 - Each value is stored in a separate variable:

 - Put all the values into an array:



Records

- Like an array, a "record" allows several values **of different type** to be stored in one variable.
 - Another view is that a record is a group of variables of different type
- Records differ from arrays in 2 ways:
 - The values/variables (called fields) in a record can be of different types.
 - Each field in a record has a NAME. A value is accessed by specifying the field name (not a subscript).
- Example (a single record with 4 fields):

field name	field value
id	1234567
midterm	60.0
exam	80.0
forCredit	TRUE

Using Records

- Suppose the preceding record was stored in a variable named `r`.
- To access the midterm mark:
`r.midterm`
- This refers to one field inside record `r`. A field can be used anywhere a variable of that type is allowed, e.g.,
`t ← r.midterm + r.exam`
`r.forCredit ← false`
- The whole record can be used in an assignment statement or passed as a parameter:

`x ← r`

(not in Java - this occurs in other programming languages like C)

Defining a Record Type

- When we discussed primitive types, we looked at:
 - What values does the type allow?
 - What operations one can do with values of that type?
- A record is a "user-defined" type that is built using types we have already:
 - Primitive types
 - Other user-defined types.
- Creating a record also has the two elements of primitive types:
 - What are the components of a record value?
 - What operations can we do with the record?

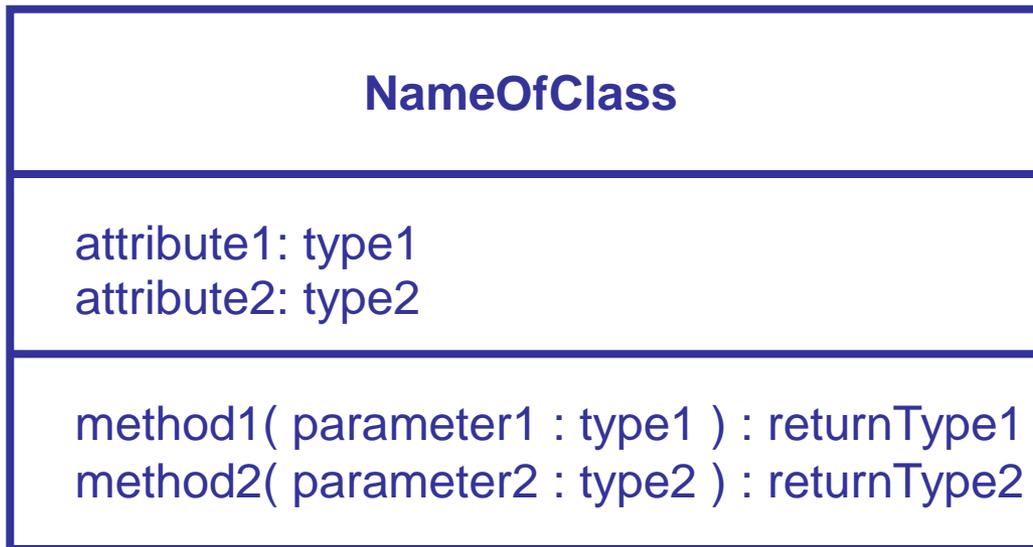
Records and Classes/Objects

- Some languages allow you to create records, without the capability to define operations on those records.
 - Examples:
 - The language Pascal has "records"
 - The language C has "structures"
- Some languages allow you to also define operations on user-defined data types.
 - The record types are then usually referred to as "classes", and specific records are called "objects"
 - Examples:
 - "Classes" in the language C++ and Java

Classes and Objects

- An object can be considered to be like a record, in that there is a set of "attributes" - named data values (variables) stored in the object.
- Each object is created from a class. An object is referenced from a reference variable (like an array).
- A "class" can be used as a template to create objects with identical sets of attributes.
 - The class can also contain methods (algorithm models) to perform calculations on the attributes of objects created from the class (and/or external data).
- A method is called on an object using the . operator, in a similar manner to accessing a record field
result ← anObject.aMethod(aParameter)

Class Diagrams



← "Attributes" are like the field variables in a record

- This form of diagram is from a notation called the "Unified Modelling Language" or UML

Translation to Java

```
public class <Class Name>
{
    // Declaration of Variables
    // public <type> <name>;

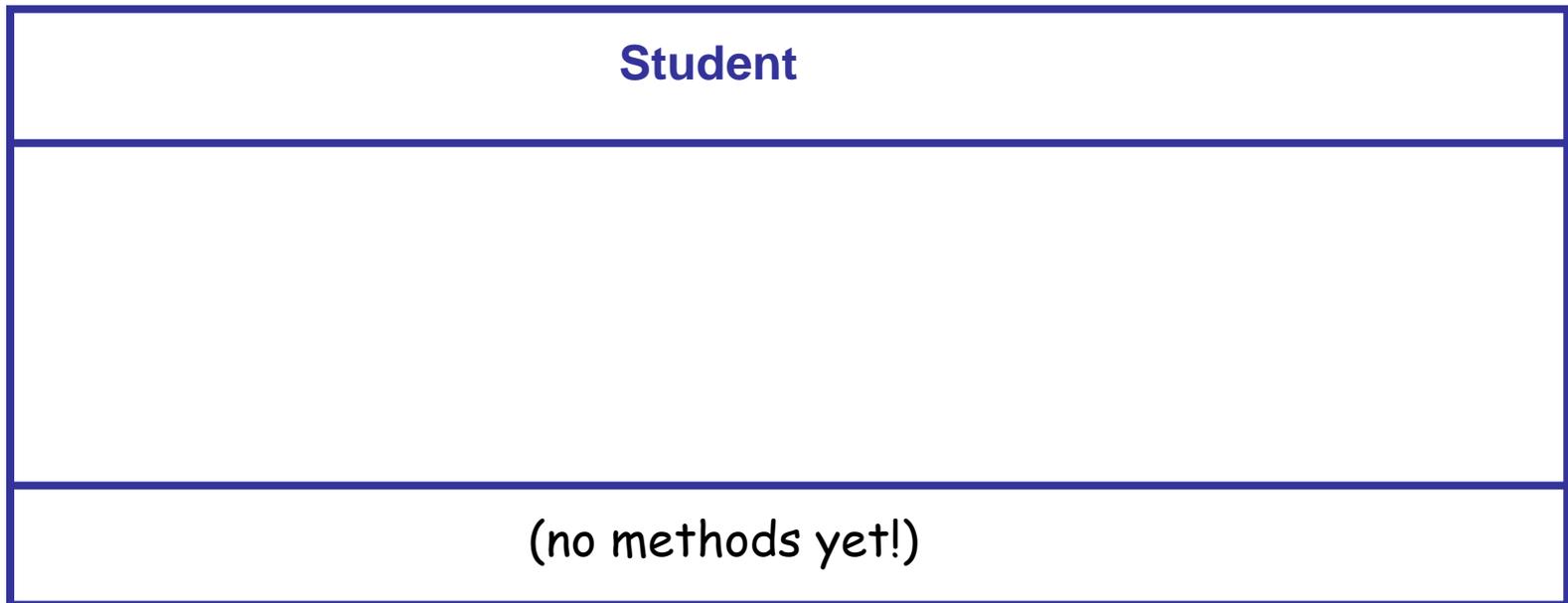
    // Methods
}
```

- The class is a "template" for how to construct objects.
 - Objects must be created using the **new** statement
 - Objects are referenced with **a reference variable**

```
// Declaring the reference variable
<Class Name> refVar;
// creating the object
aStudent = new <Class Name>( );
```

Exercise 10-2: First version of a Student Class ?

- For each student, we want to store their ID number, their midterm score, their exam score, and whether or not the student is taking the course for credit. *We shall deal with the final mark later.*



Exercise 10-2: Translation to Java

```
public class Student
{

    // methods
}

// Declare aStudent reference Variable

// Create a Student object referenced by aStudent
```

Exercise 10-3: Object usage in Java

```
Student aStudent;           // declare reference variable
aStudent = new Student();  // create new object
aStudent.id = 1234567;
aStudent.midterm = 60.0;
aStudent.exam = 80.0;
aStudent.forCredit = true;

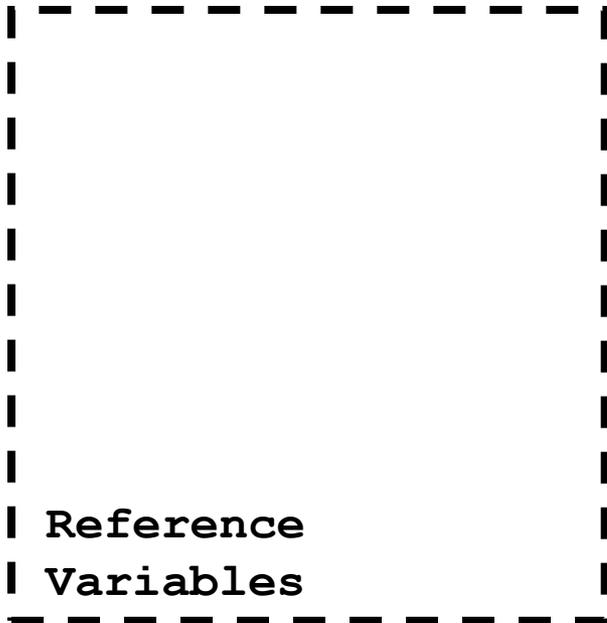
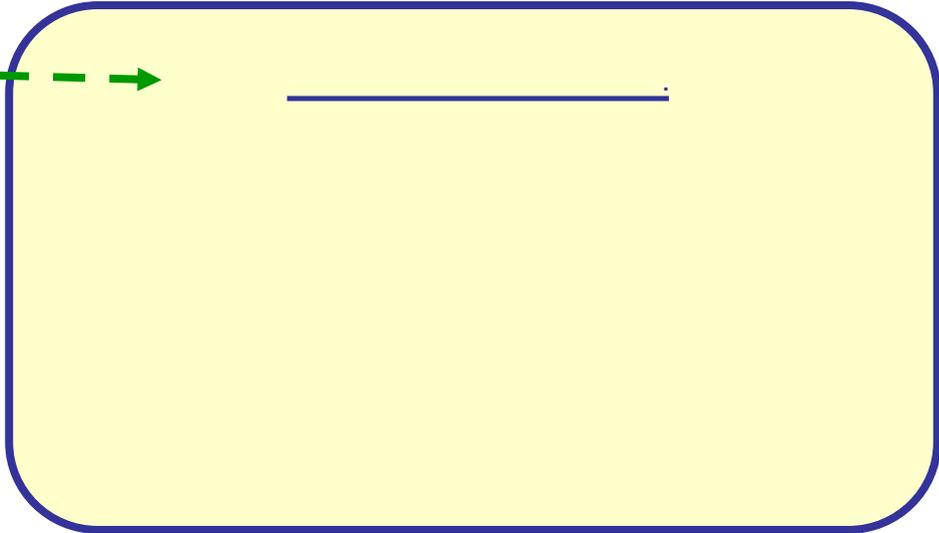
Student meToo;
meToo = new Student();
meToo.id = 81069665;
meToo.midterm = 73.0;
meToo.exam = 77.0;
meToo.forCredit = false;
```

Exercise 10-3: Object usage in Java



format:
<class name>

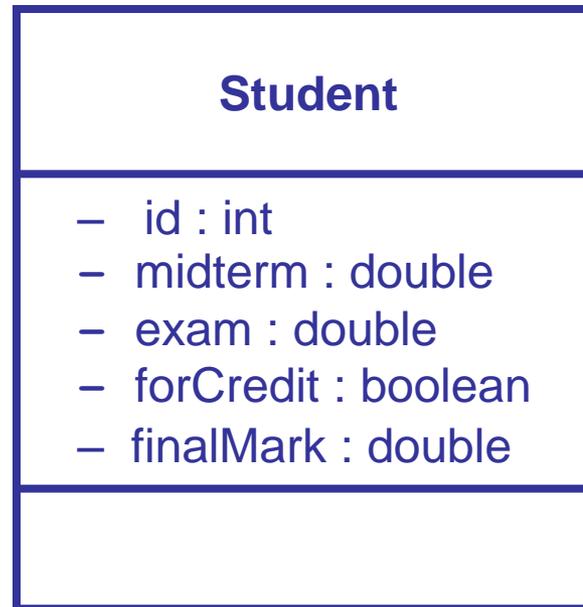
(the underlining shows
that this is an **instance**
diagram)



Information hiding

- Suppose we want to modify the `Student` class to keep the course final mark, which is 20% of the midterm mark plus 80% of the final mark.
 - We could add a field `finalMark` to our class.
- We want to make sure that
$$\text{finalMark} = (0.2 \times \text{midterm} + 0.8 \times \text{exam})$$
is always true for consistency.
- It would be useful to prevent anyone else from setting the value of `finalMark` arbitrarily.
 - Instead, if the final mark is to change, it should be done by changing the value of either `midterm` or `exam`.
- Restricting access to data is called "information hiding".

Private fields in a class



- The - in front of the variable indicates that the attribute is **private**.
- By declaring a field to be private, **only methods declared inside the class** are allowed access to the field value (either for viewing the value, or changing the value).

Information Hiding

- The field names and types represent an *implementation* of a class.
 - To ensure relative independence relative to other parts of your program (which helps reduce the effort of maintenance), fields are (almost) always **private**.
 - This **information hiding** is also called **data abstraction** and also **encapsulation**).
- The private fields and methods cannot be accessed directly but only from methods in the class.
- If (and only if) necessary, can define a few public methods to allow other parts of the program to access fields.
- The public methods represent the **interface** of the class relative to other parts of the program.

Second version of the Student Class

- This time, use encapsulation.

```
public class Student
{
    // were previously public
    private int id;
    private double midterm;
    private double exam;
    private boolean forCredit;
    private double finalMark;    // new field
    // methods
}
```

How do we use the second version?

- If we try the following:

```
Student aStudent = new Student();  
aStudent.id = 1234567 ; // error!
```

the compiler returns an error since access to id is no longer allowed from outside the class.

- However, we can create additional access methods in the class **Student**:
 - "accessor": requests to see the value of a private field.
 - "modifier": requests to modify the value of a private field.

Accessors and Modifiers

- **Accessor**
 - A public instance method (called using a reference to an object);
 - Returns the value of the field of the object;
 - Has no parameters;
 - Often called **getFieldName** (also called a getter method).

- **Modifier**
 - A public instance method (called using a reference to an object);
 - Assigns a value to a field;
 - Accepts values in a parameter of the same type as the field;
 - Often called **setFieldName** (also called setter method).

Accessors and Modifiers

- Examples for the `forCredit` field in the class:

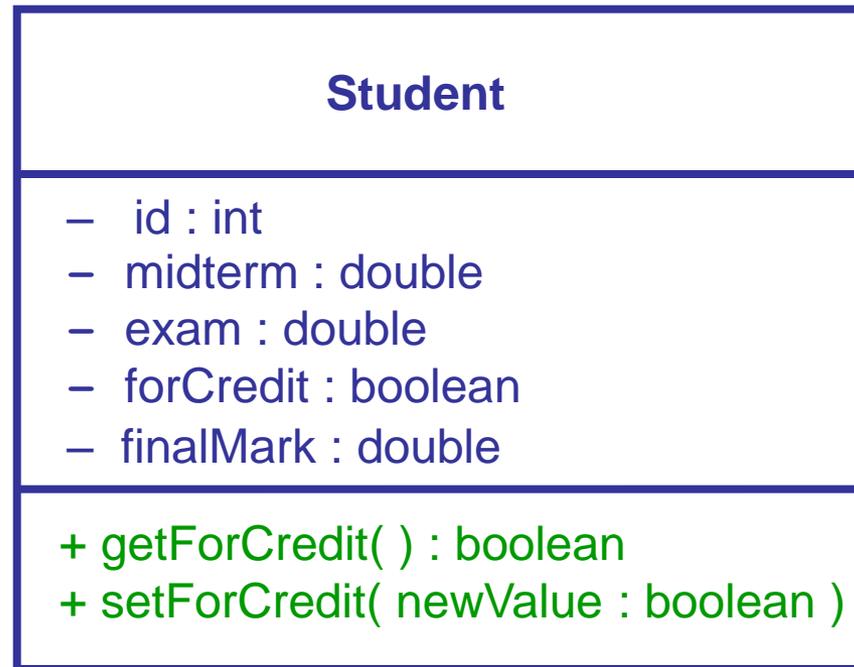
+ `getForCredit() : boolean`

- method to return the value of `forCredit`
- the `+` indicates that the method has `public` visibility
- the return type is `boolean`, and in UML notation, appears at the end of the method.

+ `setForCredit(newValue : boolean)`

- method to change the value of `forCredit`
- one parameter `newValue`, of type `boolean`
- `no` return value

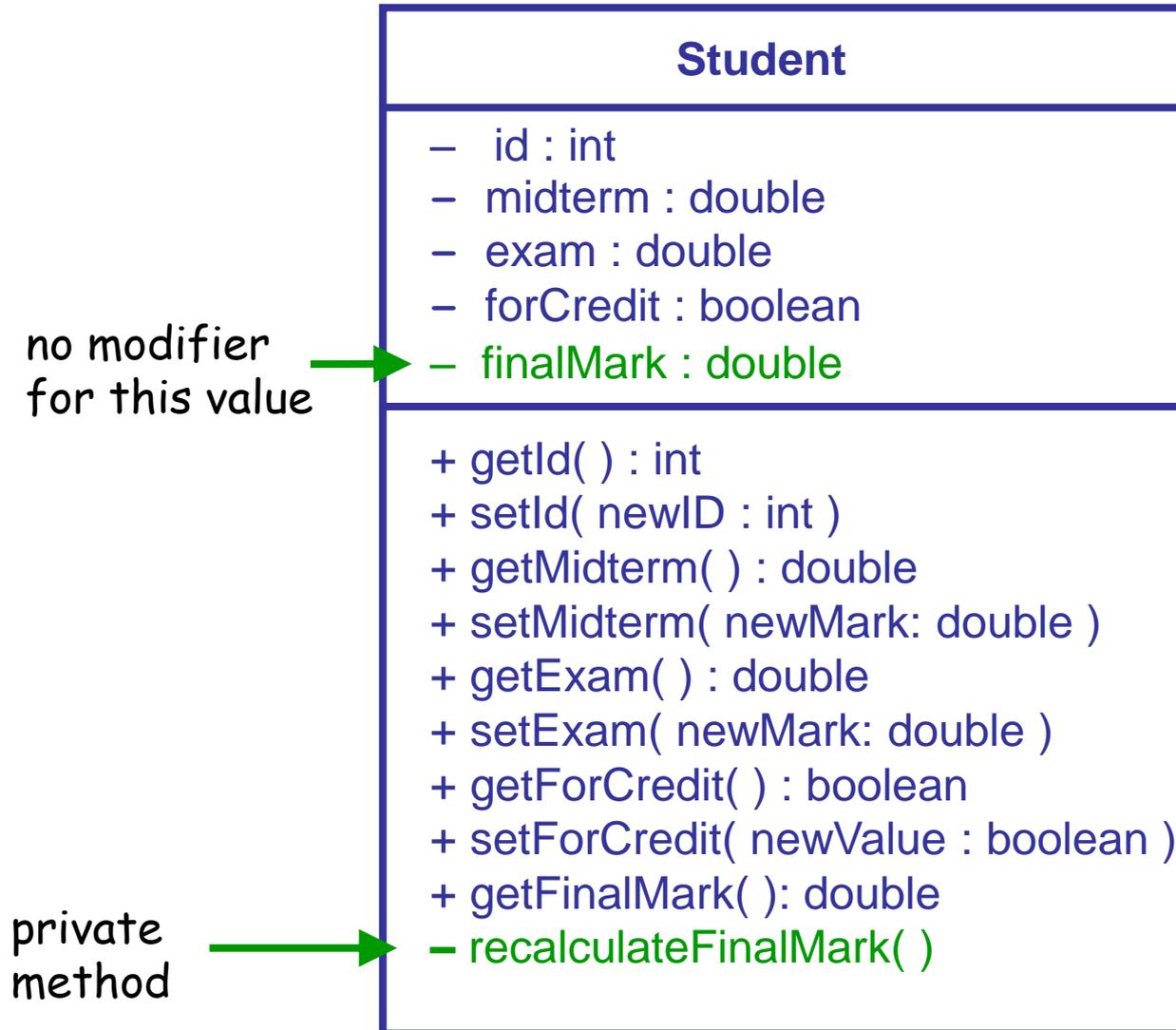
Class diagram with accessors and modifiers



Back to Information Hiding

- To implement our strategy of hiding the `finalMark` field, we can do the following:
 - We will provide an accessor method for `finalMark`, but **NOT** a modifier method.
 - We can provide a method `recalculateFinalMark()` to recalculate the final mark if the midterm or exam marks are changed.
 - The modifier methods `setMidterm()` and `setExam()` will call `recalculateFinalMark()` so that they automatically update the final mark.
- We should also restrict access to `recalculateFinalMark()` because it isn't meant for use outside the class.

Student class with Information Hiding



Translation to Java

```
public class Student
{
    // Attributes

    private int id;
    private double midterm;
    private double exam;
    private boolean forCredit;
    private double finalMark;

    // Methods

    public int getId()
    {
        // insert code here
    }
    public void setId( int newId )
    {
        // insert code here
    }
    public double getMidterm()
    {
        // insert code here
    }
    public void setMidterm( double newMark )
    {
        // insert code here
    }
    // continued at right
```

```
// continued from left side

    public double getExam()
    {
        // insert code here
    }
    public void setExam( double newMark )
    {
        // insert code here
    }
    public boolean getForCredit()
    {
        // insert code here
    }
    public void setForCredit( boolean newValue )
    {
        // insert code here
    }
    public double getFinalMark()
    {
        // insert code here
    }

    private void recalculateFinalMark()
    {
        // insert code here
    }
} // end of class Student
```

Calling Java Accessor and Modifier Methods

- Again, use the dot operator (.)
- The following code causes errors, why?

```
Student aStudent = new Student();  
aStudent.id = 1234567;           // error here!  
int myId = aStudent.id;         // error here!  
System.out.println( myId );
```

- The compiler enforces the private access to `id`.
- Solution: Instead, use the modifier and accessor methods.

```
Student aStudent = new Student();  
aStudent.setId( 1234567 );      //ok!  
int myId = aStudent.getId( );  //ok!  
System.out.println( myId );
```

Implementing Java accessors and modifiers

```
public class Student // not all attributes/methods shown!
{
    // attribute
    boolean forCredit;
    // ... other attributes declared

    // accessor: return the requested value
    public boolean getForCredit()
    {
        return this.forCredit ;
    }

    // modifier: save the requested value in object's
    // attribute

    public void setForCredit( boolean newValue )
    {
        this.forCredit = newValue;
    }
    // ...other methods are similar
}
```

Where did **this** come from?

- When the fields of our Student class were public, we distinguished between the same field in two record objects with the variable name and the dot operator:
 - **aStudent.forCredit** versus **meToo.forCredit**
- Likewise, when a method **inside the class** wants to work with “the value of the field for the object on which I was called”, **this** refers to the called object.
- During the call **aStudent.getForCredit()**, **this** is a reference to **aStudent**
 - ... and so **this.forCredit** is **aStudent.forCredit**, which is true.
- During the call **meToo.getForCredit()**, **this** is a reference to **meToo**
 - ... and so **this.forCredit** is **meToo.forCredit**, which is false.

Methods Operate on Object Data

- Adding a method to a class such as Student provides operations on the data in the objects created with the class (user defined types).
- Note that we have called the method `getForCredit` without parameters in two different cases
 - Each different case calls to the methods are associated to different objects and thus we get different results.
- An analogie:
 - With integers: $3 + 5$ and $4 + 3$; the same operation (+) is invoked, but with different values which gives different results
 - With Students: the same operation (`getForCredit`), but on different objects \Rightarrow different results

Implementing Information Hiding

- The following implements our strategy where the final mark can only be changed by modifying the midterm or exam values.

```
public class Student
{
    // attributes and other methods would go here
    public void setMidterm( double newValue )
    {
        this.midterm = newValue;
        this.recalculateFinalMark( );
    }

    public void setExam( double newValue )
    {
        this.exam = newValue;
        this.recalculateFinalMark( );
    }

    private void recalculateFinalMark()
    {
        this.finalMark = 0.2 * this.midterm + 0.8 * this.exam;
    }
}
```

Benefits of Information Hiding (1)

- One of the most common causes of problems historically has been when all parts of a program have access to all program variables.
 - For example, when someone makes a change to a large program, the new code may make changes to data that some other part of the program assumed would not be modified.

"Successful software always gets changed." - *F. Brooks*

- With information hiding, we can keep the code better partitioned so that changes will be less likely to cause unwanted side effects.

Benefits of Information Hiding (2)

- We can also make changes inside a class that will not affect users of the class.
- Example: Suppose we decide that the `finalMark` field really doesn't need to be stored in the `Student` class.
 - Instead, we can calculate the final mark when anyone asks for it:

```
public double getFinalMark()  
{  
    return 0.2 * this.midterm + 0.8 * this.exam;  
}
```
 - This means we can remove the method `recalculateFinalMark()`, and the calls to it in `setMidterm()` and `setFinal()`.
- Making these changes will not affect any user of the class:
 - For example, `meToo.getFinalMark()` still behaves as it did before.
 - Since `recalculateFinalMark()` was private, code outside the class was not able to call this method, and therefore it can be safely removed.
- So we don't have to change any code outside the class!

Compare Versions

Student

- id : int
- midterm : double
- exam : double
- forCredit : boolean
- finalMark : double

- + getId() : int
- + setId(newID : int)
- + getMidterm() : double
- + setMidterm(newMark: double)
- + getExam() : double
- + setExam(newMark: double)
- + getForCredit() : boolean
- + setForCredit(newValue : boolean)
- + getFinalMark(): double
- recalculateFinalMark()

Student

- id : int
- midterm : double
- exam : double
- forCredit : boolean

- + getId() : int
- + setId(newID : int)
- + getMidterm() : double
- + setMidterm(newMark: double)
- + getExam() : double
- + setExam(newMark: double)
- + getForCredit() : boolean
- + setForCredit(newValue : boolean)
- + getFinalMark(): double

this, again

- In most cases, we don't actually have to use **this** to refer to the object on which a method is called.
 - Inside the Student class:
 - **exam** can be used instead of **this.exam**.
 - **recalculateFinalMark()** can be used instead of **this.recalculateFinalMark()**.
- There are 2 occasions when we really do need **this**:
 1. An object wants to pass itself as a parameter to a method of another class.
 2. An object wants to return a reference to itself as the result of a method.

"A class, in Java, is where we teach objects how to behave."
-- R. Pattis

Section 11: Object-oriented design

Objectives:

- Constructors
- Array fields in classes
- Classes versus instances
- Class design

Historical note ...



- The Xerox Palo Alto Research Center (PARC), founded in 1970, is at the origin of many important contributions:
 - The first workstation (Alto) with graphical user interface (GUI, with windows and icons) and mouse
 - The first text editor WYSIWYG
 - The InterPress language (predecessor of PostScript) for describing pages to be printed
 - The Ethernet protocol for local area networks
 - The Smalltalk object-oriented programming language, with graphical development environment (designed by Alan Kay)
 - The laser printer
 - ...

Object-orientation

- The approach we have taken with our student class is an “object oriented” approach:
 - We have a class that is a template for the creation of objects.
 - Student objects can be referred to as **INSTANCES** of the **CLASS** Student.
 - Object instances have **instance methods** that use the field values for a specific object
 - e.g. `getFinalMark()` will have different results for different objects because `this.exam` is different for different objects
 - If you want the object to do something for you, you have to ask it by calling a method on that object.
 - That is, you can't sneak inside an object from outside the class and change the private field values.
 - You also can't call an instance method, without using an object reference:
 - `x ← 3.0 + getFinalMark()` is meaningless. Whose final mark are we referring to?

Initialization of Objects

- When we create a new **Student**, we will usually want to provide values for all the fields in the object.

```
aStudent = new Student( );  
aStudent.id = 1234567;  
aStudent.midterm = 60.0;  
aStudent.exam = 80.0;  
aStudent.forCredit = true;
```

- A special kind of method called a **CONSTRUCTOR**, can be used to initialize values inside an object as the object is being created.

```
aStudent = new Student( 1234567, 60,0, 80.0, true);
```

Constructors

- A constructor is a special method in a class used to create an object.
 - the name of the method is the same as the class;
 - no return type
 - usually public;
 - may or may not have parameters.
- The parameters, if any, in a constructor are used to initialize the values of the object.
- Because there may be different ways to initialize an object, a class may have any number of constructors, distinguished from each other by **different parameter lists**.

Implementation in Java

- The following is a constructor that sets a value for all of the fields in the **Student**:

```
class Student
{
    // ... fields would be defined here ...
    public Student(int theId, double theMidterm, double theExam, boolean
                    isForCredit)
    {
        this.id = theId;
        this.midterm = theMidterm;
        this.exam = theExam;
        this.forCredit = isForCredit;
    }
    // ... Other methods ...
}
```

- This constructor could be used as follows:

```
Student aStudent = new Student(1234567, 60.0, 80.0, true);
```

Constructors of class Student

- If we are doing course registrations, we may only want to provide the ID number and whether the student is taking the course for credit. (We don't know the student's marks yet!)
- We could **also** provide the following constructor:

+ Student(theID : int, isForCredit : boolean) *UML*

```
public Student(int theID, boolean isForCredit )
{
    this.id = theID;
    this.midterm = 0.0;           // a "safe" value
    this.exam = 0.0;            // a "safe" value
    this.forCredit = isForCredit;
}
```

- When there is more than one constructor, they must have parameter lists that can be distinguished by the **number**, **order**, and **type** of parameters.

Add Constructors to the Class

Student

- id : int
- midterm : double
- exam : double
- forCredit : boolean

- + Student(theID : int, theMidterm : double, theExam : double, isForCredit: boolean)
- + Student(theID : int, isForCredit: boolean)

- + getId() : int
- + setId(newID : int)
- + getMidterm() : double
- + setMidterm(newMark: double)
- + getExam() : double
- + setExam(newMark: double)
- + getForCredit() : boolean
- + setForCredit(newValue : boolean)
- + getFinalMark(): double

Constructors of class Student

- Here is a constructor with no parameters:

+ Student()

```
public Student( )
{
    this.id = 0;
    this.midterm = 0.0;
    this.exam = 0.0 ;
    this.forCredit = false;
}
```

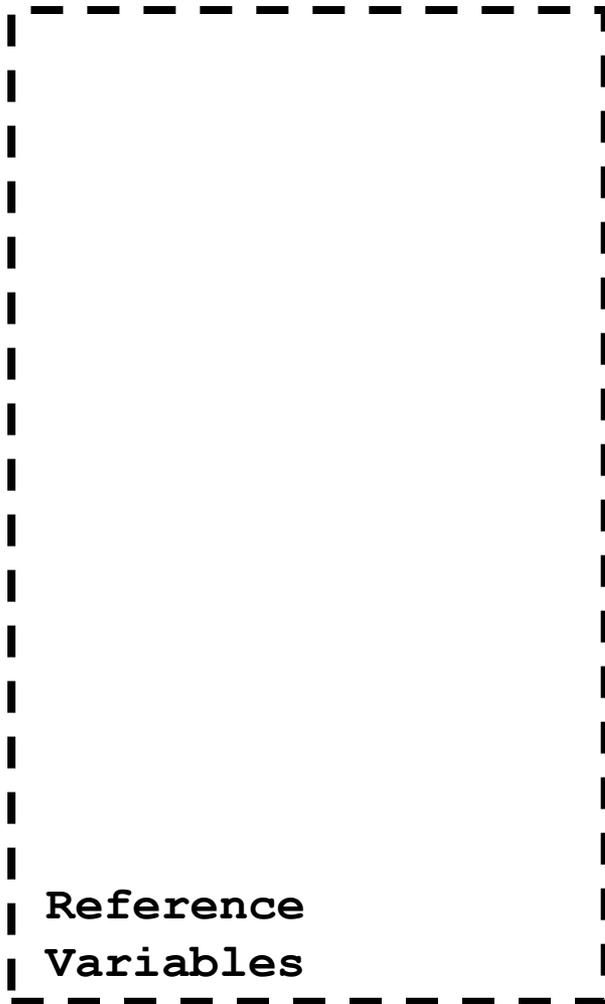
- A constructor without parameters is called a **default constructor**.
 - It is recommended to always define a default constructor that sets every field to a "safe" value.
- If, and only if, a class does not define any constructor, the Java compiler invisibly creates a default constructor that does nothing.

Exercise 11-1: Using Constructors

- Show how objects are created and referenced by the following main method.

```
public class Section11
{
    public static void main(String [] args)
    {
        Student aStudent; // reference variable
        Student meToo;    // another reference variable
        Student bStudent; // a third reference variable
            •
            •
        aStudent = new Student(1234567,60.0,80.0,true);
        meToo = new Student(7654321,true);
        bStudent = aStudent;
            •
            •
            •
    }
}
```

Exercise 11-1: Using Constructors



Array Fields in Classes

Student

- id : int
- midterm : double
- exam : double
- forCredit : boolean
- assignments : double[]

+ Student(theID : int, theMidterm : double,
 theExam : double, isForCredit: boolean)

+ Student(theID : int, isForCredit: boolean)

+ getId() : int

+ setId(newID : int)

+ getMidterm() : double

+ setMidterm(newMark: double)

+ getExam() : double

+ setExam(newMark: double)

+ getForCredit() : boolean

+ setForCredit(newValue : boolean)

+ getFinalMark() : double

- A field of a class may have any type. In particular, a class may have a field of an array type (**reference variable**).
- Add an array of double to class **Student** representing assignment marks:
- Remember, **arrays are not created automatically!** The array **assignments** will have to be created after a **Student** object is created. 350

Array Fields in Classes

```
public class Student
{
    private int id ;
    private double midterm ;
    private double exam ;
    private boolean forCredit;
    private double[] assignments;

    // methods
}
```

- The array reference variable `assignments` contains the value `null`.

Array field initialization

- Here is a constructor that creates and initializes an array in an object. The constructor has a parameter that is the number of assignments.

```
public Student( int numberOfAssignments )
{
    this.id = 0;
    this.midterm = 0.0;
    this.exam = 0.0 ;
    this.forCredit = false;
    this.assignments = new double[numberOfAssignments];

    // loop to initialize each item in array
    int index;
    for ( index=0; index < numberOfAssignments; index = index+1 )
    {
        this.assignments[index] = 0.0;
    }
}
```

Accessors for an Array Field

- An accessor for an array field could:
 - Return a reference to the entire array
+ `getAssignments() : double[]`

UML

```
public double [] getAssignments()  
{  
    return assignments;  
}
```

- Return one of the values in the array, with an extra parameter to select the array index.
+ `getAssignment (assignNumber: int) : double`

UML

```
public double getAssignments( int assignNumber )  
{  
    return assignments[assignNumber];  
}
```

- Which approach is better?

Exercise 11-2: Calculation of the Final Mark ?

- Write Java methods `calcAssignAvg()` and `getFinalMark()` for our `Student` class that returns a `double` with a student's final mark, where:
 - The final mark is 55% of the exam, plus 20% of the midterm, plus 25% of the average of 5 assignments.

```
public double calcAssignAvg()  
{
```

```
}
```

Exercise 11-2: Calculation of the Final Mark

- Write Java methods `calcAssignAvg()` and `getFinalMark()` for our `Student` class that returns a `double` with a student's final mark, where:
 - The final mark is 55% of the exam, plus 20% of the midterm, plus 25% of the average of 5 assignments.

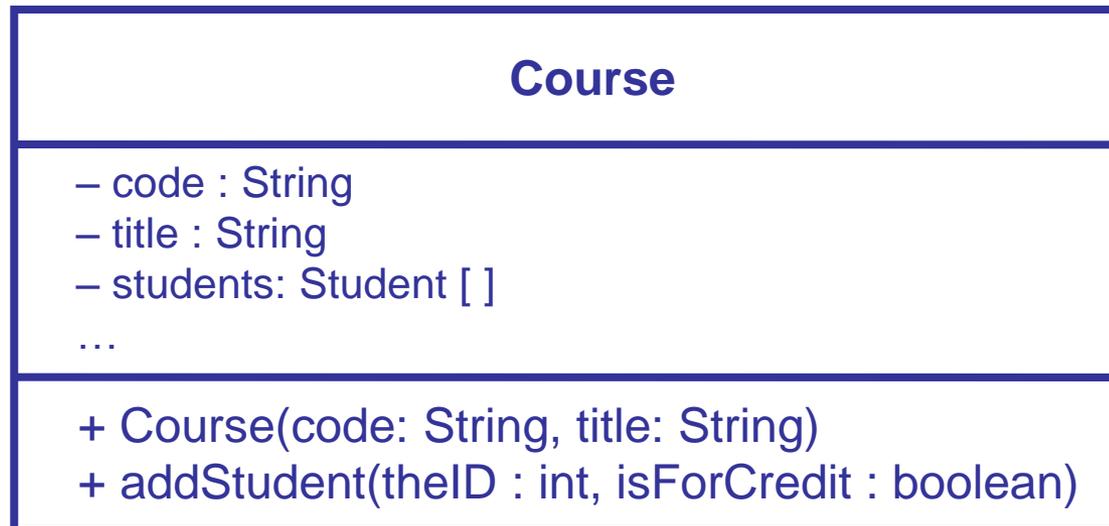
```
public double getFinalMark()
```

```
{
```

```
}
```

Course information

- Now that we have a class that stores information about one **Student**, how can we use this create a class **Course** that stores information about all students?



Exercise 11-3: Arrays of objects

- Show how objects are created and referenced by the following **main** method.

```
public class Section11
{
    public static void main(String [] args)
    {
        Course aCourse;
        aCourse = new Course("ITI1120","Intro. to Comp.");
        aCourse.addStudent(123456,true);
        aCourse.addStudent(654321,false);
    }
}
```

Exercise 11-3: Array of Student Objects ?



Common values for a Class

- For our student and course objects, we have been using **instance variables**, and creating **instance methods**.
 - There is a set of instance variables created for each new object.
 - The instance methods use the instance variables for the object on which they are called.
- Suppose we have a value that we want **every** student object to know about.
 - Examples: The weights of the final exam, midterm, and assignments for calculating a student's final mark.

Adding weights as attributes

Student

- assignWeight : double = 0.25
- midtermWeight : double = 0.20
- examWeight : double = 0.55
- id : int
- midterm : double
- exam : double
- forCredit : boolean
- assignments : double []

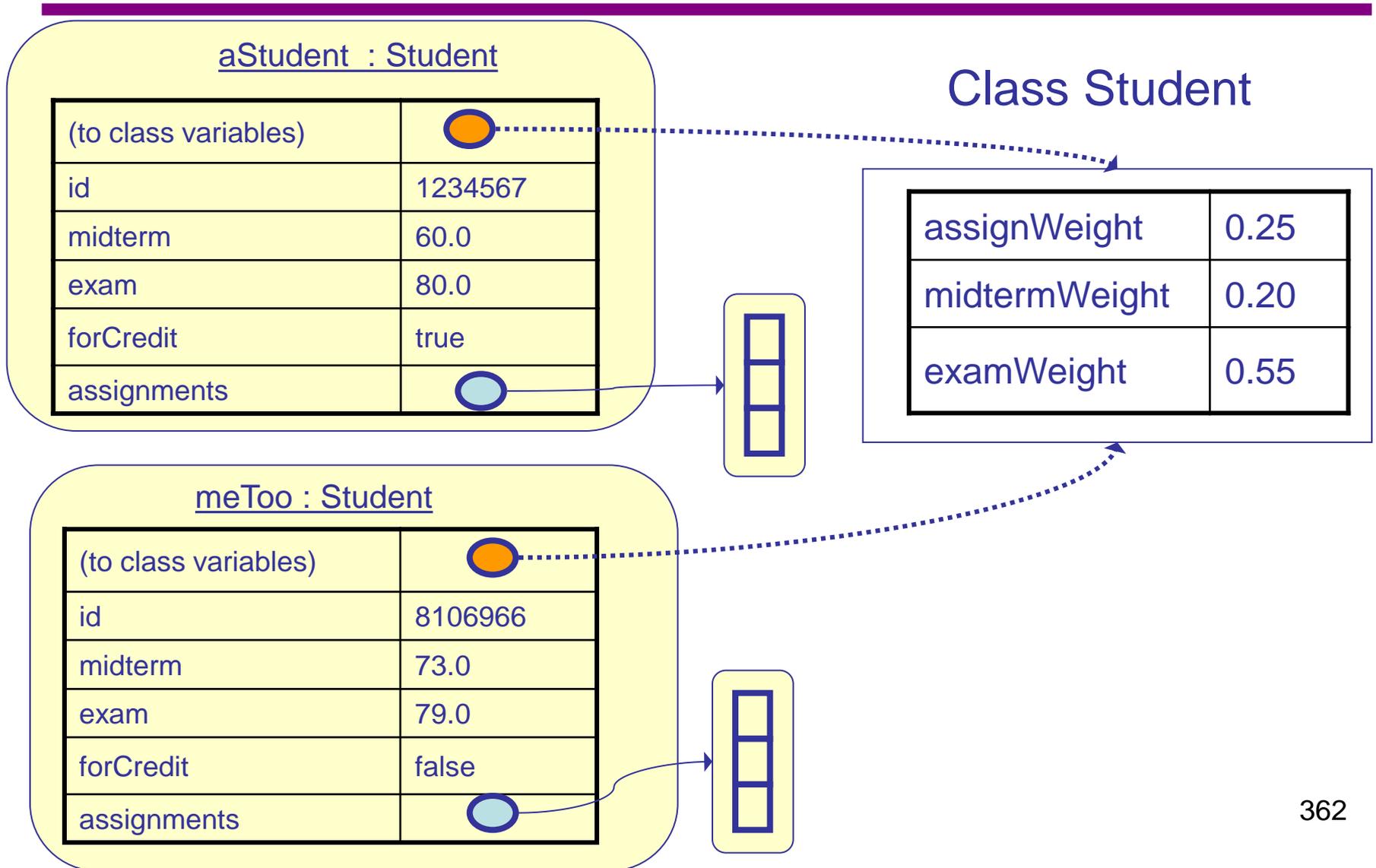
```
+ Student( theID : int, theMidterm : double,
           theExam : double, isForCredit: boolean )
+ Student( theID : int, isForCredit: boolean )
+ getId( ) : int
+ setId( newID : int )
+ getAssignment( assignNum : int ) : double
+ setAssignment( assignNum : int,
                 newMark : double )
+ getMidterm( ) : double
+ setMidterm( newMark: double )
+ getExam( ) : double
+ setExam( newMark: double )
+ getForCredit( ) : boolean
+ setForCredit( newValue : boolean )
+ getFinalMark( ) : double
- calculateAssignAverage( ) : double
```

- The problem with this approach is that **EVERY** Student object will have copies of assignWeight, midtermWeight, and examWeight, which takes up extra storage for each student
 - How much extra storage would this take if there were one object for every ITI 1120 student?
- If the weights change, every object would have to be updated.
- We want the weights to be consistent among all Student objects.

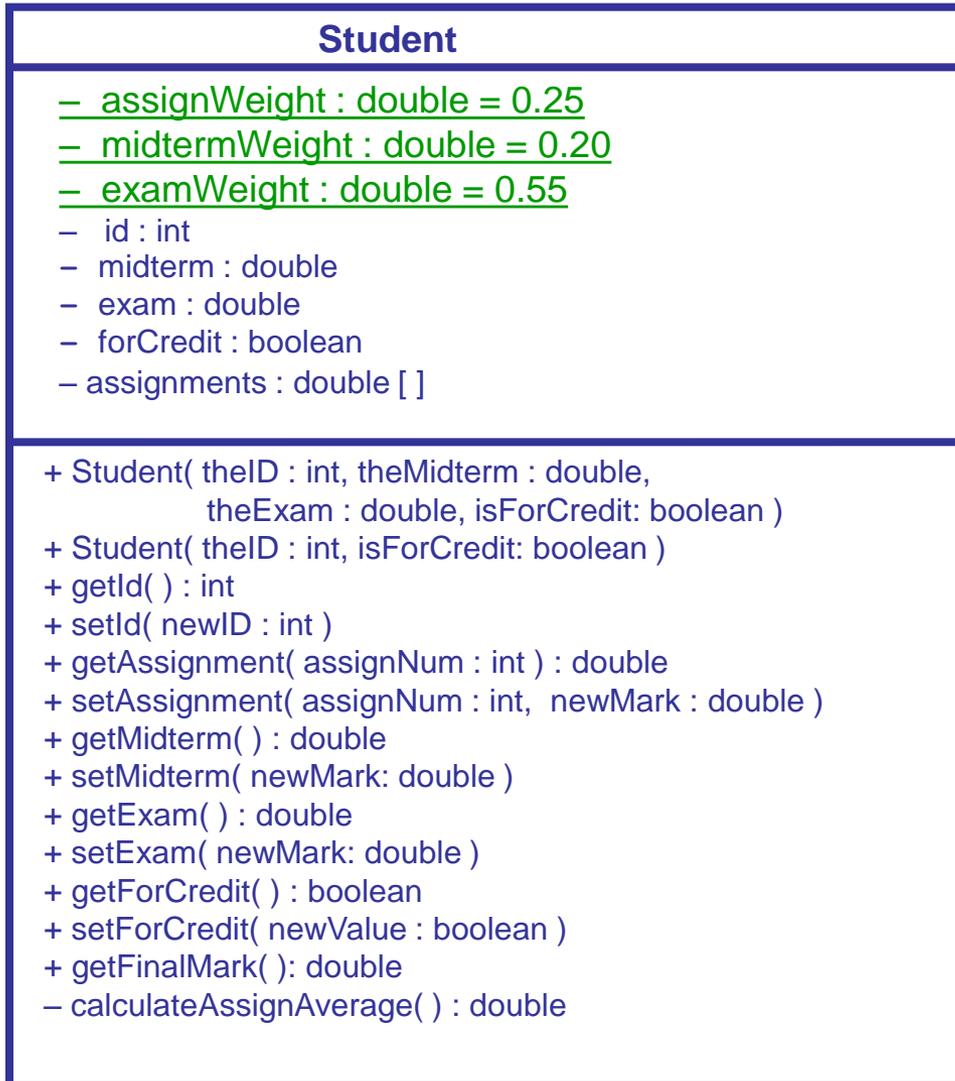
Class Variables

- Another type of variable we can have with a class is a “class variable” (also known as “static variable”).
- Class variables are NOT stored inside individual objects, because they belong to the entire class.
 - Instead, they are stored with the class.
- EVERY object created from the class has access to the class variables, even if they are private.
 - If class variables are public, then methods outside the class also have access to the class variables.

Class and Instance variables



Class variables in UML diagrams



- Class variables are underlined; instance variables are not.
- Note that the initial values of the class variables has been specified.

Translation to Java

```
public class Student
{
    // Class variables                                (applies to all students)
    static private double assignWeight = 0.25;
    static private double midtermWeight = 0.20;
    static private double examWeight = 0.55;

    // Instance variables                            (one copy per student object)
    private int id ;
    private double midterm ;
    private double exam ;
    private boolean forCredit;
    private double[] assignments;

    public double getFinalMark()
    {
        double assignAvg = this.calculateAssignAverage( );
        double finalMark = Student.midtermWeight * this.midterm
            + Student.examWeight * this.exam
            + Student.assignWeight * this.assignAvg;
        return finalMark;
    }
}
```

Class Methods

- Now we can change the final mark calculation for all students by changing the value of the weights.
- We want to write a modifier method for each weight.
 - This modifier should be a **CLASS** method, because the values are associated with the class instead of the objects.
 - We can also set the weights before creating any student objects.
- A class method is called as follows:
`ClassName . aMethodName()`
 - Just like the **Math** class!
- A class method **CANNOT** use any instance variables, because it is not associated with any particular object.

Class methods in UML diagrams

Student

- assignWeight : double = 0.25
- midtermWeight : double = 0.20
- examWeight : double = 0.55
- id : int
- midterm : double
- exam : double
- forCredit : boolean
- assignments : double []

+ getAssignWeight () : double
+ setAssignWeight (newWeight : double)
+ getMidtermWeight () : double
+ setMidtermWeight (newWeight : double)
+ getExamWeight () : double
+ setExamWeight (newWeight : double)
+ getId () : int
+ setId(newID : int)
+ getAssignment(assignNum : int) : double
+ setAssignment(assignNum : int, newMark : double)
+ getMidterm () : double
+ setMidterm(newMark: double)
+ getExam () : double
+ setExam(newMark: double)
+ getForCredit () : boolean
+ setForCredit(newValue : boolean)
+ getFinalMark () : double
- calculateAssignAverage () : double

- Class methods are underlined; instance methods are not.

static methods in Java

- As with class variables, class methods are indicated by the keyword **static**.
 - Note that **main** is always a class method.
- Write Java class methods for **Student** to set the values of the weights for the assignments, midterm, and final exam

```
public static void setExamWeight( double newWeight )
{
    Student.examWeight = newWeight ;
}
public static void setMidtermWeight( double newWeight )
{
    Student.midtermWeight = newWeight ;
}
public static void setAssignmentWeight( double newWeight )
{
    Student.assignWeight = newWeight ;
}
```

Exercise 11-4 - Using Class Variables and Methods

- What output would be produced by the following `main` method.

```
public class Section11
{
    public static void main(String [] args)
    {
        int anum;
        Student aStudent; // reference variable
        Student meToo;    // another reference variable
        aStudent = new Student(1234567,60.0,80.0,true);
        meToo = new Student(7654321,54.5, 83.4, true);
        for(anum=0 ; anum<5 , anum=i+1)
        {
            aStudent.setAssignment(anum, 60.0);
            meToo.setAssignment(anum, 65.0);
        }
        System.out.println("The mark for student "+aStudent.getId()+
                           " is "+ aStudent.getFinalMark());
        Student.setMidWeight(0.30);
        Student.setAssignmentWeight(0.15);
        System.out.println("The mark for student "+meToo.getId()+
                           " is "+ meToo.getFinalMark());
        System.out.println("The mark for student "+aStudent.getId()+
                           " is "+ aStudent.getFinalMark());
    }
}
```

Exercise 11-4 - Using Class Variables and Methods



Terminal Window



Summary of Class Design (1)

- In an object-oriented language such as Java, designing a class is a large part of the effort to create software.

"Classes struggle, some classes triumph, others are eliminated." --Mao Zedong

- Decisions have to be made as to:
 - What information should be in the class?
 - What fields should each object have?
 - What fields should be associated with the class?
 - What type are the fields?
 - How do we initialize, set, and change the fields?
 - What are the operations we may want to ask the class to perform?
 - What other instance methods are needed?
 - What other class methods are needed?
 - What are the algorithms for all of these methods?

Summary of Class Design (2)

- Some things to keep in mind when making class design decisions:
 - How might the class be modified in the future?
 - Is there anything in the class that is "hard coded" that perhaps should be a variable?
 - Safety:
 - Is there a chance that a variable is used before it receives a value?
 - When a method is called, what does the method assume about the parameter values? Are those assumptions checked?

Example of a Class Design: Fraction

- We shall define class from which objects represent objects such as $2/3$, $(-1)/5$, or $7/4$.
- Since $2 / 3 = 4 / 6 = 6 / 9$, and $(-1) / 2 = 1 / (-2)$, etc., fractions with different numerators and denominators can still represent the same fraction.
- We shall store fractions in a « standard form »:
 1. The numerator and denominator do not have any common factor other than 1 or -1.
 2. The denominator must be positive.
- In math, the denominator of a fraction can never be 0. But we may not prevent users from creating a fraction with a 0 denominator. Java has a mechanism called **exception handling** which handles error conditions (such as divide by 0). We shall NOT study exception handling in this course.

Specification for a Fraction class

- A fraction consists of a numerator and a denominator.
- The numerator of a fraction is an integer.
- The denominator of a fraction is an integer not 0.
 - If the denominator is not specified at creation, it is assumed to be 1.
- A fraction is always in "standard form"; that is
 - The greatest common divisor (GCD) of the numerator and denominator is always 1
 - The denominator is always positive.
 - Example: $6/-9$ should be represented as $-2/3$
 - Special case: if the numerator is 0, the fraction is represented as $0/1$
- A fraction with denominator 1 should be displayed as the equivalent integer; otherwise in the form numerator/denominator.

Exercise 11-5: Designing a Fraction class ?

- What information do we need to store in a Fraction?
- What operations do we need?
 - [Aside from creating fractions, the only mathematical operation we will implement is addition of two fractions]

Exercise 11-6: Simplify Fraction to Standard Form



-
- To make sure that each **Fraction** instance will be in lowest terms, a method **simplify** will be used.
 - Assume that you have a method **gcd(a,b)** that will return the greatest common divisor of two integers.
 - Write Java methods to put a fraction into standard form.

Exercise 11-7: Method for GCD



-
- A recursive GCD algorithm for $\text{gcd}(a,b)$:
 - If $a \bmod b$ is 0, $\text{gcd}(a, b)$ is b
 - $a \bmod b$ is the remainder when a is divided by b
 - Otherwise, $\text{gcd}(a,b)$ is $\text{gcd}(b, a \bmod b)$
 - Question: will this algorithm always reach the base case?
 - Note that $a \bmod b$ is at most $b - 1$.
 - Careful: what if b is set to 0?

Exercise 11-8: Fraction Constructors ?

- Write constructors for a **Fraction** that:
 - take 2 integers: the numerator and the denominator
 - takes 1 integer, representing an integer that is to be converted to a Fraction

Exercise 11-9: Displaying Fractions



-
- Write a Java method to display a **Fraction**.
 - Sample usage:
 - **Fraction f1 = new Fraction (6, -9);**
 - **f1.display();**
 - Result: **-2/3**

Exercise 11-10: Adding Fractions



- Write a Java method that will add two Fractions.

- Sample usage:

```
Fraction f1 = new Fraction( 1, 2 );
```

```
Fraction f2 = new Fraction( 1, 3 );
```

```
Fraction sum = f1.addTo( f2 );
```

```
sum.display( );
```

- Result: 5/6

Exercise 11-11: Adding an Integer to a Fraction



- Write a method that will add an integer to a Fraction:

```
Fraction f1 = new Fraction( 5, 2 );  
Fraction sum = f1.plus( 3 );  
sum.display( );
```

- Result: 11/2

Other Arithmetic Operations

- As an additional home exercise, try to create similar methods for other operations (subtraction, multiplication and division) on two fractions and with a fraction and integer.
- What is special about division?

Some final words...

- "At the source of every error which is blamed on the computer, you will find at least two human errors, one of which is the error of blaming it on the computer."
 - Anonymous
- "We shall do a much better programming job, provided we approach the task with a full appreciation of its tremendous difficulty, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers."
 - Alan Turing