

School of Electrical Engineering and Computing

SENG2200 PROGRAMMING LANGUAGES & PARADIGMS (S1, 2021)

Introduction

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

Outline

- Course Introduction
- Java Review
- Preliminaries
- UML

Course Organisation

Lectures

Tuesday 08:00 – 10:00 @ CT202

Labs

Monday 10:00 – 12:00 @ ES210

Monday 10:00 – 12:00 @ EFG14

Tuesday 10:00 – 12:00 @ ES210

Tuesday 12:00 – 14:00 @ ES137

(No labs in Week 1, 8 and 13)

Consultation Times

Lecturer: Dr. Nan Li

Office: ES222

Email: Nan.Li@newcastle.edu.au

Phone: 4921 6503

Consultation Times

Tuesday: 14:00 – 16:00

Other Teaching Staff

- **Mr. Daniel Bell**
 - *Email: daniel.p.bell@newcastle.edu.au*
 - *Demonstrator and Marker*
- **Mr. Terry Truong**
 - *Email: terry.truong@newcastle.edu.au*
 - *Demonstrator*
- **Mr. Cody Lewis**
 - *Email: cody.lewis@newcastle.edu.au*
 - *Demonstrator*

Contact

- Discussion Forum
 - *General discussion*
 - *A1 ~ A3*
- Consultation time
- Email
 - *It is the **preferred** method if the forums and consultation time do not suit you.*

What is about the course?

- What will be covered?
 - *Object-Oriented Programming (OOP) topics*
 - *Memory management*
 - *Parameter-passing methods*
 - *Concurrency*
 - *Functional and logical programming*
- Knowledge required
 - *Data Structure*
 - *UML*
 - *Programming: Java and C/C++*

Course Objectives

On successful completion of this course, students should be able to

- Use Elementary Language Theory and Specification to describe language features.
- Program using advanced features of object-oriented languages.
- Compare the low level implementation of language mechanisms, such as pointers and references, activation records, method tables, memory allocation/de-allocation and garbage collection, thread activation and communication.
- Compare and analyse alternate parameter passing mechanisms.
- Describe the operation of non-object programming paradigms.

Contents

Week	Topic	Assessment
1	Introduction & Review	A1 out
2	Memory Management	
3	Inheritance	
4	Polymorphism	A1 due
5	Generics	A2 out
6	Iterators	
7	-	Midterm Test
8	Parameter-Passing	A2 due
9	Concurrency	A3 out
10	Functional Programming I	
11	Functional Programming II	
12	Logical Programming	A3 due
13	Revision	

Note: The order of topics and due date of assignments are subject to change.

Assessments

	Assessment Name	Due Date	Involvement	Weighting
1	Assignment 1 – Programming	Week 4	Individual	10%
2	Assignment 2 – Programming	Week 8	Individual	15%
3	Assignment 3 – Programming	Week 12	Individual	15%
4	Midterm Test	Week 7	Individual	10%
5	Formal Examination*	EXAM PERIOD	Individual	50%

Assessments

- Self-Quiz
 - *Aim: Check if you understand the topic of the week.*
 - *Weekly*
 - *Blackboard*
 - *Formative*

References

- RW Sebesta, Concepts of Programming Languages, 10e, Pearson, 2012. ISBN 13:978-0-13-139531-2

Part 1 - Java Review

Overview

- What does this course expect you to know about Java?
 - *Control Structures (all).*
 - *What is a reference? What is a primitive data item? Why?*
 - *How do procedures/functions, parameter passing, and return values work for primitives and references?*
 - *Classes: declaration, instantiation, initialization, access, update.*
 - *Arrays: declaration, instantiation, access and update.*
- What things out of SENG1120 will be of help?
 - *Basic knowledge of data structures eg linked lists, stacks, queues.*
 - *Knowledge of C++ templates and basic memory management an advantage*
- Where to from here?
 - *We will advance your knowledge of Java.*
 - *We will re-do a reasonable amount of C++.*
 - *Expanding O-O concepts to include Inheritance, Abstract Classes, Pure Abstract Classes and Interfaces, comparing Java and C++ as we go.*

Control Structures

- Assignment and Evaluation
- If-Else
- While and Do-While
- For
- Switch Case Break Default
- Try Catch

Primitive Data and References

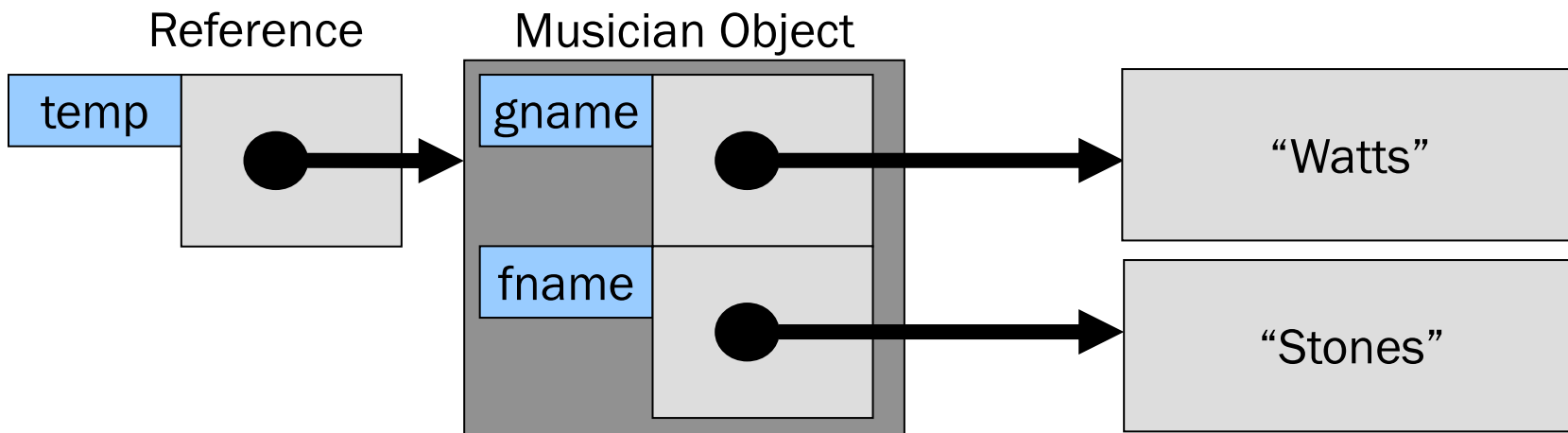
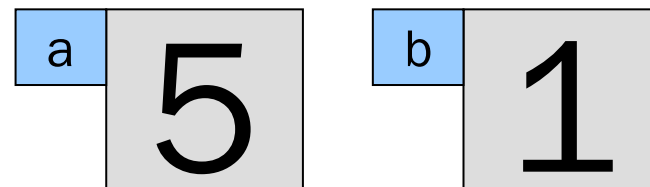
// Compare the actions taken when

```
int a=1, b = a; a = 5;
```

// with those when

```
Musician drums = new Musician("Watts","Stones");
```

```
Musician temp = drums;
```



Evaluation

- What does each of the operators do?

+ - * / % < >= > <= == !=

- What else do we have to take into account here?

Functions: Parameters and Return values

```
int procN1(int p1,int p2,int p3){           //Compare
    int i = 0, j = 0, k = 1;
    ...
    return k;
}
Student procN2(Student s1,Student s2){//with
    Student t1, t2;
    ...
    return t2;
}
```

- What does each function assume about data from the caller?
- Describe in detail what happens to each variable and return value during each of the function calls.

Classes and Static Data/Methods

- What does it really mean for data to be static in Java?

```
private static int numberOfAccounts = 0;
```

```
public static int getCount( ) {  
    return numberOfAccounts;  
}
```

- When do you use static data and static methods within a class?
- Java and C++ use and implement static data and methods in similar ways, but C++ uses the keyword `static` in extra (non-O-O) ways.

Arrays

- What are the steps in setting up an array?

- *Declaration*
- *Allocation*
- *Initialization*

```
int[] arr1;  
int[] arr2 = new int[5];  
Student[] arr3, arr4;
```

- *How to access/update these arrays?*
- *What happens with the statements:*

```
arr1 = arr2;      arr1[2]=arr2[1];  // and  
arr3 = arr4;      arr3[0] = arr4[1];
```

Exceptions

- What is an exception in Java? An Object (special)
- When it is used? Unexpected event
- For what purpose? Notifying the event
- What sort of mechanism is this really? Procedure escape
- Do you know how to catch/throw exceptions?

Prelude to C++

- C++ was developed from C.
“C with O-O extensions.” – Bjarne Stroustrup.
- Java was developed under heavy influences from C++, it was made completely O-O and also was developed under the need to easily access the Internet.
- Learning C++?
 - *If you find it difficult to learn C++ then you are more likely to have misunderstood the O-O concepts in Java, rather than C++ itself being too difficult.*

Where to from here?

- How Java does Linked Structures
- How Java and C++ allow for Software Reuse
- How Java and C++ allow for Software Extensibility

Which of the following make you anxious?

What can you remember from your previous programming experiences?

- Can you write a Student class in Java?
- Can you instantiate an array of Student objects?
- Can you search that array for a particular Student?
- Can you write a Student class in C++?
- Can you write a doubly linked list template class for C++?
- Can you use it to construct a doubly linked list of Student objects?
- Can you pass both your array and your linked list as a parameter to a method (in their respective language)?
- Can you construct your array or linked list within a method and return it as the result of the method?

Part 2 - Preliminaries

Reasons for Studying Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of the significance of implementation
- Overall advancement of computing

Programming Domains

- Scientific applications
 - *Large number of floating point computations*
 - *Fortran*
- Business applications
 - *Produce reports, use decimal numbers and characters*
 - *COBOL*
- Artificial intelligence
 - *Symbols rather than numbers manipulated*
 - *LISP*
- Systems programming
 - *Need efficiency because of continuous use*
 - *C*
- Web Software
 - *Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)*

Language Evaluation Criteria

- Four evaluation criteria
 - *Readability*
 - *Writability*
 - *Reliability*
 - *Cost*
- Language Characteristics
 - *Simplicity*
 - *Orthogonality*
 - *Data Types*
 - *Syntax Design*
 - *Support for Abstraction*
 - *Expressivity*
 - *Type Checking*
 - *Exception Handling*
 - *Restricted Aliasing*

Language Evaluation Criteria

Table 1.1 Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Cost is not included in the table because it is only slightly related to the other criteria and the characteristics that influence them.

Language Evaluation Criteria

- Readability
 - *The ease with which programs can be read and understood*
 - e.g. by someone else (or the programmer at a later time)
- Writability
 - *The ease with which a language can be used to create programs*
 - e.g. power of constructs vs clarity of constructs
- Reliability
 - *Conformance to specifications (i.e., performs to its specifications) in ALL situations – particularly as it applies to a software developer. Does the language help a programmer to write correct programs?*

Language Evaluation Criteria

- Cost
 - *The ultimate criterion*
 - not just the cost of the initial development
 - *The truth is that it always comes down to money.*
 - *But remember - cheapest in the short-term is not always cheapest over the life of a software artifact.*

Readability

- Overall simplicity
 - *A manageable set of features and constructs*
 - *Low feature multiplicity (means of doing the same operation)*
 - *Minimal operator overloading*
- Orthogonality
 - *A relatively small set of primitive constructs can be combined in a relatively small number of ways*
 - *Purely Orthogonal - Every possible combination is legal*
 - *No language will be purely orthogonal – all will have a level of orthogonality*

Readability

- Control statements
 - *The presence of well-known control structures*
 - e.g., `while` statement, `if-then-else` statement
- Data types and structures
 - *The presence of adequate facilities for defining data structures*
 - Or more generally – abstraction and modularization
- Syntax considerations
 - *Identifier forms: flexible composition*
 - *Special words and methods of forming compound statements*
 - *Form and meaning: self-descriptive constructs, meaningful keywords*

Writability

- Simplicity and orthogonality
 - *Few constructs, a small number of primitives, a small set of rules for combining them*
- Support for abstraction
 - *The ability to define and use complex structures or operations in ways that allow details to be ignored*
- Expressivity
 - *A set of relatively convenient ways of specifying operations*
 - Example: What's wrong with the `while` statement?
 - Example: the inclusion of `for` statement in many modern languages

Reliability

- Type checking
 - *Testing for type errors*
- Exception handling
 - *Intercept run-time errors and take corrective measures*
- Aliasing
 - *Presence of two or more distinct referencing methods for the same memory location*
- Readability and Writability
 - *A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability*

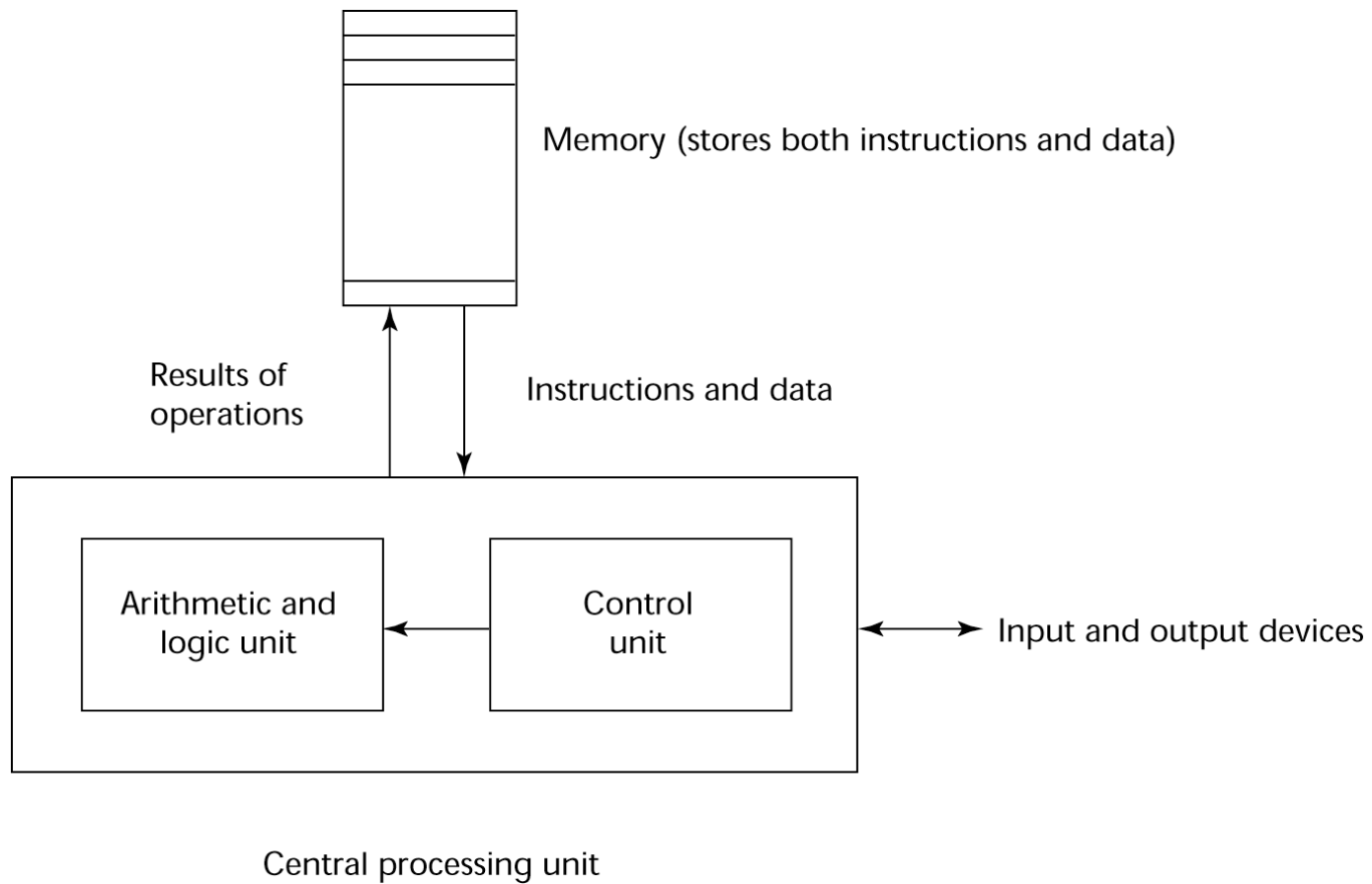
Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs
 - *This does not just refer to fixing errors*
 - *Extending a program and integrating it with/into another system is part of maintenance (and often the most important part)*

Influences on Language Design

- Computer Architecture
 - *Languages are developed around the prevalent computer architecture, known as the von Neumann architecture*
- Programming Methodologies
 - *New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages*

Computer Architecture



Von Neumann Architecture

Sebesta 10e

Computer Architecture

Imperative languages, most dominant, because of von Neumann computers

- Data and programs stored in memory
- Memory is separate from CPU
- Instructions and data are piped from memory to CPU
- Basis for imperative languages
 - *Variables model memory cells*
 - *Assignment statements model piping*
 - *Iteration is efficient*

Programming Methodology

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
 - *structured programming*
 - *top-down design and step-wise refinement*
- Late 1970s: Procedure-oriented to data-oriented
 - *data abstraction*
- Middle 1980s: Object-oriented programming
 - *Data abstraction + inheritance + polymorphism*

Language Categories

- Imperative
 - *Central features are variables, assignment statements, and iteration*
 - *Examples: C, Pascal*
- Functional
 - *Main means of making computations is by applying functions to given parameters*
 - *Examples: LISP, Scheme*
- Logic
 - *Rule-based (rules are specified in no particular order)*
 - *Example: Prolog*

Language Categories

- Object-oriented
 - *Encapsulation and Information Hiding*
 - Data abstraction
 - *Inheritance*
 - Re-use
 - *Polymorphism (late binding)*
 - Extensibility
 - *Examples: Java, C++, C#, Eiffel, Smalltalk*
- Markup
 - *New; not a programming category per se, but used to specify the layout of information in Web documents*
 - *Examples: XHTML, XML*

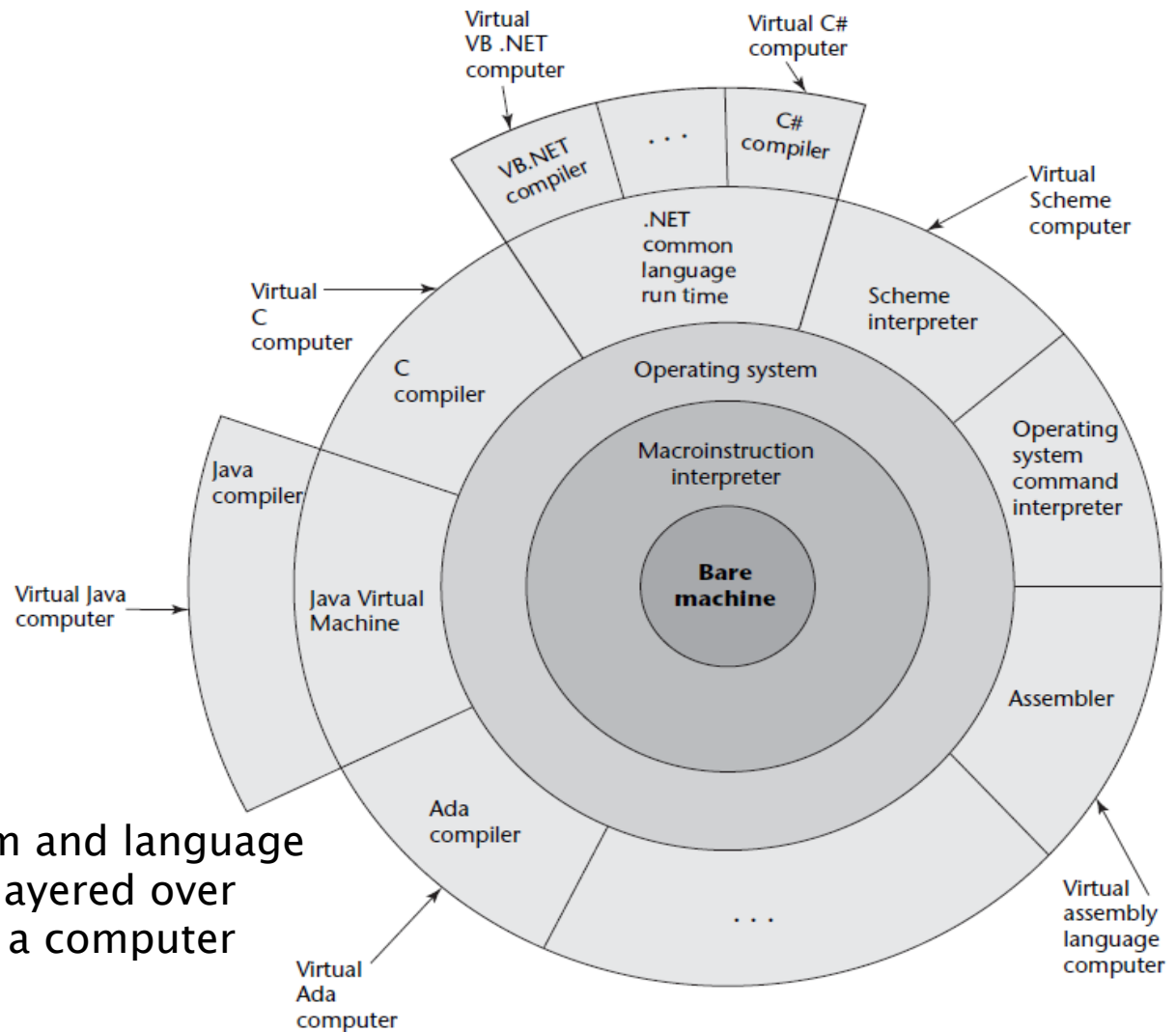
Language Design Trade-Offs

- Reliability vs. cost of execution
 - *Conflicting criteria*
 - *Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs*
- Readability vs. writability
 - *Another conflicting criteria*
 - *Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability*
- Writability (flexibility) vs. reliability
 - *Another conflicting criteria*
 - *Example: C++ pointers are powerful and very flexible but not reliably used*

Implementation Methods

- Compilation
 - *Programs are translated into machine language*
- Pure Interpretation
 - *Programs are interpreted by another program known as an interpreter*
- Hybrid Implementation Systems
 - *A compromise between compilers and pure interpreters*

Layered View of Computer

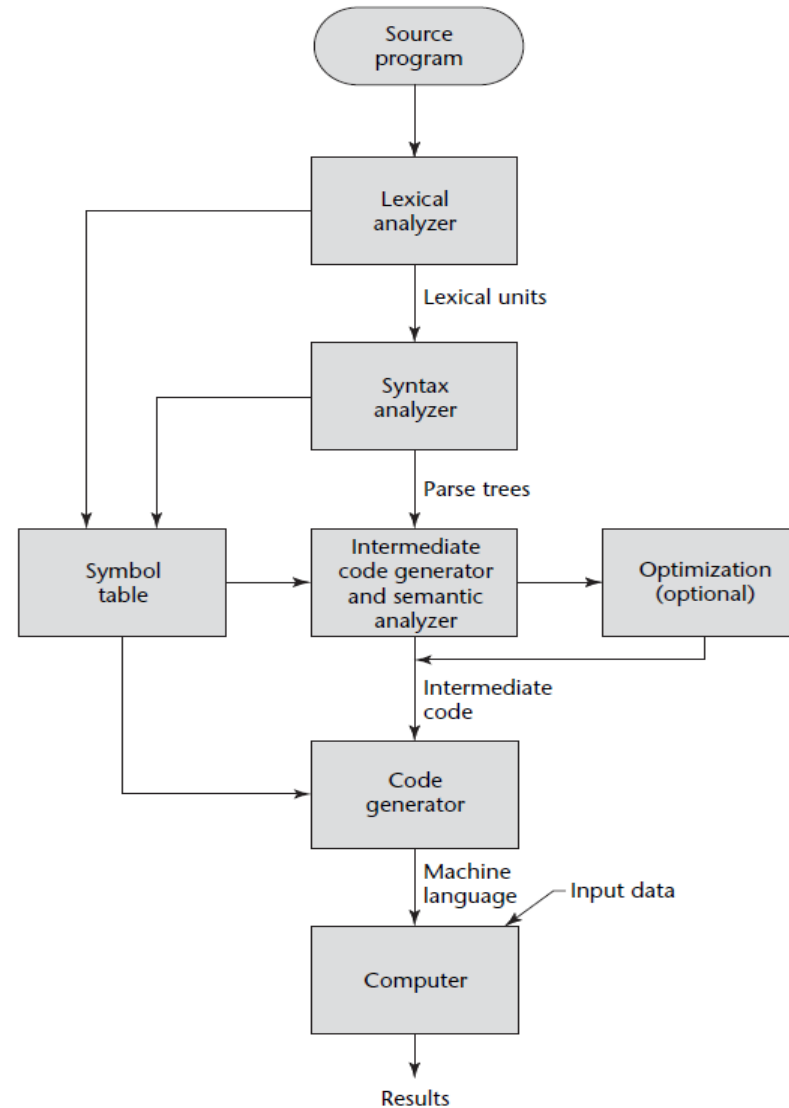


The operating system and language implementation are layered over Machine interface of a computer

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
 - *lexical analysis: converts characters in the source program into lexical units*
 - *syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of program*
 - *Semantics analysis: generate intermediate code*
 - *code generation: machine code is generated*

Compilation Process



Sebesta 10e

Additional Compilation Terms

- Load module (executable image): the user and system code together
- Linking and loading: the process of collecting system programs and linking them into a user program

Execution of Machine Code

Fetch-execute-cycle (on a von Neumann architecture)

initialize the program counter

repeat forever

 fetch the instruction pointed by the counter

 increment the counter

 decode the instruction

 execute the instruction

end repeat

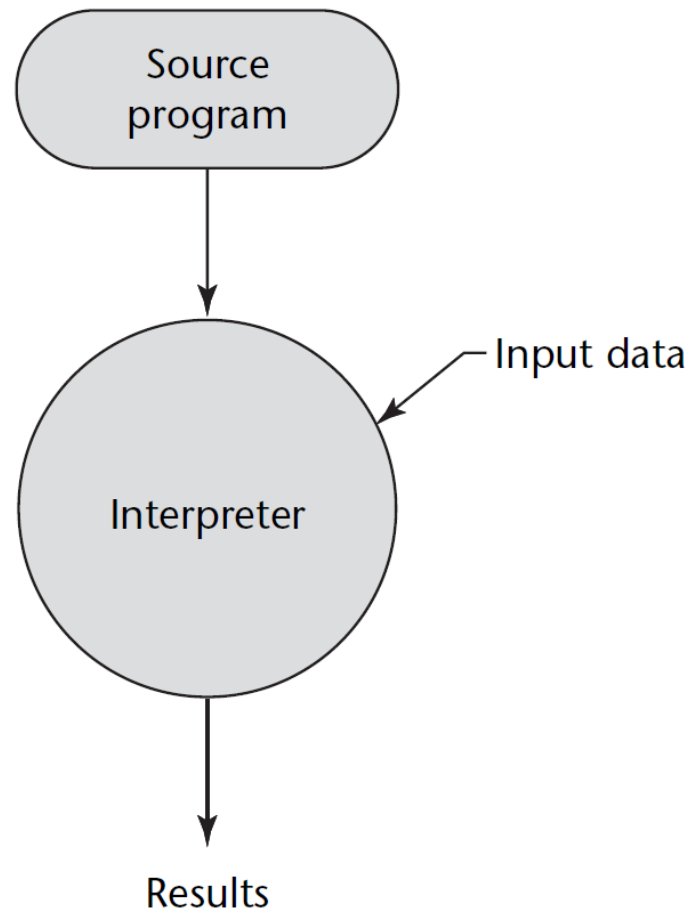
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a bottleneck
- Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can be easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript)

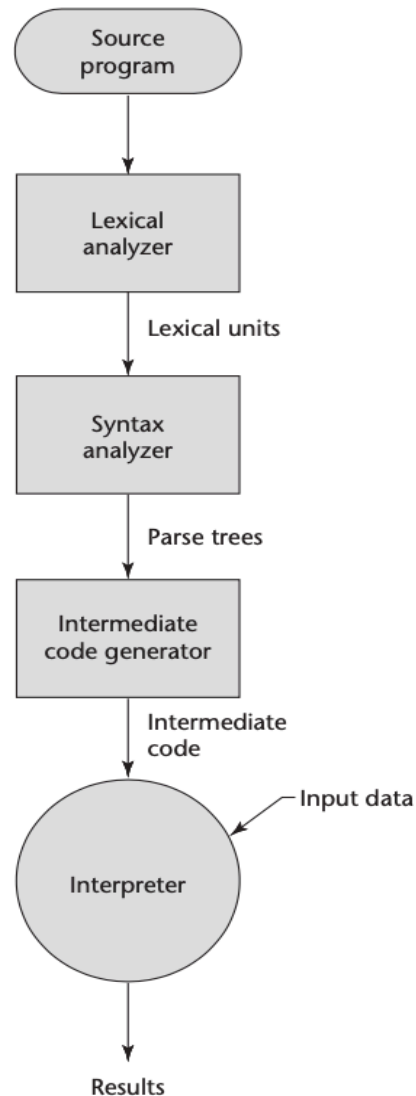
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
 - *Perl programs are partially compiled to detect errors before interpretation*
 - *Initial implementations of Java were hybrid; the intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called the Java Virtual Machine)*

Hybrid Implementation Systems



Sebesta 10e

Just-in-Time (JIT) Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- JIT is also used in some research languages in attempting to implement “live activation” of updated features.

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
 - *expands #include, #define, and similar macros*

Programming Environments

- The collection of tools used in software development
- UNIX
 - *An older operating system and tool collection*
 - *Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that run on top of UNIX*
- IntelliJ IDEA
 - *An integrated development environment for Java*
- Microsoft Visual Studio.NET
 - *A large, complex visual environment*
 - *Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++*

Summary of Part 2

- The study of programming languages is valuable for a number of reasons:
 - *Increase our capacity to use different constructs*
 - *Enable us to choose languages more intelligently*
 - *Makes learning new languages easier*
- Most important criteria for evaluating programming languages include:
 - *Readability, writability, reliability, cost*
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation

Part 3 - UML

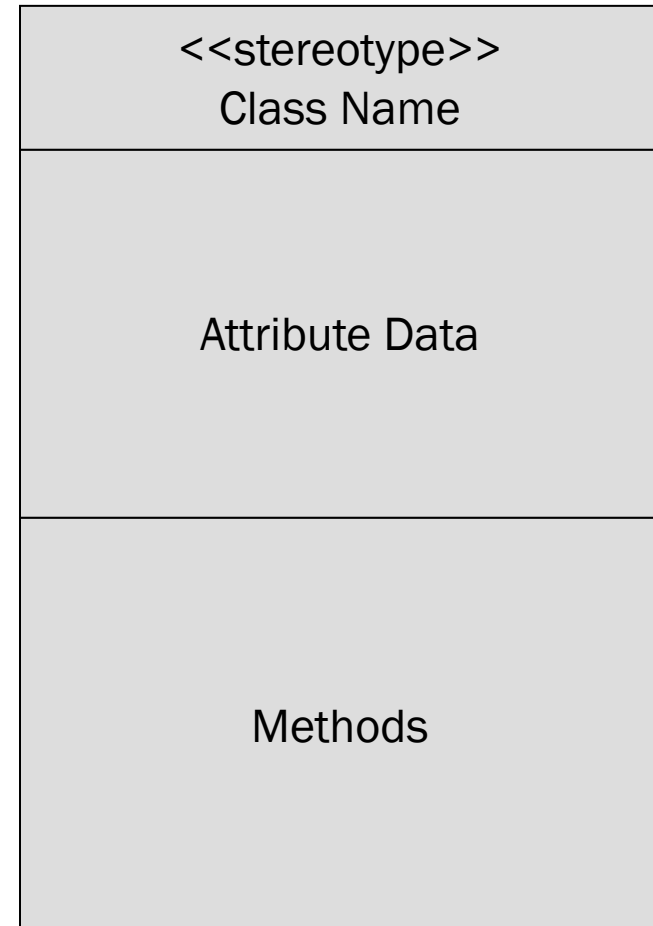
UML

- As we compare implementations of various abstractions and algorithms, Unified Modeling Language (UML) will be the means of presenting the designs being implemented.
- Class Specifications
- Class Diagrams
- Class Interactions

- We will not need:
 - *Use-cases*
 - *Sequence Diagrams*
 - *Etc.*

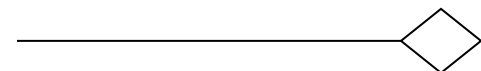
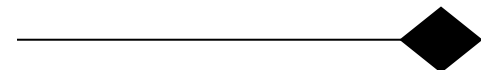
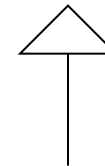
UML Class Specification

- Each class has a name, and the stereotype is optional
- A set of data items or attributes
- A set of methods
- + - and # indicate public, private or protected items
- Public methods constitute the interface of the class



Relationship

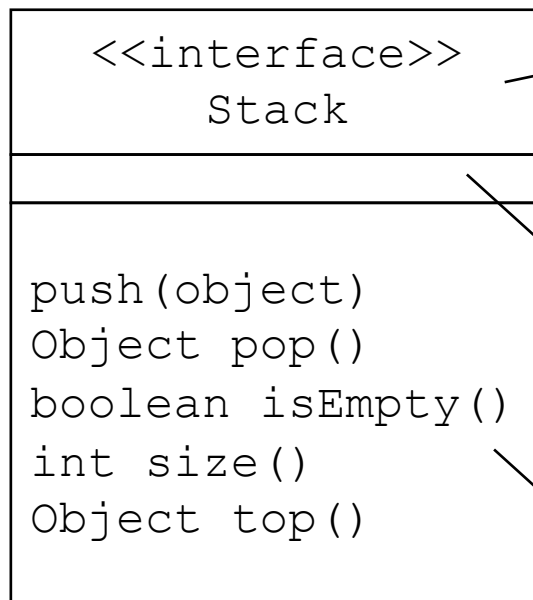
- **Association** - may have been used in examples for data structures
- **Inheritance** - you may not have seen this, there is also a version for implementation of an interface
- **Composition** – strong lifecycle dependency
- **Aggregation** – “part of”



Examples

- Interface
- Class
- Realization
- Class Diagram

Interface - “Stack”



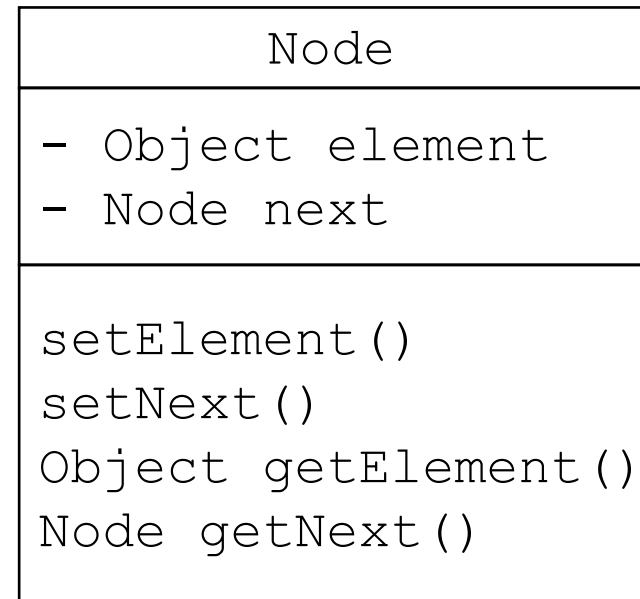
`<<interface>>` is a UML “stereotype”

An interface has no attributes (data)

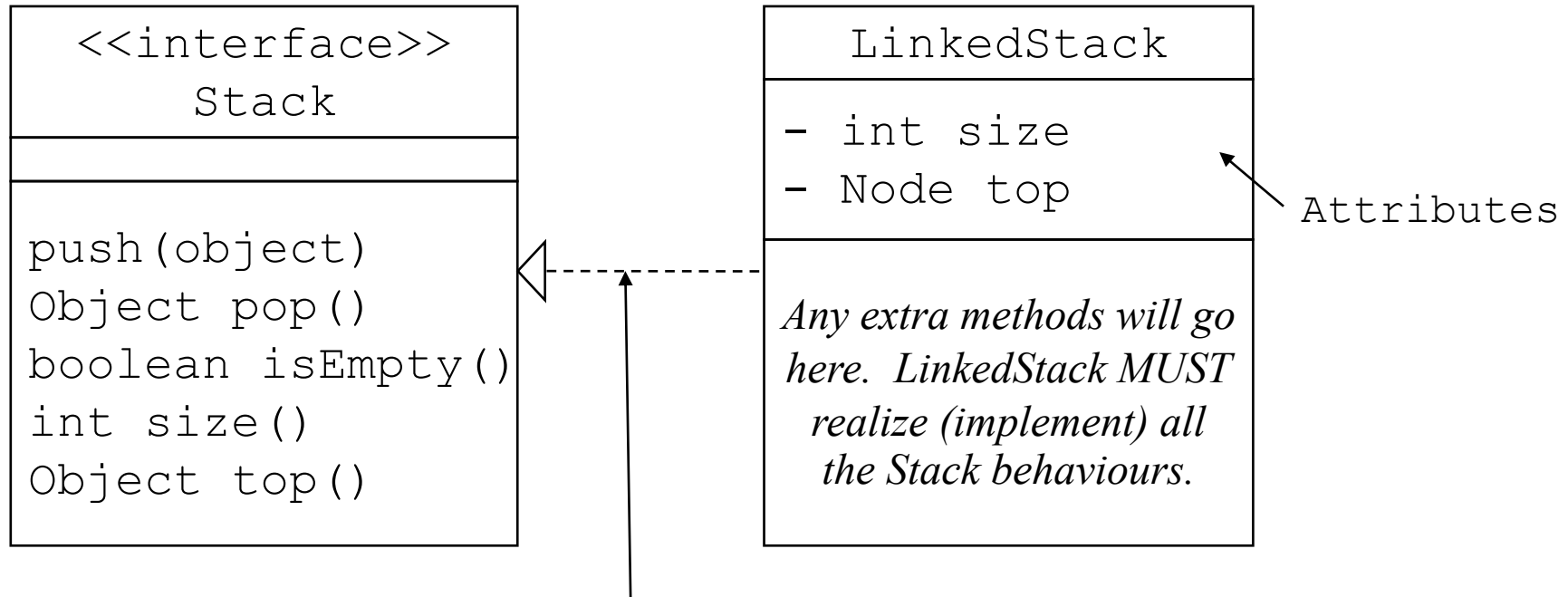
An interface lists a set of behaviours (effectively - specification of messages)

UML for Class “Node”

- Note that the Node Class has a Node attribute (a reference) as a link to the next element in the list.
- It also has a reference of type Object to hold the object itself.
- The Node Class becomes part of a set of classes and interfaces that will become the linked list.

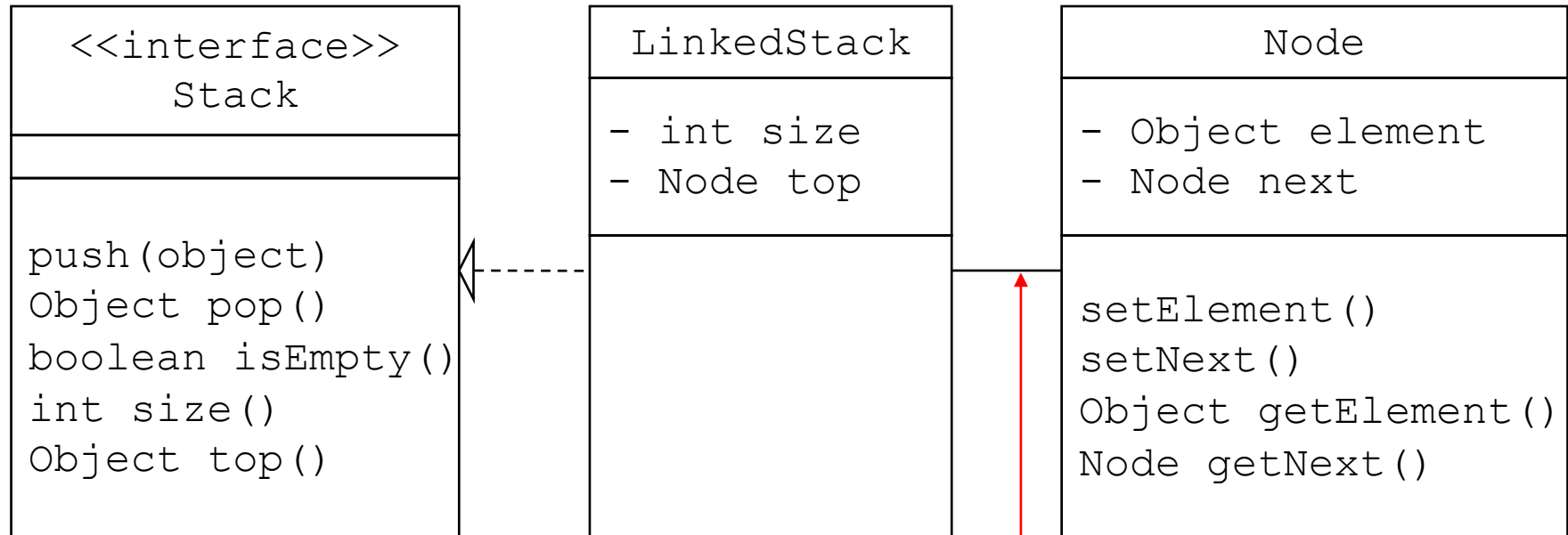


Realization



- `LinkedStack` is a realization of interface `Stack`
- – the UML term is **implementation (implements)**
- – following the Java keyword which is also ***implements***

Class Diagram



The — relationship between classes can also show direction and number.

E.g. $1 \xrightarrow{\quad} 0..*$

Indicates that Class `LinkedStack` **uses** or refers to Class `Node`.