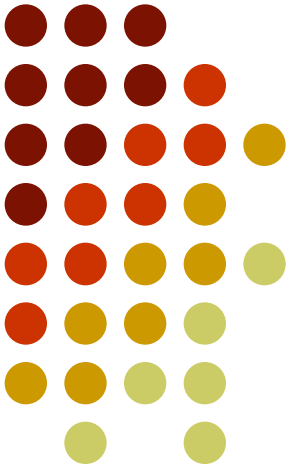
School of Electrical Engineering and Computer Science

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

SENG2200/6220 Programming Languages & Paradigms



Topic 1

Admin & Introduction:

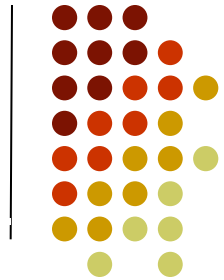
Java, C++, and Data Structures

Dr Nan Li

Office: ES222

Phone: 4921 6503

Nan.Li@newcastle.edu.au



Subject Organisation

Lectures: Wed. 17:00-19:00 @ MS202

Computer Labs: Mon. 16:00-18:00 @ ES137 or
Tue. 12:00-14:00 @ ES210 or
Tue. 16:00-18:00 @ ES205 or
Wed. 10:00-12:00 @ ES205

Consultation Times

Lecturer: Dr Nan Li

Office: ES222

Email: Nan.Li@newcastle.edu.au

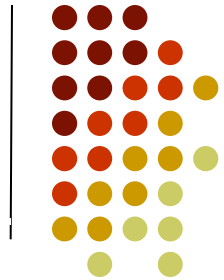
Phone: 4921 6503

Consultation Times:

- Tuesdays: 10:00-12:00
- Thursdays: 14:00-16:00

Contents

Topics	Assignments
Introduction	Week 2 – A1 out
Memory Management	Week 4 – A1 due
Inheritance	Week 5 – A2 out
Polymorphism	Week 7 – A3 out
Generics	Week 8 – A2 due
Parameter-Parsing Methods	Week 12 – A3 due
Concurrency	
Functional Programming	
Logical Programming	



Topic 1 Overview

Reasons for Studying Concepts of Programming Languages

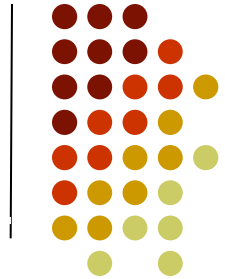
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Java and C++ Review

- O-O Basics
- First Class Objects in Java and C++

UML Introduction/Review

- Class Specifications, Interactions and Diagrams



Part 1 - Reasons for Studying Concepts of Programming Languages

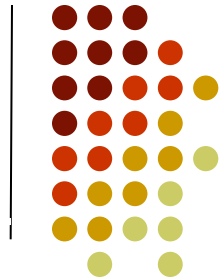
Increased ability to express ideas

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

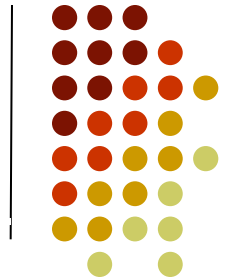
Business applications
- Produce reports, use decimal numbers and characters
- COBOL

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 (Not really an introduction??)



We are only a few minutes into the lecture material

We have only just begun an “introduction” to the course

BUT

We meet our first examinable piece of material

We will rate languages on their

- Readability
- Writablility
- Reliability
- Cost

And we will do so by looking at various easily identifiable aspects or characteristics that programming languages might have.

Language Evaluation Criteria (1)

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

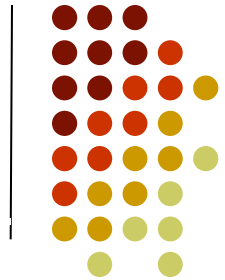
Reliability

- 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 (2)

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



Language Characteristics

Simplicity

Orthogonality

Data Types

Syntax Design

Support for Abstraction

Expressivity

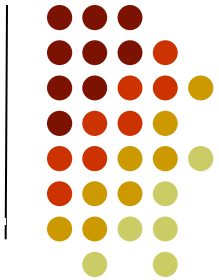
Type Checking

Exception Handling

Restricted Aliasing

See Table 1.1 p8 Sebesta 10e

Each of these may have aspects which may positively and/or negatively affect each of the major criteria of Readability, Writability, Reliability and Cost.



Evaluation Criteria: Readability

Overall simplicity

- A manageable set of features and constructs
- Few 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
- Orthogonality can be taken “too far” (e.g. C expressions as they apply to arrays).

Evaluation Criteria: Readability (2)

Control statements

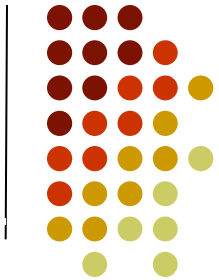
- The presence of well-known control structures
  - e.g., `while` statement, `if-then-else` statement

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



Evaluation Criteria: Writability

Simplicity and orthogonality

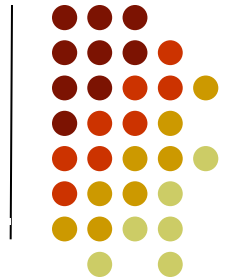
- Few constructs, a small number of primitives, a small set of rules for combining them

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



Evaluation Criteria: 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

Evaluation Criteria: Cost

Training programmers to use the language

Writing programs (closeness to particular applications)

Compiling programs

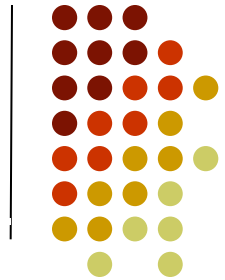
Executing programs

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)



Evaluation Criteria: Others

Portability

- The ease with which programs can be moved from one implementation to another

Generality

- The applicability to a wide range of applications

Well-defined-ness

- The completeness and precision of the language's official definition

Influences on Language Design

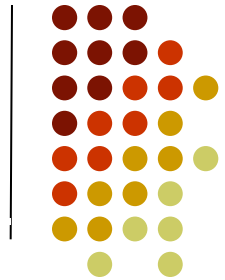
Computer Architecture

- 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 Influence

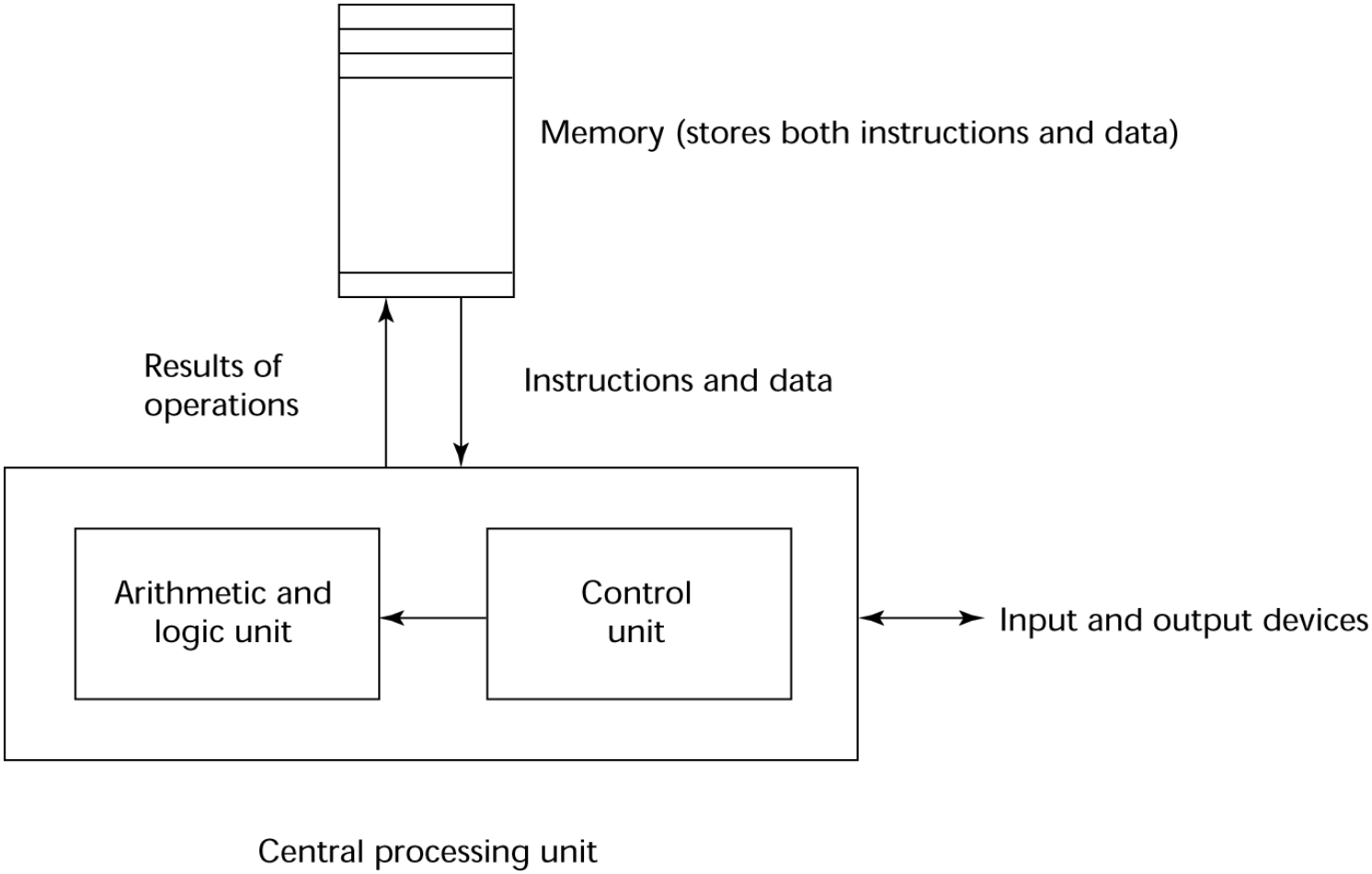


Well-known computer architecture: Von Neumann

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

The von Neumann Architecture



Influences on Programming Methodology

1950s and early 1960s: Simple applications; worry about machine efficiency

Late 1960s: People efficiency became important; readability, better control structures

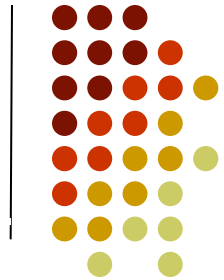
- structured programming
- top-down design and step-wise refinement

Late 1970s: Procedure-oriented to data-oriented

- data abstraction

Middle 1980s: Object-oriented programming

- Data abstraction + inheritance + polymorphism



Language Categories

Imperative

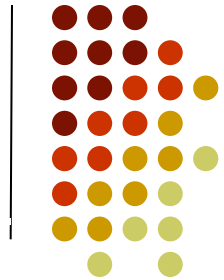
- Central features are variables, assignment statements, and iteration
- Examples: C, Pascal

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



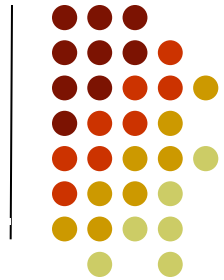
Language Categories (2)

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

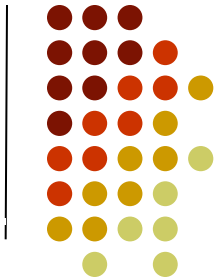
- Conflicting criteria
- Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs

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

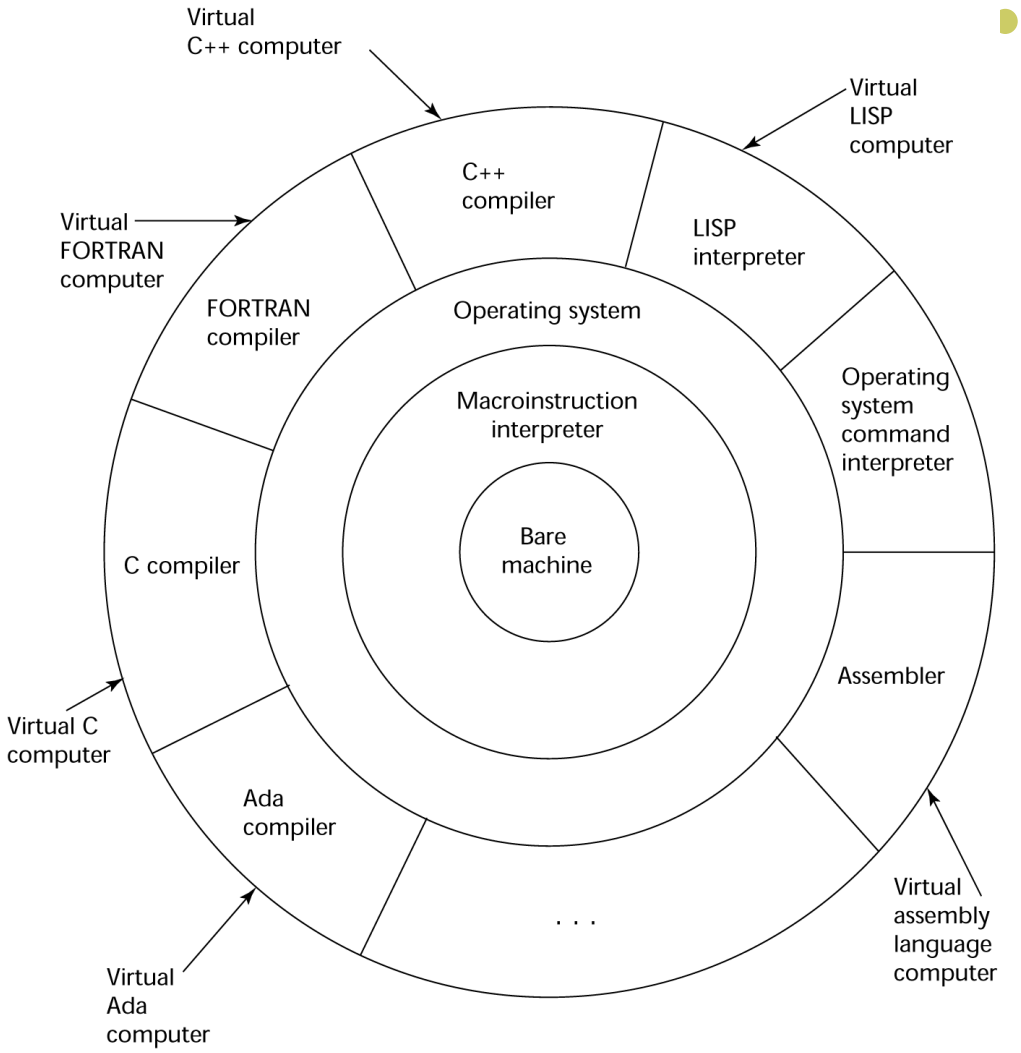
- Programs are interpreted by another program known as an interpreter

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

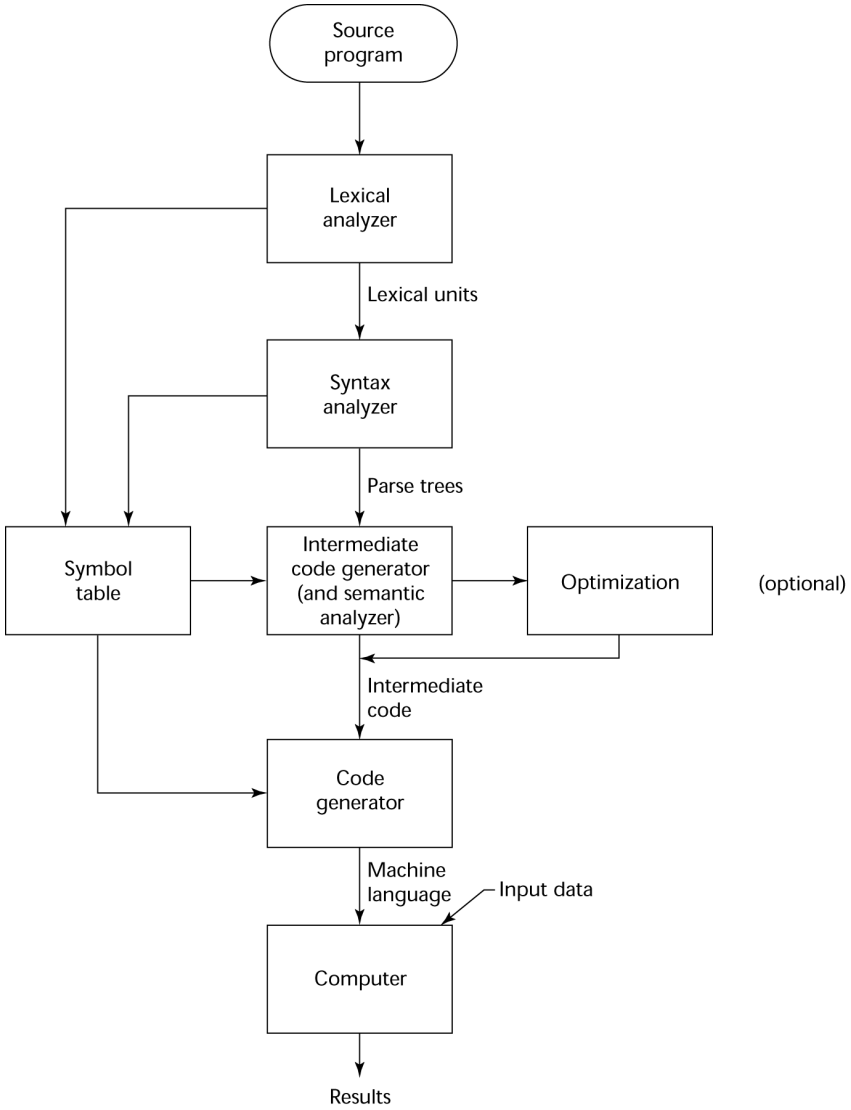
Translate high-level program (source language) into machine code (machine language)

Slow translation, fast execution

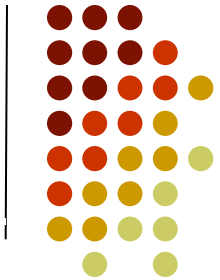
Compilation process has several phases:

- 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

The Compilation Process



Additional Compilation Terminologies



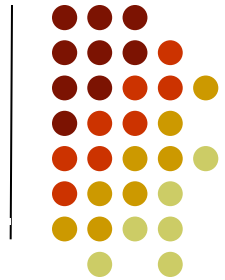
Load module (executable image): the user and system code together

Linking and loading: the process of collecting system program and linking them to 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
```



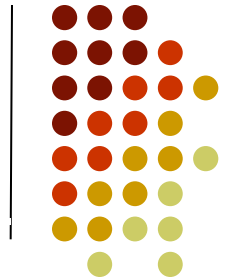
Von Neumann Bottleneck

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 easily and immediately displayed)

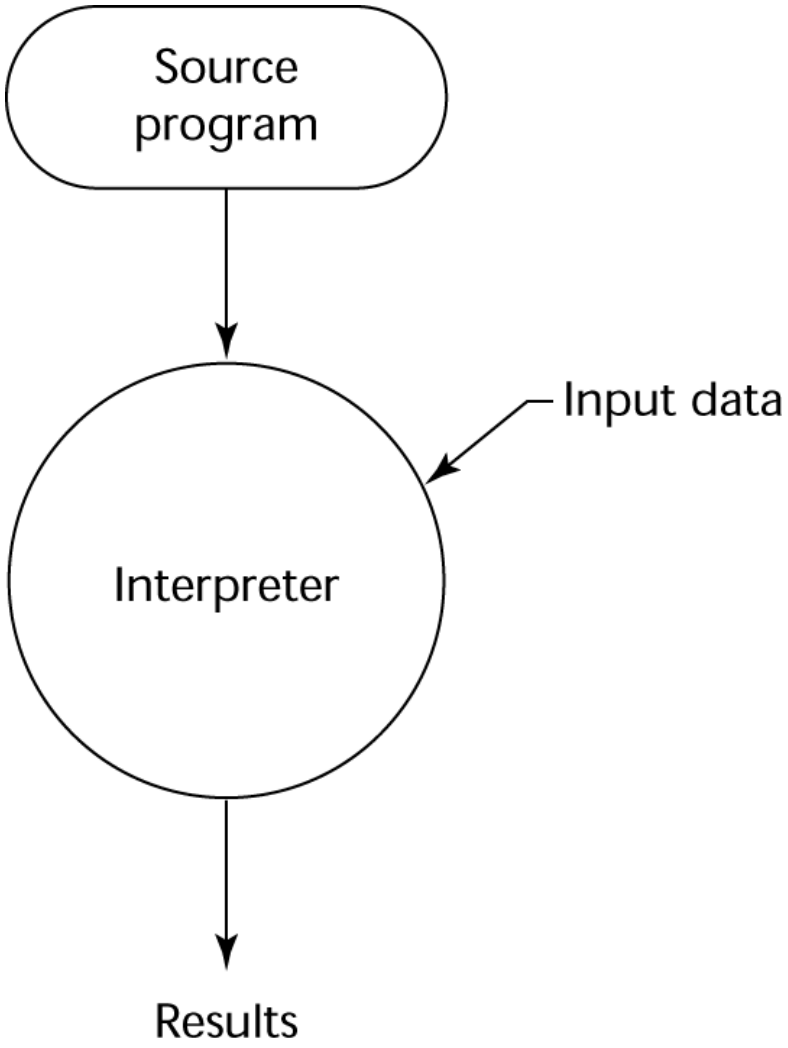
Slower execution (10 to 100 times slower than compiled programs)

Often requires more space

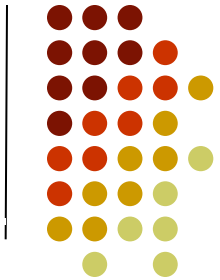
Becoming rare on high-level languages

Significant comeback with some Web scripting languages (e.g., JavaScript)

Pure Interpretation Process



Hybrid Implementation Systems



A compromise between compilers and pure interpreters

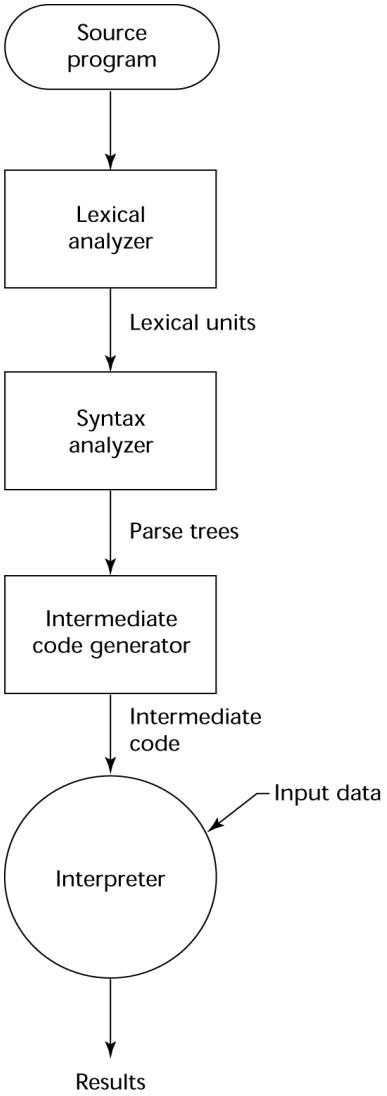
A high-level language program is translated to an intermediate language that allows easy interpretation

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 Java Virtual Machine)

Hybrid Implementation Process



Just-in-Time Implementation Systems

Initially translate programs to an intermediate language

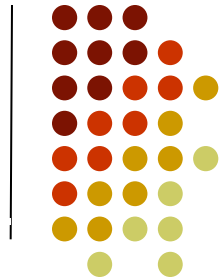
Then compile intermediate language into machine code

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.



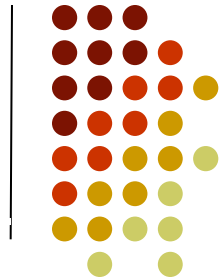
Preprocessors

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

Borland JBuilder

- 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 1 of Topic 1

The study of programming languages is valuable for a number of reasons:

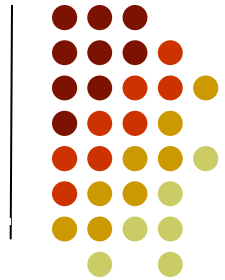
- Increase our capacity to use different constructs
- Enable us to choose languages more intelligently
- Makes learning new languages easier

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



Topic 1 Part 2 - Java Review

What does this course expect you to know about Java?

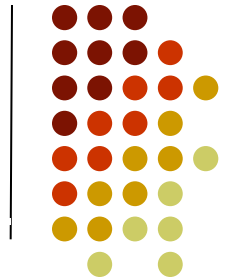
- Control Structures (all).
- What is a reference? What is a primitive data item? Why?
- How do procedures/functions, parameter passing, and return values work for primitives and references?
- Classes: declaration, instantiation, initialization, access, update.
- Arrays: declaration, instantiation, access 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 (implementing some of the SENG2130 structural concepts), comparing Java and C++ as we go.



Control Structures

Assignment and Evaluation

If Then, and If Then Else

While, and Do While

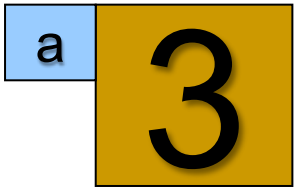
For

Switch Case Break Default

Try Catch (not a really detailed knowledge)

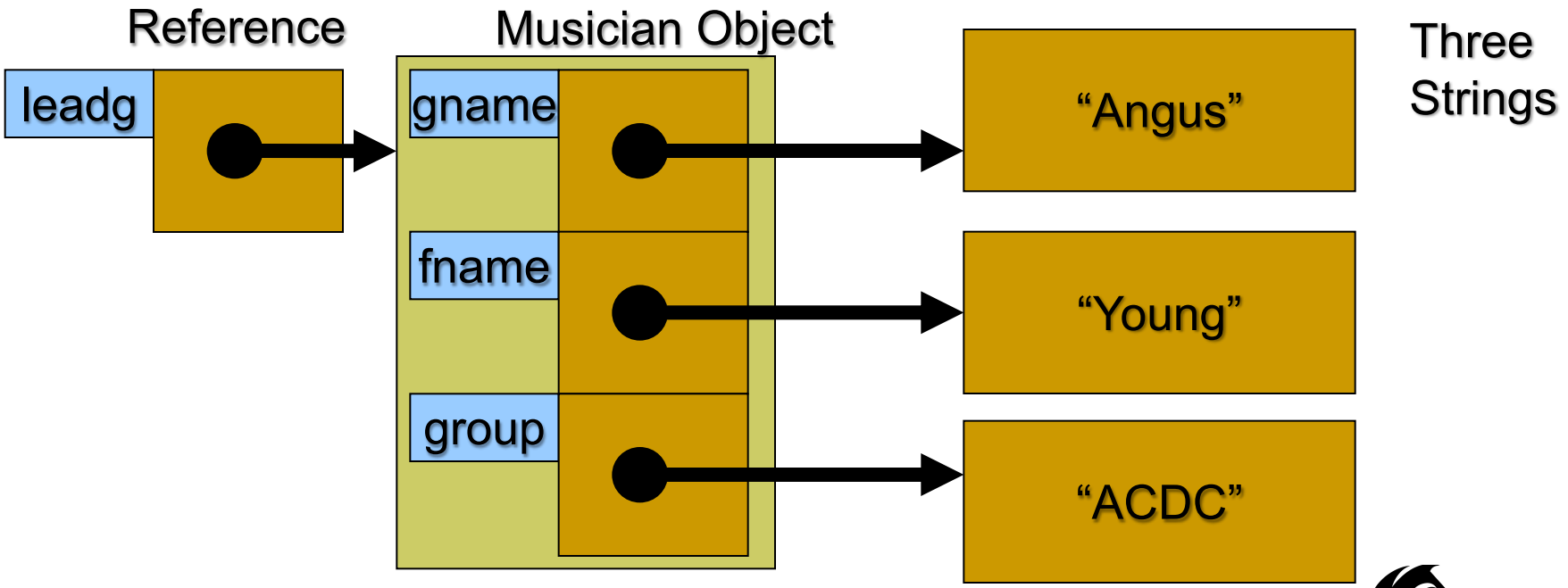
Primitive Data and References

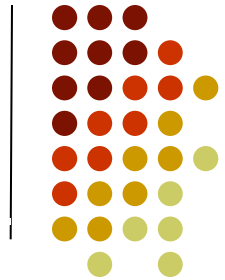
```
int a = 3;
```




```
// Compare the actions taken when
int b = a; a = 5;   // with those when
drums = new Musician("Ch...","Watts","Stones");
Musician temp = drums;
```

```
Musician leadg = new
  Musician("Angus","Young","ACDC");
```





Evaluation

What does each of the operators do?

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

What else do we have to take into account here?

We've already looked at = on previous slide

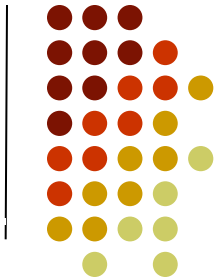
Procedures, Functions, Parameters, Locals, Return Values

```
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;
}
```

Describe in detail what happens to each variable and return value during each of the function calls.

What does each function assume about data from the caller.

Classes and Static Data and Methods



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

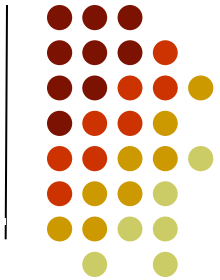
```
private static int numberOfAccounts = 0;

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



Two steps in setting up an array. What are they?

Allocation

Initialization

```
int[]     arr1,arr2; // alloc & init?
Student[] arr3,arr4; // alloc & init?
```

How do you access these arrays?

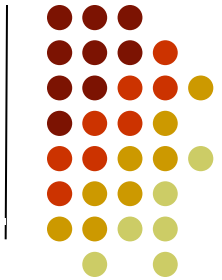
How do you update these arrays?

What happens with the statements:

```
arr2[1]=arr1[2]; arr4[3]=arr3[4];  // and
arr2 = arr1;     arr4 = arr3;
```

Show in detail.

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 exceptions?	SENG1110/6110
Do you know how to throw exceptions?	SENG1120/6120?

Prelude to C++

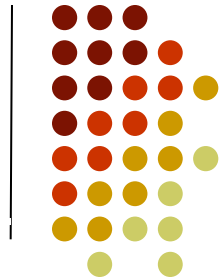
C++ was developed from C.

"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.



Review of Part 2 of Topic 1

Java Programming and Data Structures

- Imperative statements
- Methods
- Classes
- Arrays
- Linked Structures
- Class Templates

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

Review (Topic 1 Part 2) – 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?

Topic 1 Part 3 - UML Revision

As we compare implementations of various abstractions and algorithms, 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

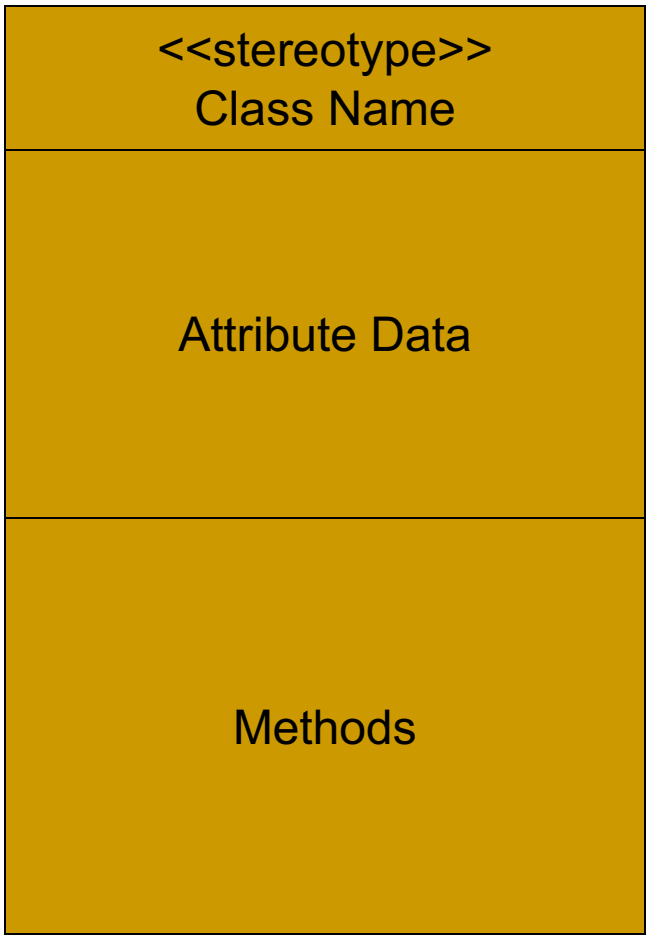
Each class has a name, and the stereotype is optional

A set of data items or attributes

A set of methods

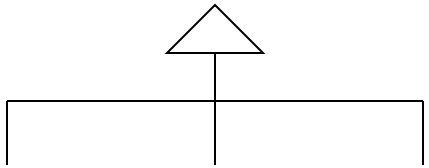
+ - and # indicate public, private or protected items

Public methods constitute the interface of the class



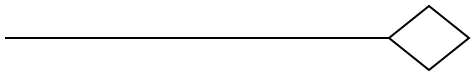
UML Class Interactions and Class Diagrams

Inheritance - you may not have seen this, there is also a version for implementation of an interface

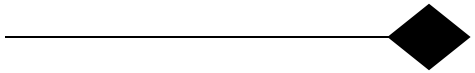


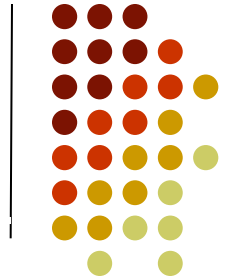
Association - may have been used in examples for data structures

Aggregation - a very difficult o-o concept



Composition - used in the adapter pattern





Examples of UML

Interfaces

Implementation

Standard Interfaces

Concrete Classes and Associations

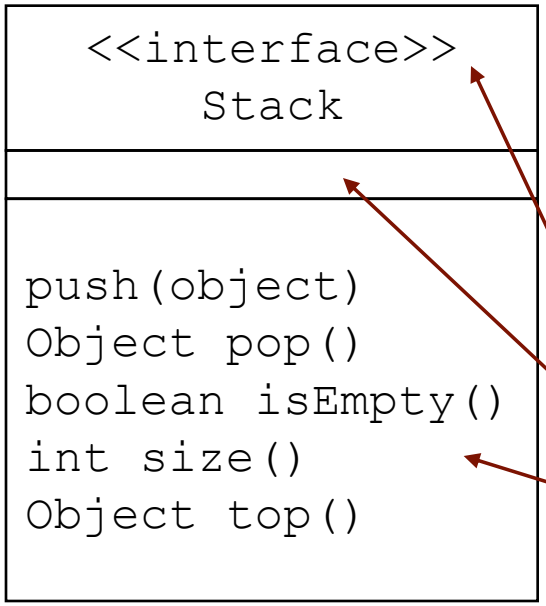
The Adapter Pattern and Composition

Modelling a Stack Interface

We will use the Unified Modeling Language (UML).

It's not the only way and many reckon it's not the best way, but it has been accepted by the OMG as a standard.

Here is how to model a Stack Interface in UML



This interface can then be directly used or mentioned in UML Class Diagrams to show how implementations of it coordinate with other classes in the problem solution.

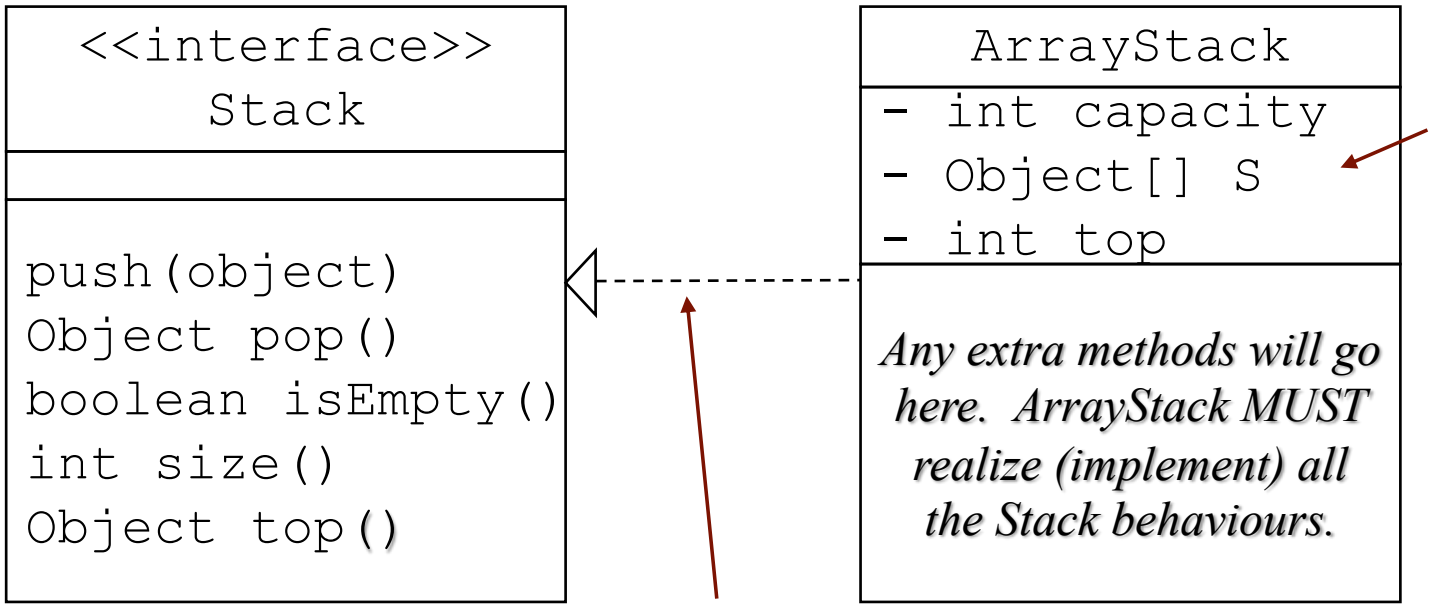
<<interface>> is a UML "stereotype"

An interface has no attributes (data)

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

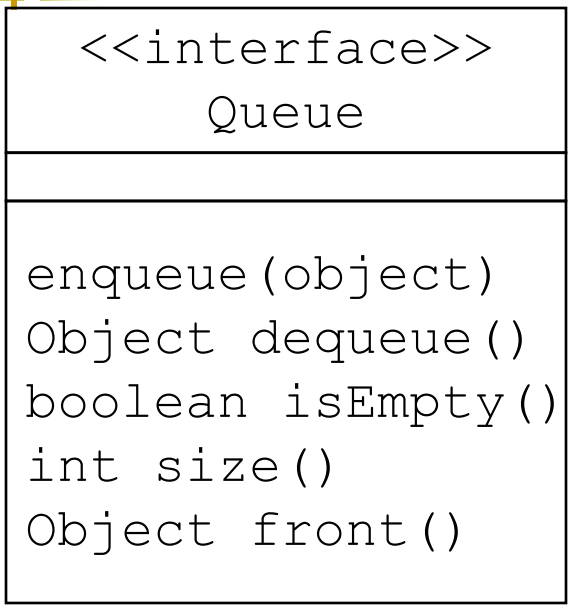
Using a UML Interface in a Design

A realization of the interface (in Java terms an implementation) is shown in a UML Class Diagram as:



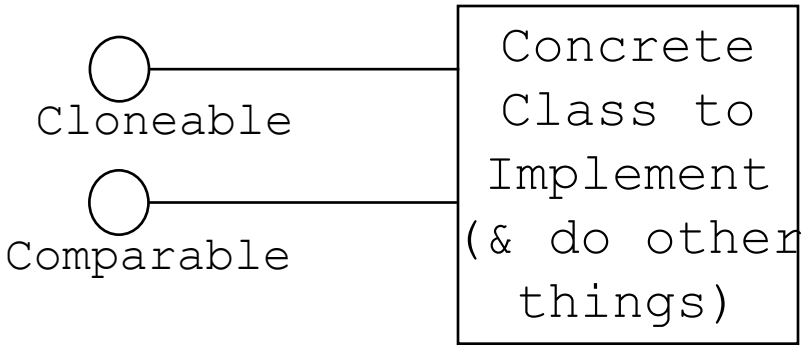
ArrayStack is a realization of interface Stack
– the UML term is implementation (implements)
– following the Java keyword which is also *implements*
Later, we will look at how to model the exceptions thrown.

Queues - A UML Interface

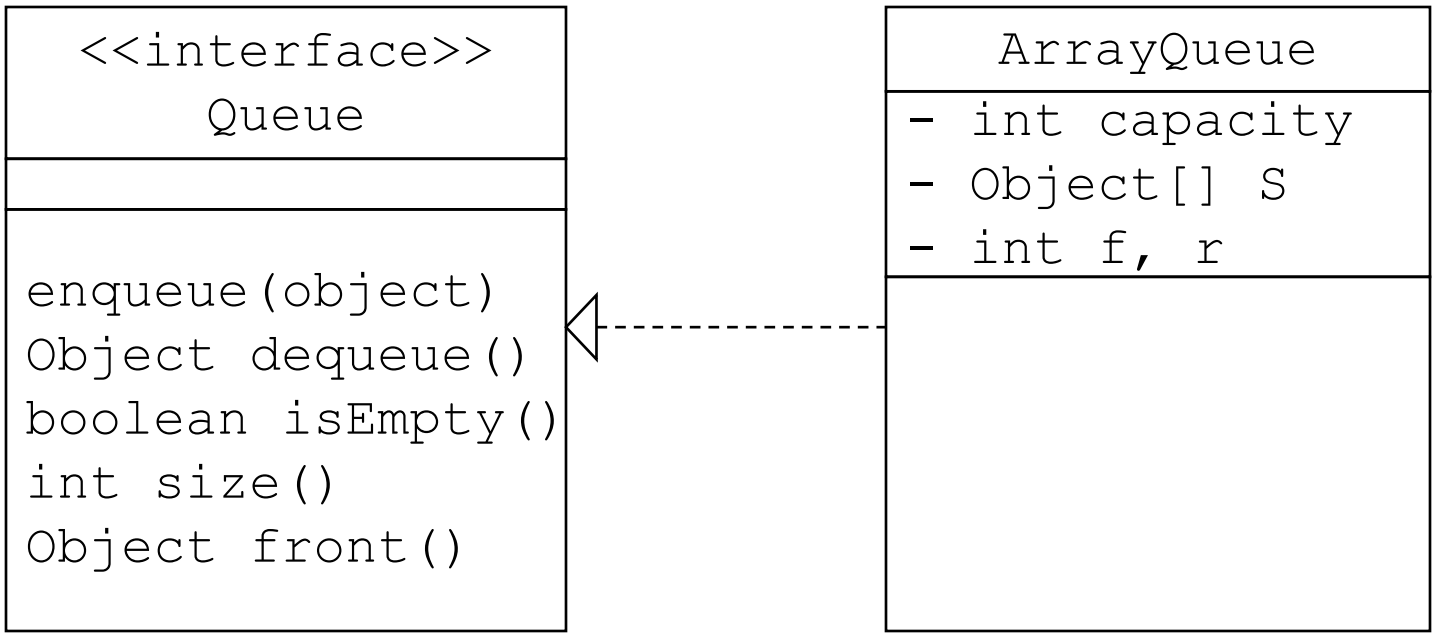


There are lots of standard interfaces in Java. Eg. *cloneable*, for which you supply a **clone( )** method, and *comparable*, for which you supply a **compareTo( )** method.

Rather than drawing a complete class-style UML box each time these are implemented by a class, it is usual to refer to them in UML as follows:



UML Model of the Queue



The data items for ArrayQueue are very similar to ArrayStack.

Maybe *capacity* is now an inappropriate name since the actual capacity of the queue is one less than the size of the array.

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.

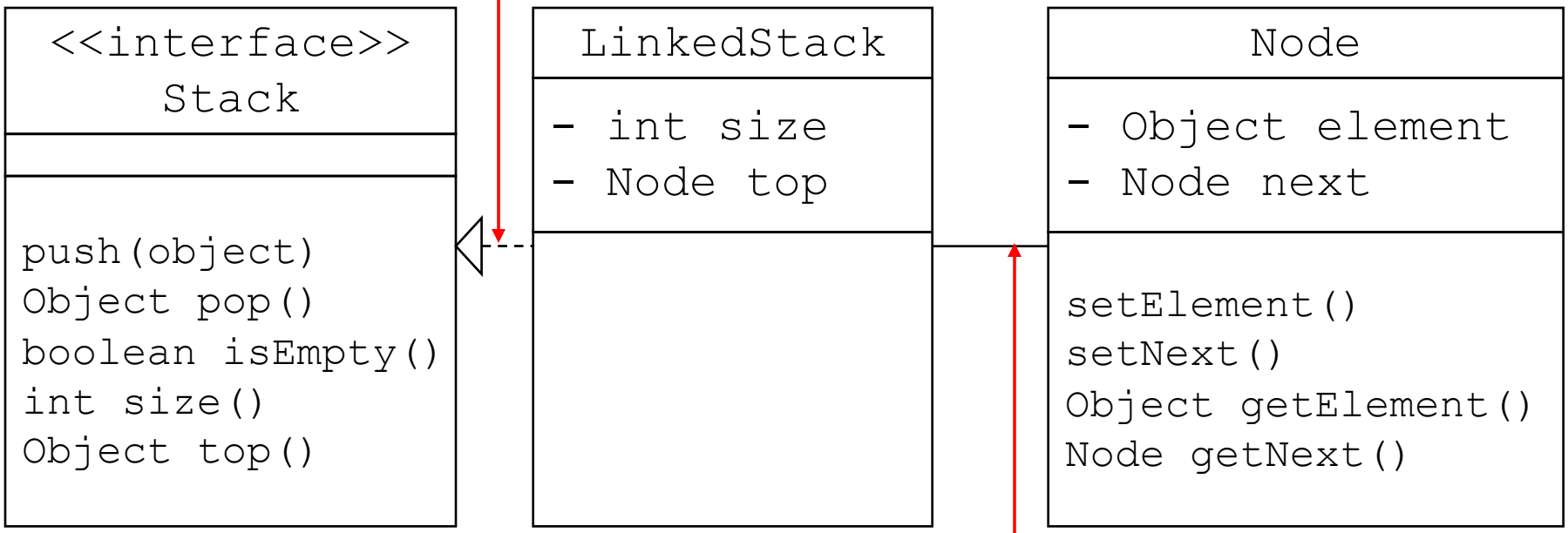
The Node Class becomes part of a set of classes and interfaces that will become the linked list.

Node
- Object element - Node next
setElement() setNext() Object getElement() Node getNext()

So the relationships between these classes must also be shown on a **UML Class Diagram**.

UML Class Diagram

LinkedStack **implements** Stack.



The ——— relationship between classes can also show direction and number.
Eg 1 ———→ *

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