- intro to the course
- intro to data structures
- correct, simple, clear

Next week and homework
- next week: intro to stacks
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, section 15.5 on stacks

Lab
- homework assigned this week. See Canvas for due date
- how to write Javadoc comments

Intro to the course
Objective: what do you need to know now, what will you learn, how will you learn it?

- in Canvas, Modules, 'Course information', please review the 'How to take this course online', 'Syllabus' and 'Course contents' documents

<u>Summary</u>

- objective of this course:
  - is the third semester, 'advanced Java programming' course
  - you get to write lots of complicated computer programs ☺
- immediate prerequisite is CSCI 114 Programming Fundamentals II, so you already know how to write Java:
  - 114 is the second semester, 'introduction to object-oriented programming (OOP) and Java' course
- prerequisite to 114 is CSCI 112 Programming Fundamentals I, so you already know how to write C
  - 112 is the first semester, 'introduction to computer programming' course, uses non object-oriented C
- every week I post a large amount of original material I have created to cover the next topic. Each week you MUST:
  - do the assigned textbook reading, then
  - watch my lecture videos, then
  - print then carefully work through my detailed `.pdf` lecture doc
  - download, run and understand my example programs as indicated
- then you <u>learn</u> how to program by applying the ideas from these lectures to all of the many programming projects that will be assigned
  - textbook reading + lecture videos + detailed lecture doc + example programs =

    successful completion of

programming assignments

- please be aware that programming classes are notoriously challenging
    - you will have to spend many hours working really hard all semester to do well in this class

Intro to data structures

Objective: introduce some ideas and terminology, and a fundamental distinction in implementation

- "study of the organization of data in main memory as the program runs, and of algorithms to manipulate the data"
  - organization and algorithms go hand in hand, as we'll see
- we've found in computer science (CS) that there are a limited number of data structures used all the time:
  - stack
  - queue
  - list
  - tree
  - graph
  - each has different uses
  - we'll cover each in turn

<u>Static vs. dynamic implementation</u>

- important distinction in the implementation of data structures:
  - <u>static</u> vs. <u>dynamic</u> implementations
  - illustrate the difference by looking at implementations of a list

<u>A list is a "sequence of items"</u>

- a list is simply a "sequence of items" e.g. implement an ordered list of chars:

  A B D E H T

  Want to be able to:

  - print the list

- search the list
- insert new char
- delete old char
- so would write a method for each of these e.g. print(), search() etc.

Static implementation of a list

- "size and shape of data structure fixed at compile time" (hence static)
- we already know how to do a static implementation! e.g. use an array to implement the list of chars

`char list[]`

	A	B	D	E	H	T
index:	0	1	2	3	4	5

Figure 1 static implementation of a list

- advantages of static implementation:
  - structured data type provided by most languages - quick, simple, convenient, easy to implement and use e.g.
  - print() - simply increment index
  - search() - simple increment
- disadvantages of static implementation are major!
  - have to allocate max memory in advance, and this cannot change (hence static)
  - have to allocate too much, otherwise not enough :-)
  - insert() - have to move lots of data around e.g. insert C
  - delete() - have to move lots of data around e.g. delete D
- use arrays to implement small, fixed-size tables where we don't have to move data around too much

Now implement as a dynamic data structure

- "size and shape of data structure varies as the program runs" (hence dynamic)

"pointer or reference is said to point to or refer to an object in memory"

- depends upon the concept of a pointer or reference (both are the same idea)
  - "pointer or reference is said to point to or refer to an object in memory"
  - e.g.:

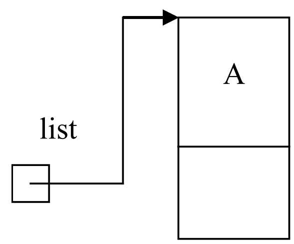

Figure 2. the list reference refers to an object

  - a reference is a variable that can have a name (e.g. list here)
  - can set it to refer to something in memory (e.g. to A node)
  - can access what it refers to
  - can change it to refer to something else (e.g. to B node)
  - can set a reference to refer to nothing – the null pointer or reference

How a reference works

- a reference is a variable that contains the memory address of the object it points to e.g.

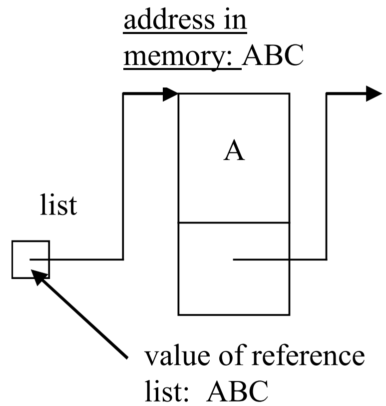

Figure 3. the value of the reference is the memory address of the object

- every object in memory has a unique address (e.g. A node)
- to make a reference point to the object, set the value of the reference to this address

- for example, how would you:
  - make list point to A (e.g. create A node with address, value of list reference is?)
  - make A point to B (e.g. value of reference in A node?)

- items in the data structure are implemented as nodes, which can be connected by references e.g.

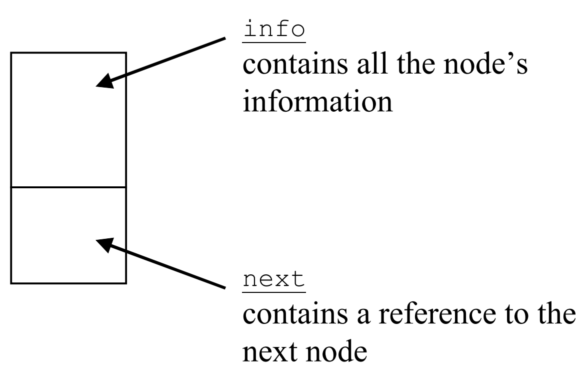

Figure 4 a node

  - must allocate memory for a new item
  - must free memory belonging to an old item

- (must avoid memory leaks – is programmer's responsibility in C vs. automatic garbage collection in Java)

- e.g. use a single linked list to implement the list of chars

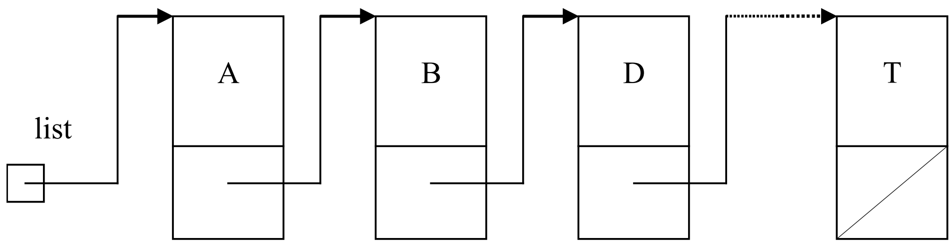

Figure 5 a single linked list to implement the list of chars

  - print() – simple traversal by following references

  - search() – simple traversal e.g. search for E

- this overcomes disadvantages of static implementation at the cost of extra complexity for the dynamic implementation and extra storage for references

  - only have to allocate memory needed

  - insert() - don't have to move lots of data around e.g. insert C: create a new C node and update 2 references

  - delete() - don't have to move lots of data around e.g. delete D: update just a single reference in the previous node

Summary

- an abstract data structure can be implemented statically or dynamically

- dynamic implementations are more efficient, versatile, general, powerful

- created using references, where we build in main memory any shape of data structure that we want!

- our job is to design the most appropriate data structure, that supports the most appropriate algorithm, to solve the problem

Correct, simple, clear

Objective: these are the three essential attributes of all good software. So these are the criteria on which I grade all of your programming labs!

Software Life Cycle

- a software project traditionally used to pass through stages, called the Software Life Cycle (SLC)

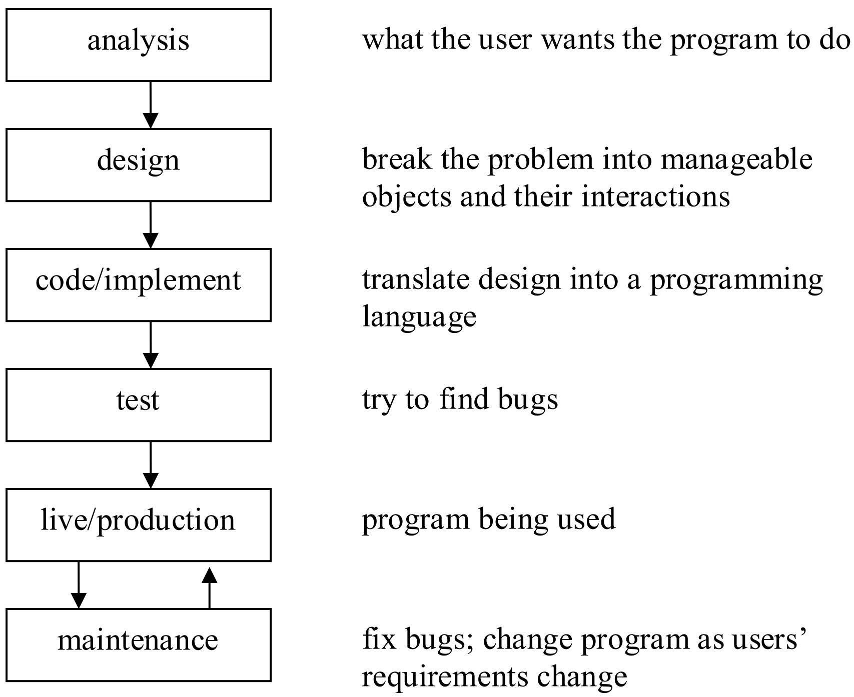

Figure 6 the Software Life Cycle (SLC)

Example – Palomar's PeopleSoft project

- computerize all of Palomar's operations
  - a huge, expensive project handled by a team of people over several years
- analysis – done by systems analysts, working with end users
  - e.g. payroll and payroll law are incredibly complex, and ever-changing
  - computer people know nothing about payrolls
  - payroll people know nothing of programming
  - users typically do not know what they want!

- very important, difficult, frustrating process
- result is confused, incorrect, incomplete, contradictory 'requirements definition' document (2.5% of total project cost)

• design - systems analysts and programmers

- system required is overwhelming in complexity and detail
- break the problem into many simpler, smaller objects and their interactions
- objects in the program correspond to objects in the problem e.g.

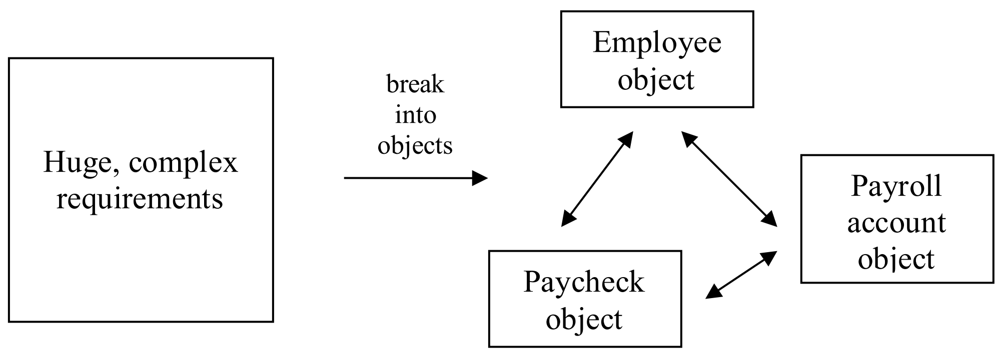

Figure 7 the design stage

- very important, difficult, intuitive, experiential process (5% of cost)

• code – programmers

- translate each object into a programming language
- with experience of programming, easy process! (5% of cost)

• test – programmers and users

- prepare test data and expected output for every possible situation
- apply test data to real program and compare outputs

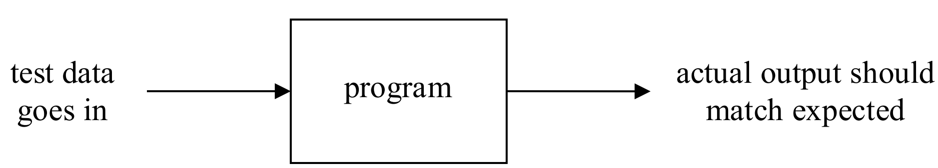

Figure 8 testing a program

- programs are so large and so complex that it is not possible to test every logical path!

- there is no realistic way to prove that software is correct

- difficult, lengthy process (12.5% of cost)

• live / in production – users

  - all conversion, training, documentation done and program handed over for day-to-day use

• maintenance – (inexperienced junior) programmers

  1. fix bugs missed during testing
  2. change program as users' requirements change
  3. fix our fixes ☹

  - the longest stage of a program's life....

  - IMPORTANT: largest part of the project cost! (75% of cost!!!)

  - (after many changes, program can eventually become unmaintainable – we no longer dare change it!)

The three essential attributes of good software

• this amazing 75% figure results in the surprising attributes of good software

  - we must develop code that is not only correct (obviously)…

  - but it must also be maintainable

1. Correct

• hopefully, some of my students go on to careers as professional computer programmers

  - perhaps working on flight control software at Boeing or Airbus

  - if you make programming errors in flight control software, "the plane crashes and everyone dies"…

  - …and this is the standard of correctness I use to grade all of your programs in this class!

Maintainability

- code is maintained by the most junior, inexperienced programmers
  - to change code confidently, must first be able to understand it, therefore...
  - every line of code we write must be simple and clear!...
  - did I write this program as simply and as clearly as I can, or should I take more time to do so?

2. Simple

- what is simple?
  - during design: many simple objects directly representing the important parts of the problem, with simple interactions
  - during coding: simple algorithms

3. Clear

- what is clear?
  - during coding: standard code layout or indentation
    good variable names
    standardized comments required for every class, method

The programmer's mantra – correct, simple, clear

- NOTE: I evaluate and grade all the code you write on these three essential attributes
  - correct
  - simple
  - clear

Summary

- it is important to remember that coding is only a small and straightforward stage of a large and difficult process

- all the code you ever write must be correct, and as simple and as clear as you can make it!

Next week and homework

- next week: intro to stacks

- homework by next week:

  - download, run and understand this week's example programs

  - read Horstmann, section 15.5 on stacks

Lab

- homework assigned this week. See Canvas for due date
- read 'How to write Javadoc comments'
  - Javadoc is an essential part of the clarity of a computer program
  - so Javadoc comments are required in all programming assignments
  - good, full Javadoc comments are required for full credit