APS 105
Spring Term, 2005
Lecture Notes

1 Introduction

1.1 Instructor

- John Carter
  - I am a lecturer
    - teaching
    - writing
    - no research
- education
  - undergraduate — math
  - graduate — computer science
- experience
  - many years teaching high school
  - ECE since 1999

1.2 How to find me

- refer them to information sheet
- what to do if office hours are not suitable
- encourage them to use me as a resource

1.3 Lectures

- refer them to handout for times and locations
- lectures will stay close to text but not conform exactly
- lectures will be posted on course website at the end of each week
  - lectures are not posted in advance
    - explain why

1.4 Tutorials

- one per week
- run by TA's
- TA's will have material to present but questions are welcome

1.5 Labs

- a two-hour period is assigned each week
- TA's will be available for help in the labs during those time slots
- not necessary that they go to the lab
- it is necessary that they submit the labs on time
  - probably no exceptions — unless system goes down, building on fire, etc.
- can do labs from home by connecting to ECF
  - if they have not done so before, they should try to get connected now

1.6 Evaluation

- quizzes — 10%
  total of 10 quizzes
  given in tutorials
  not in first week, last week, or week of midterm test
  best 8 count — each one worth 1.25%
  administered in tutorials
  questions in quiz will be based on lectures of previous week
    (or previous two weeks if no quiz in previous week)
  questions will be similar to those in text
    possibly identical!

- labs — 15%
  a total of eight
  all of them will count
  they must run on ecf — no exceptions
  we use an automarker that is very picky about form of output
    try not to upset the automarker
  half of the marks on labs are for performance, half for style
  I will explain what is required for style as we proceed

- midterm test — 25%
  around end of February
  details shortly

- final exam — 50%
  summative
  emphasis on latter half of course

- problems with labs, quizzes, test
  what to do if you miss something
    form available on course website
  what to do if you are unhappy with your mark on a piece of work
    form available on course website
  what to do if you need special consideration
    form available on course website

1.7 Text

- fabulous!

- we will be following the text fairly closely

- has a few (small) errors
  see link on course website
    `Errors in Text` at left side of page
  if any new errors found, send me a note

1.8 Getting help

- refer them to handout — discuss briefly

- encourage them to form study groups
  have them learn the name, phone number, and email address of a neighbour
    give them time for this

- contrary to what they may have heard, (most of us) care about our students
  we are available and eager to help
  take advantage of tutorials, office hours

- other support
  - counselling and learning skills services offers
    - help with academic skills
    - personal counselling
    - website: www.calss.utoronto.ca
- encourage them to get help *quickly* if they are in *any* kind of trouble
- for those having difficulty
  - don't let things get out of control
    - get help fast
- ask questions in class
  - don't worry about looking foolish — nobody here is stupid
- if you see something on the board that is confusing, ask
  - I do make mistakes — I appreciate having them corrected
- if questions are slowing things down too much I may ask you to see me later
- whatever works (and does not involve cheating)

1.9 Academic integrity

- infractions are treated very seriously
- you may get away with cheating but, if you don't, very bad things can happen to you
- consequences can be very severe — far worse than a poor mark
- see Engineering Calendar pp 164–172 for details of consequences

1.10 In-class rules

- no talking to fellow students
  - why? — creates an aural fog
- try to be on time
  - I will (normally) start on time
  - late arrivals are distracting
- early departures
  - if you are going to have to leave early
    - please inform me ahead of time
    - please sit in a location that minimizes disturbance
- food and drink
- cellphones and other noisemakers

1.11 Using the course website

- URL in handout
  - you will need your student number to log in
- please give your ecf e-mail address on course web site
  - if you usually use something else, simply create a `.forward` file on ECF
  - contents of file are address(es) to which you want mail forwarded
  - a copy of all mail to ECF will be sent to your usual address

1.12 Overview of APS 105

- title is "Computer Fundamentals"
  not simply computer programming

- also looks at some aspects of
  computer organization (very little)
  how information can be stored in a computer
  the concept of an algorithm — a recipe for solving a problem
    algorithms for important tasks like searching and sorting
    how we can compare algorithms — why is one "better" than another

- for some, there will very little here that is new
  these people may wish to forego lectures — no problem
  just make sure you do labs, write tests and quizzes

- for others, the amount of material and the speed may appear to be overwhelming
  we will be asking you to do lots of programming
  labs are only worth 15% but
    the experience is important
    rarely does anybody fail who completed all the labs

- in addition to the lab problems, try to do as many other problems as possible
  this will not be easy given the limited time available but ...

1.13 Java

- appears to be a winner for various reasons
  "object oriented" — more later
  Internet aware and friendly
  easier to produce reliable programs in Java than in many other languages

- I think that we will be using JDK 1.4 on ECF

- can obtain it free from Sun Microsystems
  see Section 1.3 in text
  might be a good idea to get it now before too many other demands on your time

- if you want to develop programs at home and you have Windows
  there are instructions to help you get started on the course website
  see General Handouts: Getting Started in Java

- final version of any program *must* work on ECF
  no exceptions!
  really!

2 Getting Started

2.1 Computer systems

- Two primary divisions
- hardware
  - physical objects that comprise a computer
    - keyboard
    - video display
    - mouse
    - disks
    - etc.
- software
  - sets of instructions, called programs
    - used to tell the computer what tasks to perform

2.2 A closer look at hardware

- all systems consist of three basic components
  - memory
  - processor
  - I/O (input/output)
- tied together physically and logically
- chat a bit about each component
- memory
  - chalkboard analogy — "short term memory"
  - organized into numbered cells — called addresses
  - usually one *byte*/cell
    - a byte is 8 bits — explain
      - a bit is capable of being in one of two states
      - usually denoted by zero or one
      - name comes from: binary digits
    - a byte is 8 bits — 256 possible states — why?
  - memory may contain millions of bytes
- processor
  - sometimes called central processing unit (CPU) or microprocessor
  - the "brains" of the computer
  - does computations
    - discuss registers, accumulators very briefly
  - registers are like displays of very simple calculators
    - separate registers for integer and floating point arithmetic
    - integer arithmetic is faster
  - makes very simple decisions
    - *e.g.* is the contents of a register > 0?
  - uses a "clock" to synchronize operations
    - one very simple operation in one tick of clock
  - density and clock speeds rising rapidly
- I/O
  - allow machine to interact with outside world
  - when we write, we are sending data from memory to an output device
  - when we read, we are bringing data from an input device to memory

- auxiliary storage devices
  hard drives, CD drives, tapes
  sometimes classified as I/O devices

- How the CPU executes a program
  can fetch data from a location in memory
  can store data in a cell in memory
  memory can contain both data and instructions
  instructions are stored sequentially

2.3 Operating systems

- a large program that starts running as soon as the machine is turned on
- acts as the interface between the user and the machine
- allocates resources
- allows users to request the use of a compiler, a word processor, etc.
- when a program stops running, control passes back to operating system

2.4 UNIX

- underlying OS on PC's is DOS
- underlying OS for us is UNIX
- we will be using a command line interface to UNIX
  to those familiar with DOS, this is not much different
  e.g. copy a file: DOS: copy — UNIX: cp
  list files in current directory: DOS: dir — UNIX: ls
- often have a GUI on top of operating system (or as top layer of operating system)
  Microsoft Windows
  X windows
- Lab 0 will get you started with UNIX

2.5 Using High level languages

- different computers, with different chips, have different instruction sets
  e.g. `e2` on one machine might mean "add" while on another it might mean "compare"
- a high level language is one that is more or less the same for all machines
  e.g. FORTRAN, COBOL, Pascal, Turing, BASIC, LISP, C, etc.
- for each computer on which we want to run a program, we need a *compiler*
  a program that translates the high level language into the language of the computer
  produces *machine code* (*object code*) from *source code*
- two steps
  compile program
  run (or *execute*) program

2.6 Using Java

- Java's claim to fame is that it is completely machine independent — even for graphics
  "Write once, run anywhere."

- Java compiler translates program into machine code (called *byte code*)
  resulting code could run on the appropriate machine
  but there is no real machine that will run the code!
  the machine code is that of an imaginary machine

- to actually run the program on a particular machine, an *interpreter* must be written
  each real machine has its own Java interpreter — called the JVM
  interpreter translates from byte code one instruction at a time
  versions of the JVM are available for most machines — free from Sun

- can be slow — about 1/3 speed of compiled C code
  people are working on making this faster

- what is the advantage of such a scheme?
  write a program
  compile it into byte code
  distribute byte code over a network
  don't worry about destination machine

3 Creating Programs

3.1 Getting started with Java

- A first program
  traditionally, writing `Hello, world`

-
```
// This program prints: Hello, world

class Hello
{
  public static void main (String[] args)
  {
    System.out.println("Hello, world");
  }
}
```

- develop this slowly, with lots of explanation (as shown in the following notes)

3.2 Analyzing the features of the first program

- comments
  note alternative forms

- `class Hello`
  always have programs within a class

- `main` method
  programs may contain many methods — working together
  ours only has one
  it must be called `main`
  the first (*header*) line must be as shown
    to be explained later — for now, just write it as shown

- `System.out`
  sends output to standard system output device — the screen
  `println` *vs.* `print` — examples to show difference

- use of `{}`
  to indicate start–end of class
  to indicate start–end of method within class

- whitespace and use of indentation

- Java is case sensitive — explain

- use of + for printing very long strings

- escape sequences
  how do we print `"`?
    use `\"` — an escape sequence
    explain
  to print `\`, use `\\`

3.3 The programming cycle

- Step 1 Edit
  use a text editor, not a word processor
    why?
  examples
    gedit in labs

notepad under Windows
vi, emacs, pico under UNIX
if the program class is `Hello`,
save the program (source code) in the file `Hello.java`

- Step 2: Compile
  run the Java compiler by typing
  `javac Hello.java`
  compiler translates the source code into Java bytecode
  compiler automatically saves bytecode in the file `Hello.class`

- Step 3: Run
  run the java interpreter (JVM) for your machine by typing `java Hello`
  (without `.class`)
  if all goes well, the interpreter should *run* or *execute* your program

- Step 4, 5, 6, ... Debug
  many opportunities for introducing many kinds of errors
  syntax errors (*compile-time errors*)
  to be explained shortly
  javac will catch any syntax errors
  only creates the file `Hello.class` when no syntax errors exist

3.4 Classifying errors

- syntax errors
  "grammar" errors
  give examples in English and in Java

- in natural languages, we can also have *semantic errors*
  semantics of a statement is the meaning attached to it
  a statement in a natural language may be ambiguous
  e.g. "Time flies like an arrow."
  if all is well, computer languages should not have ambiguities
  hence semantic errors should not occur
  (but, occasionally, they have done so)

- *run-time* errors
  e.g. an attempt to divide by zero
  an attempt to read a value from a file when none exists

- *logical errors*
  program compiles and runs but does not do what the programmer intended

- help with errors
  sections at end of most chapters discusses techniques for
  avoiding errors
  eliminating errors (debugging)

4 Primitive Types and Identifiers

4.1 Basic storage ideas

- recall bits: capable of being in one of two states
  bytes: 8 bits — $2^8 = 256$ possible states
- we store data of various kinds as patterns of bits
  the same pattern of bits might represent a variety of things
  e.g. a number, some text, part of a picture, part of a song
- in writing a program, we may, for example, need an integer
  we can then ask Java to reserve space in memory for an integer
  whatever pattern of bits is stored at that location will be *interpreted* as an integer value
- on another occasion, the *same* bit pattern could be interpreted as a colour
  we get whatever interpretation we request
- spend some time on this
  try to make sure that they understand the concept

4.2 Integer types

- examples of integers
- Java has four sizes of integers: `byte`, `short`, `int`, `long`
  prefers `int` — explain
- use of `l` or `L` for long constants
- summary (from text — Page 17)

Type	**Size (in bits)**	**Range**	**Approximate Range**
`byte`	8	-2^7 to $2^7 - 1$	± 100
`short`	16	-2^{15} to $2^{15} - 1$	$\pm 30\ 000$
`int`	32	-2^{31} to $2^{31} - 1$	$\pm 2\ 000\ 000\ 000$
`long`	64	-2^{63} to $2^{63} - 1$	$\pm 9 \times 10^{18}$

4.3 Floating point types

- examples of floating point constants as normal decimals
- examples — include use of e or E notatione
  similar to scientific notation
- floating point values are stored as two parts
  digits (mantissa) and exponent are stored separately
  not whole story — values are stored in binary rather than decimal form
  we don't need to know the whole story
- Java has two forms of floating point numbers
  `float` and `double`
  Java prefers `double` — explain
- use of `f` or `F` for float constants
- summary (from text — Page 19)

Type	Size (in bits)	Precision	Approximate Range
`float`	32 1 for sign 8 for exponent 23 for mantissa	at least 6 decimal digits	$\pm 3.4 \times 10^{38}$
`double`	64 1 for sign 11 for exponent 52 for mantissa	at least 15 decimal digits	$\pm 1.8 \times 10^{308}$

4.4 Characters

- examples
  `'A' '$' '\\' '\n' '"' '\''`
- strings *vs.* characters
- need enough bits to store all the characters we might want
  Java doesn't fool around — uses 16 bits — Unicode
  - why?
  - 2^{16} possible characters
  - no details necessary — see Appendix C if you are curious

4.5 Boolean values

- `true`/`false`
- one bit

4.6 Identifiers

- purpose — to give names to items
- rules
  - any number of characters
  - upper or lower case letters plus digits plus `_` and `$`
  - first character cannot be a digit
- some identifiers reserved by the language — called reserved words:
  e.g. `int`, `double`, `main`, `static`, `if`, `do`
  - see text — p 23
- conventions (not rules)
  - lower case for variables
  - use of lower-upper
    - *e.g.* `valueAtStart`
  - use of underscore as in `value_at_start` — I won't use this
  - class identifiers conventionally written as upper-lower
    - *e.g.* `Hello`
  - upper case for constants as in `PI`
- be sensible about identifiers
  - not too short
  - not too long
- be careful to avoid confusion
  - `0` and `O`
  - `1`, `l`, and `I`

5 Using Variables

5.1 Declaring variables

- general form:
  <type> <identifier list>;
- examples

```
int age;          // in years
float mass;       // in kg - to nearest g
double a,b,c;     // sides of a triangle
char sexCode;     // M or F
```

- what does a declaration do?
  explain
  no *initialization* (explain term) can be expected

5.2 Assigning values to variables

- meaning of an assignment statement — contrast with equality
- can be done as soon as the variable is declared
  e.g. `int i = 0;`
- or at any time after that
  e.g. `int i;` ... `i = 3;`
- conversions and casts
  widening and narrowing conversions
  see table in text (p. 28) for full list of widening conversions
  use of a cast
  floating point to integer — lose fraction
  be careful — *e.g.* `byte b = (byte) 200;`
  compiler will accept this but ...
  can never convert to a `char` without a cast
  cannot involve `boolean` in a conversion

5.3 Strings

later

5.4 Printing values of variables

- floats printed with up to 8 digits
- doubles printed with up to 16 digits
- integer and char values printed in just the right amount of space
- boolean values printed as the words true/false
- concatenation
  recall use of `+` in printing long string constants
  concatenation of variables and string constants

5.5 Reading values using `In`

- discuss concept of "input"
  from where?
  to where?

- Java does not provide the simplest input scheme known

- to solve this problem, a class `In` has been created to make reading values easier

- to read a double into the variable `x`, use
  `x = In.getDouble();`

- to read an integer into the variable `i`, use
  `i = In.getInt();`

- similar for `char`, `String`, `float`:
  `getChar()`, `getString()`, `getFloat()`

- on ECF, can gain access to `In` by copying `In.java` from the file
  `/share/copy/aps105/In.java`
  use `cp` command, as discussed in Lab 0

- `In.java` is also available on the course website
  along with `Out.java` — discussed in Appendix A

- once you have a copy, compile it (using `javac In.java`) to get `In.class`

5.6 Interactive programs

- briefly

- use of *prompts*

5.7 Constants

- literal constants

- defined constants
  how?
    use of `final` attribute
  why?
    clarity
    ease of revision
    avoidance of errors

- identifiers of constants
  use of upper case letters

- discuss concept of *magic numbers*
  note consequences of using them on assignments

6 Mathematics

6.1 Basic arithmetic operations

- four basic operators
  `+ - * / ( )`
  usual *precedence* rules
  do not use square or brace brackets for math
- use of `/` with integers and floating point values
- one more operator: `%`
  explain and do a few examples with `%`
- if types of operands are mixed, wider type is contagious
  do some examples
- can use casts (although not usually necessary)
  casts have precedence higher than other operators
  e.g. `(int) 0.8 * 2.5` gives `0.0`
- division by zero (including `%`)
  integer values
    an *exception* is *thrown*
    explain briefly — no details
  floating point values
    `Infinity`, `-Infinity NaN`

6.2 Assigning expressions to variables

- note sequence: evaluation and then assignment
  look at

```
i = 5;
i = i + 1;
```

- multiple assignments
  applied from right to left
- other assignment operators
  briefly
  examples

```
int i = 3, j = 5;
i += j;
i *= j + 2;
```

6.3 Increment and decrement operators

later

6.4 Printing expressions

later

6.5 Math methods

- we can find square roots, sines, ...
- to do so, we can use built-in methods in class `Math`
  part of Java's *Application Programming Interface — API*
  like mathematical functions
  they, generally, have double arguments and return double values

- some have various versions
  e.g. one for handling integers, another for floating point values
  we say that such methods are *overloaded*
  like + operator
  details in text

- a partial list

Method	**Value Returned**
`Math.sqrt(x)`	$\sqrt{x}$
`Math.pow(x,y)`	x^y
`Math.log(x)`	$\ln x$
`Math.exp(x)`	e^x
`Math.abs(x)`	$\lvert x \rvert$ (`int` OK here)
`Math.floor(x)`	largest integer $\leq x$ (as a `double`)
`Math.ceil(x)`	smallest integer $\geq x$ (as a `double`)
`Math.sin(x)`	$\sin x$ (in radians)
`Math.cos(x)`	$\cos x$
`Math.atan(x)`	$\arctan x$

- also two constants:
  `Math.PI` — π
  `Math.E` — e

6.6 `Math.round`

- converts to nearest integer

- `double` converted to `long`
  `float` converted to `int`

- show how to round to something other than the nearest integer

6.7 `Math.random`

later

6.8 Arithmetic with `char` values

later

7 Other Topics from Chapters 1 and 2

7.1 String variables

- purpose
  storing strings
  like those we have seen
- compare strings *vs.* characters
- declaring strings
  type is `String` (note upper case)
  `String` is a class that defines characteristics of string objects
  same as declaring primitive types
    no value in location after declaration

7.2 Assigning values to string variables

- `String` is a *reference* type
- look at an example
  `String s = "Sample";`
- action is very different from that with primitive types
  draw diagram
- note action on assignment
  Java creates a new string object and sets our variable to refer to it
- explain a reference
  value stored in the variable is a memory reference — the address of the object
- changing assignments
  draw diagram to show effect — old string is *garbage*
  strings are *immutable*

7.3 Increment and decrement operators

- briefly
  examples of both prefix and postfix forms
- don't combine with other operators
  (or read Section 2.3 *very* carefully before doing so)
  perhaps one example

```
i = 3;
j = 5;
k = ++i * j--;
```

7.4 Printing expressions

- simply put the value or expression inside the parentheses
  no double quotes
- explanation
  expression is evaluated
  result is converted to a string
  string of characters is sent to output device

- can concatenate strings with expressions
  we say that + is *overloaded*
  e.g. `System.out.println("Result is " + 3 * 5);`
  `System.out.println("Result is " + 3 + 5);`
  `System.out.println("Result is " + (3 + 5));`

- refer them to Section 2.2, Example 2 in text

7.5 `Math.random`

- method returns a random `double` value in the interval $[0, 1)$

- show (step by step) how to obtain an integer from 1 to 4

7.6 Arithmetic with char values

- brief look at concept of storing char values in 2 bytes
  e.g. `'¿'` is stored as 0000 0000 1011 1111
  this bit pattern can be considered as a base 2 integer
  explain briefly
  then find value of `'¿'` as an integer
  $2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 32 + 16 + 8 + 4 + 2 + 1 = 191$
  they do not need to know details, for now

- we can get numerical values from text — Appendix C
  from there, we can find that value of `'¿'` is 191

- if we write, in a Java program
  ```
  int i = '¿';
  System.out.println(i);
  ```
  program will print 191

- don't need to know numerical values

- should know
  A – Z, a – z, 0 – 9 are each sequential
  can move through the alphabet using arithmetic

- be careful!

```
char row = 'A'
row++;                  // sets row to 'B'
row = row + 1;          // error   explain
row = (char)row + 1;    // error   explain
row = (char)(row + 1);  // OK  explain
```

8 Control Structures for Choosing Alternatives

8.1 Introduction

- flexibility of programs is produced by their ability to take different actions when faced with different conditions

- Java has a variety of decision-making statements
  which is best depends on situation

8.2 `if` statements for choosing between two alternatives

- start with an example — printing square roots
  print square roots of negative numbers with `i`
  e.g. $\sqrt{-4} = 2i$

```
if (x < 0)
  System.out.println(Math.sqrt(-x) + "i");
else
  System.out.println(Math.sqrt(x));
```

- general form

```
if (<expression>)
  <statement>
else
  <statement>
```

- note
  indentation
  parentheses
  semi-colons

8.3 `if` statements without `else` clauses

- for choosing to perform an action only if some condition is true

-
```
if (<expression>)
  <statement>
```

- do an example

```
/* make result positive */
if (result < 0)
  result *= -1;
```

8.4 Blocks

- use of `{..}` to create a *block*

- do an example

```
if (result < 0)
{
  result *= -1;
  System.out.println("Negative result made positive");
}
```

- note indentation style — and alternatives

8.5 A common error

- be careful not to write things like the following:

```
if (result < 0);
  result *= -1;
```

- the ; following the condition is interpreted as a statement
  an *empty* statement
- fragment of code is parsed by compiler as:

```
if (result < 0)
  ;
result *= -1;
```

- thus, any value of result will have its sign reversed
  probably not what we wanted

8.6 Boolean valued expressions

- in if statements we had:
  if (<expression>) ...
- what are rules for forming <expression>?
  generally, <expression> must be a boolean-valued expression
  i.e. value must be either true or false

8.7 Relational expressions

- usually in if statements, we are dealing with relational expressions
  examples of simple relational expressions
  x < 0
  height > width
- concept of a relational expression having a value
  true or false
- list of relational operators
  <   <=   >   >=   ==   !=
  no spaces between tokens
  emphasize difference between x = 0 and x == 0
- can mix types in relational expressions
  e.g. can compare an int to a double
- can mix arithmetic operators and relational operators
  arithmetic operators have higher precedence
  give an example

8.8 Some other possible boolean-valued expressions

- constants
  true or false — silly but possible
- variables
  must be of type boolean
  more later

9 Control Structures for Making Decisions (cont.)

9.1 Comparing characters

- Unicode encoding determines order
- blank is very low — lower than any printable character
- usually we will be dealing with digits and letters
  '0' < '1' < ... < '9'
  'A' < 'B' < ... < 'Z'
  'a' < 'b' < ... < 'z'
- note that '0' == 0 is false
  explain
- other characters are scattered around
  see Appendix C for details

9.2 Comparing strings

- shouldn't use relational operators
  show meaning of s1 == s2 with diagrams
- use of equals method from String class
  note form: s1.equals(s2)
  not: String.equals(s1,s2)
  method returns a boolean value
- use of compareTo
  explain ordering informally
- use of the form
  s1.compareTo(s2) <op> 0

9.3 Boolean operators

- ! (not) && (and) || (or)
- formal definitions via truth tables
- order of evaluation
  precedence: !, &&, ||
  left to right for same operator

9.4 Properties of boolean expressions

- lazy evaluation
  <e1> && <e2>
  here <e2> is not examined if <e1> is false
  <e1> || <e2>
  here <e2> is not examined if <e1> is true
- why is lazy evaluation useful?
  an example — testing if x/y < 10
  must guard against the possibility that $y = 0$
- De Morgan's laws
  (!p) && (!q) ≡ !(p || q)
  (!p) || (!q) ≡ !(p && q)
  lots of time explaining these

10 Control Structures for Making Decisions (cont.)

10.1 Lab 2

- spend a bit of time explaining the idea of a loop

10.2 Nested `if` statements

- develop an example
  determining which of `a`, `b`, `c` is largest
  discuss strategy first

10.3 Dangling else problem

- perhaps just do this briefly
  refer them to text for details
- if there is time, show two pieces of code

```
if (p)                          if (p)
  /* p true */                    /* ??? */
  if (q)                          if (q)
    /* ??? */                       /* ??? */
  else                          else
    /* ??? */                     /* ??? */
```

- get them to fill in truth values in comments
- once they see the problem, look at ways to solve it

10.4 Indentation style with many nesting levels

-
```
if (  )
  <statement>
else if (  )
  <statement>
else if (  )
  <statement>
.
.
else
  <statement>
```

10.5 `switch` statements

- discuss briefly
- not required for this course
- if interested, see Section 3.5

10.6 Conditional operator

- discuss briefly
  do an example
- again, not required for this course
- more details in text — p 644

11 Control Structures for Repetition

11.1 General ideas

- two kinds of repetitive actions (loops)
  counted and conditional
- examples — from cooking
  stir for 300 strokes
  stir as long as there are lumps

11.2 `while` statements

- general form:

```
while (<expression>)
  <statement>
```

- effect — explain slowly and carefully — compare to if
- an example — develop slowly
  a loop to read integers until zero is read and print the sum of values

```
sum = 0;
n = In.getInt();
while (n != 0)
{
  sum = sum + n;
  n = In.getInt();
}
System.out.println("The sum is " + sum);
```

- note initialization before loop to make condition meaningful first time
- note also that
  <expression> must be meaningful before we enter the loop
  <expression> must be `true` if we are to perform the body of the loop the first time
  something in loop must make <expression> `false` if we do not want an *infinite loop*
  value of <expression> after exiting loop must be `false`

11.3 `do` statements

- recall processing of an indeterminate number of students
  as well as "while there are students, keep processing",
  we can also have "keep processing as long as there are students"
- note difference — in second case, we must process at least one student
- for first loop, Java has

```
while (<expression>)
  <statement>
```

- for second loop, Java has

```
do
  <statement>
while (<expression>);
```

- almost like `while` but now
  <expression> is not evaluated until we reach the end of the loop
- an example:
  a loop to force a user to enter a number from one to ten

```
do
{
  System.out.println("Give a number from one to ten");
  n = In.getInt();
}
while (n < 1 || n > 10); // this is what we DON'T want
```

- note
  use of semi-colon at end
  use of parentheses
  loop must be executed at least once (unlike `while`)

11.4 `for` statements

- recall:

```
<exp1>;
while (<exp2>)
{
  <statement>
  <exp3>;
}
```

- this can be written more concisely as:

```
for (<exp1>;<exp2>;<exp3>)
  <statement>
```

- an example: printing a table of squares from one to ten

```
for (n = 1; n <= 10; n++)
  System.out.println(n + " squared is " + n*n);
```

- we can write this in form

```
for (int n = 1; n <= 10; n++)
  System.out.println(n + " squared is " + n*n);
```

- in the second case, `n` is only defined in the `for` statement
  if we tried to print it outside the `for`, we would get an error
  explain, informally, idea of *scope* of a variable

12 Repetition (cont.)

12.1 Variations on `for` loops

- recall general form:

  `for` (<exp$_1$>; <exp$_2$>; <exp$_3$>)
    <statement>

  <exp$_1$> is executed once, at beginning
  <exp$_2$> is boolean, must be `true` for <statement> to be executed
  <exp$_3$> is executed after <statement>

- very flexible

- first and/or third expressions can be empty:
  an example

```
int n = 1;
.
.
for (; n <= 10; n++)
  System.out.println(...);
```

  another example

```
n = 1;
for(;n <= 10;)
{
  System.out.println(...);
  n++;
}
```

- expressions can be complex:

```
for (n = 1; n < 11; System.out.println(...),n++)
  ;
```

- note
  semi-colon on next line (for visibility)
  use of comma expressions
  (only valid in first and third part of `for` statement headers)

- another example:

```
for (m = 1, n = 10; m < n; System.out.println(m*n),m++,n--)
  ;
```

  show output

- recommendations:
  normally,
  keep things simple
  use `while` or `do` with conditional loops
  use `for` with counted loops

12.2 Loops that combine counting and conditional elements

- explain general idea
  e.g. process 100 items (or quit if done before that)

- note that `for` is often used for partially conditional loops
  probably the best solution

- *e.g.* perform an action `MAX` times, unless something special happens

```
boolean done = false;
.
.
for (int i = 1; i <= MAX && !done; i++)
{
  .
  .
  if ( .. )
    done = true; // something special happened
}
```

12.3 Nesting loops

- just as we can nest `if` statements inside other `if` statements,
  we can nest loop statements inside other loop statements
- develop an example — nesting a `for` inside another `for`
  print a triangular pattern containing n rows of numbers from 1 to n
  e.g. if `n` is 5, print

```
1
12
123
1234
12345
```

- develop code to print a triangle of stars like the following

```
****... *
.
.
***
**
*
```

- develop code to print a triangle of stars like the following

```
        *
       **
      ***
        .
        .
*...****
```

- here is one possible solution to the last problem

```
for (int row = 1; row <= n; row++)
{
  for (int col = 1; col <= n; col++)
    if (row + col <= n)
      System.out.print(" ");
    else
      System.out.print("*");
  System.out.println();
}
```

13 Methods

13.1 Introduction

- as problems get larger, they get much harder to solve
- *modularity* is a useful response to this
  explain modularity
  e.g. input/processing/output
- why is it a good idea?
  allows us to think about one part of a large problem (more or less) in isolation
  many people can work on various parts of the problem
  some parts may be identical (or at least very similar) so . . .
  modules from other programs might be useful — "plug them in"
  structure is lost if segments are too large
  may be easy to test each small module
  this may give some hope that the entire package may work

13.2 Method basics

- in Java, the basic modular unit is the method
- many pre-defined methods are available in virtually all programming languages
  we have, for example, in Java, already seen
  math methods available in the class `Math`
  input methods available in the class `In`
- methods can be divided into two types
  queries that evaluate something and return that value to the user
  e.g. given a value, find its square root
  `x = Math.sqrt(y);`
  commands that carry out some process and then simply quit when done
  e.g. print something
  `System.out.println("Hello, world");`
- some languages differentiate between two types
  use terms like
  function for query
  procedure, *subroutine* for command
  not Java — both types called methods

13.3 Mechanics of method usage

- when we use a method, we say that we are *invoking* or *calling* the method
- as an example, consider the statement:
  `discriminant = Math.sqrt(b*b - 4*a*c);`
- what happens when this statement is executed?
  processing of calling block is suspended
  value of $b^2 - 4ac$ is computed
  the *value* of this expression is *passed* as input to the method
  method is executed
  the value computed by the method is *returned* to the point of the call
  control is passed back to the calling block
  the value can then be assigned, printed, compared to another value, etc.

13.4 Defining methods

- we are not restricted to using pre-defined methods — we can define our own
- *e.g.* a definition of $n!$ — n factorial
  recall (or introduce) mathematical definition
  develop the following:

```
public static int factorial (int n)
{
  int result = 1;
  for (int i = 1; i <= n; i++)
    result *= i; // explain
  return result;
}
```

- explain header in detail
  `public static` — as usual (with `main` method)
  `int` — type of value returned by the method
  `int n` — a variable used for values passed from calling block to method
- explain the statement
  `return result;`
  must be at least one of these in a method that returns a value
- since factorial computes an `int` result,
  we can use the function wherever an `int` would be valid
  e.g. `System.out.println(factorial(5));`

```
    value = factorial(2*n);   // value, n both type int
    if (factorial(j) > max) ...
```

13.5 Program organization

- where do we put the definition of a method?
  for now, anywhere within the class that we are using
  as an example, consider a program that calculates values of $\binom{n}{r}$
  may have to explain meaning of this — lead up to program below

```
class Sample
{
  public static int factorial(int n)
  {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;
    return result;
  }

  public static void main (String[] args)
  {
    int m, n, value;
    System.out.println("Give values of n and r")
    // valid input assumed
    n = In.getInt();
    r = In.getInt();
    value = factorial(n)/(factorial(r)*factorial(n-r));
    System.out.println(n + " choose " + r + " is " + value);
  }
}
```

14 Methods (cont.)

14.1 Use of `return`

- definitions of queries (methods that return a value) must contain a statement of the form
  `return` <expression>`;`
- multiple `return` statements can appear in method definitions
  this is often seen as bad style
  why? explain — compare to `break`
  we will usually follow this style rule — occasional exceptions

14.2 Methods that do not return a value

- for commands, type of method is `void` (same as it is in `main` method)
- can simply write `return;` at point at which we wish to return to calling point
- can omit `return` entirely — it will happen when last `}` is encountered (as in main)
- perhaps, as an example, develop

```
public static void skipLines (int n)
{
  for (int i = 1; i <= n; i++)
    System.out.println();
  return;   // can be omitted
}
```

- to call this, we could write
  `skipLines(3);` (not `x = skipLines(3);`)

14.3 Parameters

- used to communicate with methods
  e.g. suppose we have a method with the header:
  `public static int fun (int n)`
  when we write `result = fun(4);`
  then, at the time of the call, the variable `n` in the header automatically gets the value 4
  the variable `n` is an example of a *parameter*
- terminology
  parameters and *arguments*
  sometimes formal parameters and actual parameters
- parameters are known throughout the method but not outside
  since they are not known outside the method,
  their identifiers can be used outside — for other variables
- call by value
  at time of call, value of argument is computed
  this value is passed to the method automatically
  be sure that they understand these ideas (passing and automatic)
  there are other ways to establish correspondence between arguments and parameters
  e.g. C, Turing, Pascal have such mechanisms
  Java, however, always uses call by value
  keep pushing this idea

- develop the following example of parameter passing

  in `main`

  ```
  int n = 7;
  result = fun(n);
  ```

  in method `fun`

  ```
  public static int fun (int n)
  {
    n = n/2;
    return n;
  }
  ```

  trace values of both variables

- automatic correspondence

  if more than one parameter, correspondence is determined by order

  types of actual and formal parameters need not match exactly

  they only need to be assignment compatible

  as an example, suppose we were to write a method to determine powers of numbers

  the header of such a method might have the form

  ```
  public static double power(double x, int n)
  ```

  to call such a method, we might write

  ```
  value = power(1.5,2);
  ```

  could also write

  ```
  value = power(15,2);
  ```

  it would, however, be invalid to write

  ```
  value = power(15,2.0);
  ```

14.4 Method overloading

- briefly

- use an example

  ```
  public static int squareSum (int a, int b)
  public static double squareSum (double a, double b)
  public static int squareSum (int a, int b, int c)
  ```

- discuss results of various calls:

  ```
  squareSum(3,4)
  squareSum(3,5,2)
  squareSum(1.5,2.5)
  squareSum(1.5,4)
  ```

14.5 Methods without parameters

- no problem — simply omit parameters in header

- resulting header has form

  ```
  public static <type> <identifier> ()
  ```

  called by expression of the form

  ```
  <identifier>()
  ```

  like `System.out.println()`

  must have `()`

15 Methods (cont.)

15.1 Methods that return `boolean` values

- briefly
  identifiers are normally of the form `is<Something>`
  e.g. `isPerfectSquare`
  a method to return `true` iff an `int` is a perfect square
- to use such a method, we might write
  `if (isPerfectSquare(n))`
- easier for somebody reading our program to understand what we are doing
- develop definition of method

```
public static boolean isPerfectSquare (int n)
{
  if ((int)Math.sqrt(n)*(int)Math.sqrt(n) == n)
    return true;
  else
    return false;
}
```

- note use of multiple `return` statements in method
- could replace this with something like

```
public static boolean isPerfectSquare (int n)
{
  boolean result = false;
  if ((int)Math.sqrt(n)*(int)Math.sqrt(n) == n)
    result = true;
  return result;
}
```

- a more concise solution:

```
public static boolean isPerfectSquare (int n)
{
  return (int)Math.sqrt(n)*(int)Math.sqrt(n) == n;
}
```

- spend some time explaining this

15.2 Scope

- *scope* of an identifier is the range in which it is recognized
- recall from previous work that, in a statement like
  `for (int i = 1; ... ; ...)`
  the variable `i` is only known inside the `for`
  i.e. the scope of `i` is the `for`
- the scope of a parameter is its entire method
- the scope of any variable declared within a method
  starts at the point of declaration
  ends at the end of the *block* in which the variable was declared
  a block starts with `{` and ends with `}`
- give examples of variables declared within sub-blocks of methods
- note that we can re-use identifiers if we are out of their scope
  use examples to illustrate this

15.3 Putting the pieces together

- recall idea of using methods to produce modularity
  think about the problem as consisting of smaller pieces
- explain strategy of top-down design briefly
  decomposing a problem into a set of smaller problems
- to begin to solve a large problem
  we might try to describe the process using *pseudo-code*
    explain briefly
- perhaps think of each of smaller pieces as being made up of even smaller ones
- continue this process until we have nice, easy to contemplate pieces
  write these as methods, then combine them
- explain Goldbach's Conjecture briefly
- start top-down design for solution to problem of testing conjecture
- refer them to Section 5.7 in the text for details

16 Objects

16.1 General introduction

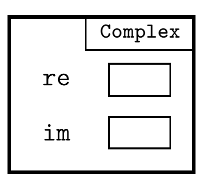
- objects are “things” that can have data and functionality
- *e.g.* cars, people
- our objects will be considerably simpler

16.2 Defining a class

- use the example of complex numbers to develop ideas
- recall that a complex number has the form
  $z = a + ib, a \in \mathbb{R}, b \in \mathbb{R}$
  a is the *real* part of z
  b is the *imaginary* part of z (although it is a real number)
- start with a very basic version of the `Complex` class

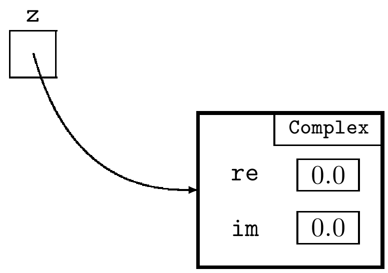
```
class Complex
{
  public double re;
  public double im;
}
```

- instance fields
  cf. variables
  why *instance*?
  because every object (every *instance* of the class) will have these fields
  why public?
  same as methods
- note that the field declarations are not contained in a method
- this simply defines what a `Complex` object should look like
  it does *not* create such an object
  think of it as a “blueprint” for an object of type `Complex`
  sometimes described as a “template”
  it is the *type* of the objects that we shall create
  draw a diagram



16.3 Creating objects

- now we can create objects of this type by writing, in `main`, say
  (emphasize that we have two different classes in separate files)
  `Complex z;`
  `z = new Complex();`
- this creates an object — an instance of the class `Complex`
  draw a diagram



- variable `z` — a reference to an object — an instance of the class
  same phenomenon that we have already seen with `String` objects
- note that the fields are initialized to zero
  this is generally true
    numeric fields initialized to zero
    `char` fields initialized to all zeros
    boolean fields initialized to `false`
    `String` (and other object) fields initialized to `null`
      explain `null` briefly
- multiple instances
  we can create as many objects of type `Complex` as we wish
- to create two `Complex` objects, we could write

```
Complex z1 = new Complex();
Complex z2 = new Complex();
```

- draw diagrams

16.4 Assignments with objects

- having created a `Complex` object, we can assign values to the real and imaginary parts
- develop an example

```
class TestComplex
{
  public static void main (String [] args)
  {
    Complex z1 = new Complex();
    Complex z2 = new Complex();
    z1.re = -3.0;
    z1.im = 4.0;
    z2.re = 2.4;
    z2.im = 1.0;
      etc.

  }
}
```

- add to diagrams
- show effect if we write

```
Complex z3;
z3 = z1;
```

17 Objects and Methods

17.1 Instance methods

- objects can have both data and functionality
  to achieve functionality, we add methods to our class definitions
- for our example, the `Complex` class might include a method to find the modulus
- recall that, if $z = a + ib$, then the modulus is defined by $|z| = \sqrt{a^2 + b^2}$
- we can include this method by extending our `Complex` class as follows:

```
class Complex
{
  double re;
  double im;

  public double modulus()
  {
    return Math.sqrt(re*re + im*im);
  }
}
```

- note that
  there is no `static` in the header
  there is no named parameter to the method
- to call this method, we could write, in the `main` method, say
  `double size = z1.modulus();`
  to assign `size` the value 5.0
- what is happening here?
  we have an *instance* method with an *implicit* argument — the object `z1`
  spend lots of time explaining this
- note that we must still have `()` even though there is no explicit parameter

17.2 `this`

- inside `modulus`, the implicit `Complex` argument has no name
  if we really need a name for this object, Java lets us use "`this`"
- using `this`, we *could* have written the definition of `modulus` with the statement
  `return Math.sqrt(this.re*this.re + this.im*this.im);`
  no need here to write `this` so we did not do so
- we can also use `this` in calls to instance methods
  as an example suppose we want to write a method to classify complex numbers
  large if $|z| > 1$
  small if $|z| \leq 1$
- to do this we could write

```
public String size ()
{
  if (this.modulus() > 1)
    return "large";
  else if (this.modulus() < 1)
    return "small";
  else
    return "just right";
}
```

- we might call this method by writing
  `System.out.println(z1.size());`
- here, again, the use of `this` is optional

17.3 Parameters to instance methods

- we can have parameters with our instance methods
  they operate in the same way as they did with our previous methods
- *e.g.* modify previous `size` method to compare to a value other than one

```
public String size (double standard)
{
  if (modulus() > standard)  // implies this.modulus()
    return "large";
  else if (modulus() < standard)
    return "small";
  else
    return "just right";
}
```

- we might call this method by writing
  `System.out.println(z1.size(3));`
- parameters can be of any type — including objects

17.4 Methods returning objects

- we can have methods that return object references
- as an example, consider a method to find and return the sum of two complex numbers
- show form of call to this method
  `Complex z3 = z1.plus(z2);`
- then develop a definition of the method

```
public Complex plus (Complex other)
{
  Complex sum = new Complex();
  sum.re = re + other.re;
  sum.im = im + other.im;
  return sum;
}
```

- another example, showing another application of `this`
  a method that returns a reference to the larger of two `Complex` objects

```
public Complex larger (Complex other)
{
  if (modulus() > other.modulus())
    return this;
  else
    return other;
}
```

18 Methods (cont.) and Constructors

18.1 Lab 3

- extend deadline to Monday
- discuss approach to problems
- look at some aspects of Part 2 — $\sin x$

18.2 Altering objects in methods

- recall concept of call by value
  consequence — cannot change argument value
- with objects, we again pass value — now value of *reference*
  consequence — we can change the contents (fields) of an object
  draw diagrams
- an example

```
public void scale (double factor)
{
  re *= factor;
  im *= factor;
}
```

- now suppose that `z1` refers to an object representing $-2 + 3i$
  draw a diagram
  if we then make the call `z1.scale(2);`
  show what happens using diagrams
- take lots of time

18.3 Constructors

- note that when we created a complex number, we wrote
  `Complex z = new Complex();`
- here `Complex()` looks like a method call
  it is, in fact, exactly that
- the method is supplied by Java — not necessary for us to write it
  called a *constructor* method
- it is the method that constructs a new `Complex` object
- we could have written it if we wanted to
  it would look like this
-
```
public Complex ()
{
  re = 0.0;
  im = 0.0;
}
```

- note that
  name of the constructor method is the same as the name of the class
  no return type is specified
  implicitly, constructor returns an object — the current object (`this`)
  no return statement in constructor

19 Constructors (cont.), Class Fields, and `toString` Methods

19.1 Customized constructor methods

- recall work of last day on default constructors
- be sure that they understand that this is supplied to them by Java
- sometimes we want to be able to do more than this with our constructors
- as an example, to create `Complex` objects initialized to any value (not just $0 + 0i$)
  we would want to create such objects with a call like
  `Complex z = new Complex(3.0,-4.0);`
- to do so, we could write the following constructor

```
public Complex (double r, double i)
{
  re = r;
  im = i;
}
```

- note that if we had called the parameters `re` and `im`, we would have a problem
  what would `re = re;` mean?
- could solve this by writing
  `this.re = re;`
  or, as we did, by choosing different identifiers for parameters

- as another example, we can define a constructor to create new, distinct objects identical to other objects

```
public Complex (Complex z)
{
  re = z.re; // or this.re = z.re;
  im = z.im; // or this.im = z.im;
}
```

- then, if we have a `Complex` object `z1`, we could write
  `Complex z2 = new Complex(z1);`
  to form `z2`, with values identical to `z1`
  would `z1 == z2` be true in this case?
- note difference between
  `Complex z2 = new Complex(z1);`
  `Complex z4 = z3;`

19.2 Replacing default constructors

- if we create our own constructor, we lose the “free” one
  if we still want to have the effect of the default, we must write it
  if we do write such a constructor, that is fine
  it could have the form

```
public Complex()
{
  re = 1.0;
  im = 1.0;
}
```

19.3 Working with multiple constructors

- no confusion between constructors — different signatures
- in our example, we can now write
  `Complex z1 = new Complex();`
  or `Complex z2 = new Complex(0.4,2.1);`
  or `Complex z3 = new Complex(z2);`
  first calls one constructor, second and third call others

19.4 Class fields

- recall that every object we create has its own copies of all instance fields
  sometimes we only want one copy of a field for the whole class
  these are called class fields
  to create such a field, we use the *modifier* `static`
- *e.g.* a variable in `Complex` that counts the number of complex objects created

```
class Complex
{
  public static int numComplex = 0;
  public double re;
  public double im;

    etc.
}
```

- then have constructors update the count

```
public Complex (double r, double i)
{
  re = r;
  im = i;
  numComplex++;
}
```

- then, in `main` method
  `System.out.println(Complex.numComplex);`
  would print current number of complex number objects we had created
- note differences in usage
  class field: <class>.<identifier>
  instance field: <object>.<identifier>
- Class fields and the `final` modifier
  a frequent use of class fields is with values that represent constants
  for these, we add the `final` modifier
  e.g. in the Math class, Java has
  `public static final double PI = 3.141592659...;`
  `public static final double E = 2.7182818...;`
  we can create such fields for our own classes
  e.g. `public static final int SIZE = 100;`

19.5 `toString` methods

- `print/println` converts arguments to strings
  for primitive types, no problem
- for objects, not quite so simple
  Java has a method called `toString` that converts objects to strings

- problem is that this general method for objects is not usually very useful
  it simply prints the class identifier and the address of the object
- as an example, if we had written
  `Complex z1 = new Complex(-4,3);`
  then `System.out.println(z1);`
  might print `Complex@4A236C`
- to get something more useful, we can override the default with our own `toString` method
  one for each class whose values we want to print
- *e.g.* for `Complex` class, with fields `re` and `im`,
  we might want to print `<value of re> + <value of im>i`
- to do so, we can, in the `Complex` class, make the following definition

```
public toString ()
{
  return "" + re + " + " + im + "i";
}
```

20 `equals` Methods, Visibility, and Object Summary

20.1 `equals` methods

- recall use of `equals` methods for strings
- objects of any class have a default `equals` method
  not very useful — acts like `==`
- we usually replace default with our own version
- might simply check if all fields are equal
  appropriate for `Complex` class
- develop `equals` method for `Complex` class
  include (and explain) check that explicit object parameter is not `null`

  ```
  public boolean equals (Complex c)
  {
    return c != null && re == c.re && im == c.im;
  }
  ```

- might be more complex
  perhaps check that `double` fields are within 1% of each other

20.2 Visibility modifiers — hiding information

- the *modifier* `public` means that a field or method is visible everywhere — in any class
  we may want some methods or fields to be invisible (and unavailable) outside the class
  to do so, we can declare them to be `private`
- why make something `private`?
  don't want another class to be able to alter it
  should only be used here — allows us to *encapsulate*
- *accessor* and *mutator* methods
  if we make a field `private`, how can we gain access to it from outside the class?
  use *methods* that are `public`
- why?
  permit controlled access only — users cannot mess things up
- give examples of accessor methods for `Complex` class
  `getRe`, `getIm`
- we may want to allow *controlled* alteration of fields from outside
  for this we use *mutator* methods
  use the example of a `Circle` class to illustrate utility of mutator methods
- chat about default visibility
  if no visibility modifier used then result is `package` visibility
  we haven't seen packages yet
  for us, a package is all classes in the current working directory
- don't give full rules — too messy — not needed for this course

Accessible to:	public	protected	package	private
Same class	yes	yes	yes	yes
Class in same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass in different package	yes	no	no	no

20.3 Comparison of class/instance fields/methods

- brief discussion
- give an example of use of both kinds of method to add complex numbers
- make sure they understand differences

20.4 Summary

- what have we achieved with our creation of a `Complex` class?
  we can now operate with complex numbers without worrying about details
  e.g. add, subtract, multiply, etc.
- this is a commonly seen feature
  hide details — user can focus on interactions between objects
- look briefly at some aspects of `Fraction` class in text
- refer them to Section 6.8 in text
  note examples — no details of fraction arithmetic

21 Arrays

21.1 Introduction

- idea of a group name for a collection of items of the same type (any type)
- *e.g.* marks on assignments

Asst	1	2	6
Mark	85	92	73

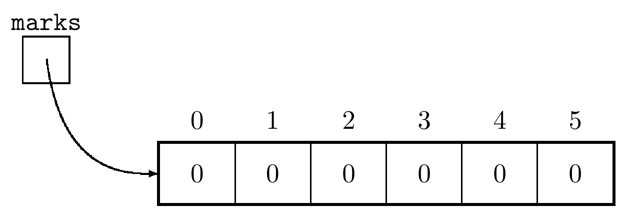
- math analogy: $x_1, x_2, \ldots, x_n$
- terminology:
  index (rather than subscript) (plural: indices)
  component or *element*

21.2 Creating arrays in Java

- index starts at 0 — no choice
- *e.g.* an array to store 6 marks would have components called
  `marks[0], marks[1], ... , marks[5]`
- to create an array for our six marks, we could write

```
int[] marks;
marks = new int[6];
```

- this is just like the creation of any other object
  the first line tells the compiler what kind of object marks will be — an `int` array
  the second line asks the compiler to actually create an instance of such an array
  and to have `marks` refer to it
- after second line, we have, pictorially



the variable `marks` stores the address in memory of the first cell of the array

- we could, and usually do, combine declaration and initialization
  `int[] marks = new int[6];`

21.3 Automatic initialization

- just as fields of an object are automatically initialized when object is constructed
  elements of arrays are also automatically initialized when array is created
- same initialization values we saw for object fields
  0 for integers
  0.0 for floating point values
  0000 0000 0000 0000 for `char`
  `false` for `boolean`
  `null` for objects

21.4 References to arrays

- suppose we have written `int marks = new int[6];`
- if we now were to write
  `int[] myMarks = marks;`
  we would *not* get a new array
  only get `myMarks` referencing the same array as `marks`
  i.e. we would copy the address in `marks` to the variable `myMarks`
- if we want `myMarks` to refer to a different array, we could write
  `int[] myMarks = new int[6];`

21.5 Initialization of arrays

- can initialize arrays in declarations (as we did for previous types)
- *e.g.* `int[] list = {5,7,10};`
- this avoids the use of `new`

21.6 Operating with arrays

- suppose we have an array declared by:
  `double[] list = new double[MAX];`
- don't need to know `MAX`
  can (and usually do) use the field `list.length`
- setting the elements of an array to one

```
for (i = 0; i < list.length; i++)
  list[i] = 1.0;
```

- note: `i < list` is preferable to `i <= list.length - 1`
- summing the components of an array

```
sum = 0;
for (i = 0; i < list.length; i++)
  sum += marks[i];
```

- reversing the elements of an array
  develop the following slowly

```
for (bottom = 0, top = list.length - 1; bottom < top; bottom++, top--)
{
  temp = list[bottom];
  list[bottom] = list[top];
  list[top] = temp;
}
```

22 Multi-Dimensional Arrays

22.1 Two-dimensional arrays

- concept
  it is often useful to be able to store data in a tabular form
- we can easily create such a structure in Java
  ```
  int[][] table;
  table = new int[3][4];
  ```
- *can* think of this as 3 rows, 4 columns
- really what we have here is an array of 3 items, each of which is an array of 4 ints
- draw diagrams of a table and three rows of four cells
- note identifiers
  ```
  table
  table[i]
  table[i][j]
  ```
  be sure that they are clear on uses and differences

22.2 Initialization

- ```
  int[][] table = {
                    {4,5,7},
                    {2,3,8}
                   };
  ```
- note: semi-colon, commas
  inner braces can be omitted but not a great idea

22.3 Non-rectangular 2-D arrays

- it is possible to have, for example,,

  ```
  int[][] raggedArray = {
                          {2,3,5},
                          {4,7},
                          {1},
                          {9,3,4,5,2}
                         };
  ```
- can use `length` field on any 2D arrays
- for the array above,
  `raggedArray.length` is 4 (# of rows)
  `raggedArray[1].length` is 2 (# of items in row 1)

22.4 Examples

- determine the result of the following code

  ```
  int i, j;
  int[][] table = new int[4][4];
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      table[i][j] = i - j;
  ```
- print a table using nested for statements
  use various orderings of rows/cols
  ```

- ```
  for (int i = 0; i < a.length; i++)
  {
    for (int j = 0; j < a[i].length; j++)
      System.out.println(a[i][j]);
    System.out.println();
  }
  ```

  etc.

22.5 Higher dimensional arrays

- no problem
- think of 2D as a table with rows and cols
  3D — book containing pages of tables
  4D — a set of books of pages of tables
  etc.
- few details — see text

22.6 Arrays of objects

- suppose we want to set up an array of 100 complex numbers
- we could start by writing, in `main`, say

  ```
  public static final int MAX = 100;
  Complex[] list = new Complex[MAX];
  ```

- this would give an array of 100 references — each one initialized to `null`
  draw a diagram to illustrate this
- to make each of these actually point to an object of type `Complex`, we could write

  ```
  for (int i = 0; i < MAX; i++)
    list[i] = new Complex();
  ```

- all the resulting `Complex` objects would be initialized to represent $0 + 0i$
  draw a diagram to illustrate this
- if we wanted to give the 5th element the value 2 + 7i, we could write

  ```
  list[4].re = 2;
  list[4].im = 7;
  ```
  ```

23 Strings

23.1 Review of strings

- so far, we have seen `String` constants (literals)
  e.g. `"Hello, world"`
- we have also seen `String` variables
  creating

  TTab `String s1 = "Hello";`
  can also use long form
  TTab `String s1 = new String("Hello");`
  explain difference
- strings are *immutable* objects
  e.g. `"Hello"` cannot be changed
  however, `s1` can be changed
- for example, we were to write
  `s1 = "hello";`
  then `s1` would now refer to the new string `"hello"`
  the old string `"Hello"` would not be alterd
- if we write
  `s2 = s1;`
  then this causes `s2` to refer to the same string to which `s1` is currently referring

23.2 Earlier string methods

- `boolean equals (String s)`
  return `true` iff implicit string and parameter are identical
- `int compareTo (String other)`
  returns < 0 if implicit `String` precedes explicit parameter
  returns 0 if implicit `String` is identical to explicit parameter
  returns > 0 if implicit `String` followss explicit parameter
- look at some examples using `compareTo`
  `"cat".compareTo("cast")` returns `> 0`
  `"CAT".compareTo("cat")` returns `< 0`

23.3 New methods from the `String` class

- `int length ()`
  if we have
  `String s = "Sample";`
  then `s.length()` returns the value 6, the number of characters in `s`
- *cf.* arrays: `list.length`
  with arrays, `length` is an instance *field*
  with strings, `length` is an instance *method*
- `char charAt (int position)`
  returns the character at the given position in a string
  for same string as in previous example,
  `c = s.charAt(3);`
  assigns `c` the value `'p'`
  (because we start counting from zero, as we do with arrays)
  similarly,
  `s.charAt(0)` would return `'S'`

- `int indexOf (char c)`
  returns index of leftmost occurrence of `c` in in its implicit object
  returns −1 if no such occurrence
  `indexOf` method is overloaded
  second form has header
  `int indexOf (char c, int i)`
  this form starts looking at index `i`
  e.g. if we have `String t = "Toronto";`
  then `t.indexOf('o',2)` returns 3

- do an example — counting all occurrences of a character `c` in a string `s`
  soluion using `charAt`
  solution using `indexOf`

- `String substring (int start, int pastEnd)`
  `String substring (int start)`
  returns a string that is a substring of the argument
  first form returns a string beginning at `start` and up to (but not including) `pastEnd`
  e.g. `s.substring(1,4)` returns `"amp"`
  note that `pastEnd` − `start` gives length of substring
  second form goes to end of string

- an example — printing all the words in a `String`, one per line
  assume that:
  there is exactly one blank separating words,
  there are no leading or trailing blanks,
  there is no punctuation

- leave this with them for now

24 Strings (cont.)

24.1 Midterm test

- return tests
- chat briefly about marks

24.2 Problem from last day

- recall problem of printing words of a string one word/line
- develop solution slowly, with backtracking
  many possibilities — here is one

```
s = s + " ";
int start = 0;
int pastEnd = s.indexOf(' ');
while (start < s.length())
{
  System.out.println(s.substring(start,pastEnd));
  start = pastEnd + 1;
  pastEnd = s.indexOf(' ',pastEnd + 1);
}
```

24.3 `String` methods (cont.)

- `String trim ()`
  returns a string with all whitespace removed from beginning and end of its string
  whitespace consists of blanks, tabs, and newlines
  e.g. `String s = "   Sample      ";`
  `s1.trim()` returns `"Sample"`
- `String toLowerCase ()`
  this method returns a lower case version of its string
  e.g. `s.toLowerCase()` returns `"sample"`
- `String toUpperCase ()`
  returns an upper case version
  e.g. `s.toUpperCase()` returns `"SAMPLE"`
- `boolean equalsIgnoreCase (String s)`
  returns `true` iff implicit string and explicit parameter are equal (if case is ignored)

24.4 Arrays of strings

- no new problems
  show how to create an array of strings
  use long way
  `String[] names = new String[3];`
  show diagram of result of this
  `names[0] = "Anna";`
  `names[1] = "Ben";`
  `names[2] = "Chloe";`
  show diagram
- then use short way — declaring and initializing at the same time
  e.g. `String[] toppings = {"pepperoni", "mushrooms", "cheese"};`
  show diagram

- discuss command line arguments briefly
  relate them to header of `main` method
    refer them to the text for more details
  note that command line arguments will not be on exam

25 Strings (cont.)

25.1 Strings from things

- we can obtain a string representation of anything
- for an object, we can use the object's `toString` method
- for primitive types, we can use the method `valueOf`
  a class method in the `String` class
- method is overloaded so that we can pass any primitive type as argument
- examples
  `String.valueOf(-27)` returns `"-27"`
  `String.valueOf('$')` returns `"$"`

25.2 Chaining of methods

- suppose that we have
  `String s = "Sample";`
  it is valid to use expressions like `s.toUpperCase().indexOf('A')`
    in this case, this will return the value 1
- explain
- note that this does *not* change string referred to by `s`

25.3 A common error with strings

- suppose we want to change the string `s` whose value is currently `"Sample"` to `"Simple"`
- we could try writing
  `s.substring(1,2) = "i";`
- show why this fails
- show correct way to do this
  `s = s.substring(0,1) + "i" + s.substring(2);`

25.4 Strings as parameters

- no problem
  can pass any data type as a parameter
- in passing an array or a `String`, the action is similar to that for any other reference type
  we pass the value of the *reference* to the object
- as an example, suppose we want a method to attach asterisks to each end of a string
  try writing code

```
public static void addStars (String s)
{
  s = "*" + s + "*";
}
```

- show why this fails and show correct version

25.5 Arrays and the swapping problem

- (this material should have been done with arrays)
- recall attempt to swap elements

```
public static void swap (double x, double y)
{
  double temp;
  temp = x;
  x = y;
  y = temp;
}
```

- trace this in attempting to exchange two elements of array `list` using the call
  `swap(list[i],list[j]);`
- be sure that they understand why this doesn't work
- attempt #2

```
public static void swap (double[] a, int i, int j)
{
  double temp;
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

- again, make sure that they understand why this *does* work

25.6 The `StringTokenizer` class

- mention briefly
- details in text — §9.4
- note that they are not responsible for this

26 Searching

26.1 Intro to searching

- problem
  given a list of values, each of which has a *key* (usually unique)
  we want to search for an item with a particular key
- in practice, we will usually be dealing with objects in which one field is the key
  for our purposes, we will simply search an array of values, called list for item
- after searching for an item, we may want to do a variety of things with it
  alter it
  print it
  delete it
  etc.
- to make the methods that we create as flexible as possible,
  we will simply have them return the index in the list of the item
  return an impossible index value (-1) if we fail to find the item

26.2 Sequential search (linear search)

- explain algorithm
- develop basic Java version

```
int seqSearch (double item, double[] list)
{
  int location = -1;
  for (int i = 0; i < list.length; i++)
    if (list[i] == item)
      location = i;
  return location;
}
```

- how can we improve this?
- develop improved version

```
public static int seqSearch (double item, double[] list)
{
  int location = -1;
  boolean found = false;
  for (int i = 0;i < list.size && !found; i++)
    if (item == list[i])
    {
      location = i;
      found = true;
    }
  return location;
}
```

- also look at and discuss shorter version

```
public static int seqSearch double item, double[] list)
{
  for (int i = 0; i < list.length; i++)
    if (item == list[i])
      return i;
  return -1;
}
```

26.3 Binary search

- if a list is sorted, we can take advantage of this in our searching
- lead up to idea of binary search
  each time we can eliminate half the remaining items
    in fact, more than half — can eliminate middle item as well as one half
- do an example
  part 1 — item in list
  part 2 — item not in list
- how do we stop the search?
  if successful, no problem
  if we fail, how do we know?
    bottom and top pointers close in on each other
      once we get down to one, if we fail there, pointers cross over

26.4 Implementation of binary search

- recall ideas from last day
- develop code

```
public static int binSearch (double item, double[] list)
{
  int low = 0;
  int high = list.length - 1;
  int location = -1;
  boolean found = false;

  while (low <= high && !found)
  {
    middle = (low + high)/2;
    if (item == list[middle])
    {
      location = middle;
      found = true;
    }
    else if (item < list[middle])
      high = middle - 1;
    else
      low = middle + 1;
  }
  return location;
}
```

27 Sorting

27.1 Introduction to sorting

- chat briefly about variety of sorting techniques
  fairly vague discussion
- we will be examining four or five sorting algorithms
  they only need to know how to implement the first
  for others, they should understand the algorithms
  not required to code them

27.2 Selection sort

- discuss idea and give an example
- develop code for selection sort slowly
  start with code for one pass
  then wrap this in an outer loop

```
public static void selectSort (double list[])
{
  int top, largeLoc, i, temp;
  for (top = list.size - 1; top > 0; top--)
  {
    // find largest from 0 to top
    largeLoc = 0;
    for (i = 1; i <= top; i++)
      if (list[i] > list[largeLoc])
        largeLoc = i;

    // put largest at top
    temp = list[top];
    list[top] = list[largeLoc];
    list[largeLoc] = temp;
  }
}
```

- note possible variation
  could start by putting smallest in first location
  seen in some texts
- sort is easily adapted to finding top ten, bottom three, etc.

27.3 Insertion sort

- discuss algorithm informally and give an example
- do not give code — it is in text if they are interested (§10.3)
- point out tricky bit
  insert when we find a smaller value or when we get to front of list
- note that it may take a lot of work to implement algorithm if list gets long

27.4 Bubble sort

- analogy of children ordering themselves by height in a line
  if each child is happy with immediate neighbours, entire line is ordered
- discuss algorithm and give an example

- do not give code — again, it is in text if they want it (§10.5)
- note problem with small values taking many passes to get to final location
  - *cocktail shaker* sort helps to some extent
    - discuss it briefly
- generally, bubble sort is very slow — don't use it

27.5 Shellsort

- a clever use of repeated insertion sorts
- much faster than any of the other sorts we have seen
- *not* on our course — in text for those interested (§10.6)

28 Recursion

28.1 Introduction to recursion

- recursion in expression evaluation — BEDMAS
  - try to get them to see the recursive aspect of the algorithm
- essential features of any recursive process
  - process is defined in terms of a "simpler" version of itself
  - must be some simple case that stops the recursion
    - a *base* case or *simple* case

28.2 Recursion in everyday processes

- consider problem of walking across room
  - one step at a time
- old way of thinking: a loop
  - keep taking steps until we reach the wall
- new way of thinking

  to WALK ACROSS A ROOM
      if we are at the far wall
          we are done // simple case
      else
          take one step
          WALK ACROSS THE REST OF THE ROOM

28.3 Recursively defined functions

- defining and evaluating a term
- consider an example something like

$$f(n) = \begin{cases} 2f(n-1)+1 & n > 0 \\ 3 & n = 0 \end{cases}$$

  analogy with phoning and putting people on hold

- note inefficiency of recursion with functions
  - recursive *statement* of function definition may be very simple
  - recursive *implementation* is very inefficient
    - recursion should *not* be used for this purpose
- as we shall see later, there are situations that
  - have the simplicity of recursion
  - are still efficient

28.4 Implementing recursion in Java

- computing values of

$$f(n) = \begin{cases} 2f(n-1)+1 & n > 0 \\ 3 & n = 0 \end{cases}$$

- develop following method
  - assuming valid argument values

```
public static int f (int n)
{
  int value;
  if (n == 0)
    value = 3;
  else
    value = 2*f(n-1) + 1;
  return value;
}
```

28.5 Another example $n!$

- recall non-recursive definition
  suggest computation using a loop
- show how to form recursive definition — with base of $0! = 1$
- develop following code

```
public static int factorial (int n)
{
  int value;
  if (n < 0)
  {
    System.out.println("Invalid argument to factorial - 0 returned");
    value = 0;
  }
  else if (n == 0)
    value = 1;
  else
    value = n * factorial(n-1);
  return value;
}
```

- note that we could also write the definition as:

```
public static int factorial (int n)
{
  if (n < 0)
  {
    System.out.println("Invalid argument to factorial - 0 returned");
    return 0;
  }
  if (n == 0)
    return 1;
  return n * factorial(n-1);
}
```

- chat briefly about merits of each form

28.6 Binary search revisited

- recall non-recursive form of algorithm
- in recursive form, think of each copy of the method working on part of the problem
- new header: `public static int binSearch (String[] list, String item, int low, int high)`
- leave problem of developing rest of code to next class

29 Recursion (cont.)

29.1 Recursive form of binary search

- recall header of last day — complete code

```
public static int binSearch (String[] list, String item, int low, int high)
{
  int location;
  if (high < low) // failure - item not in list
    location = -1;
  else
  {
    int middle = (low + high)/2;
    if (item.compareTo(list[middle]) == 0) // success
      location = middle;
    else if (item.compareTo(list[middle]) < 0) // try bottom half
      location = binSearch(item,list,low,middle-1);
    else // try top half
      location = binSearch(item,list,middle+1,high);
  }
  return location;
}
```

- note that it is a nuisance to use this method
  first call might take the form
  `position = binSearch(names,name,0,names.length-1);`
- we can eliminate this problem by breaking solution into two parts:
  an initially called method to get things started
  a recursive *helper* method to take over after the initial call
- method called by user has form

```
public static int binSearch (String[] list, String item)
{
  return binSearch(item,list,0,list.length-1);
}
```

- most of the work is done by the helper method
- usually make recursive helper method `private`
  in this way, the user does not need to worry about specifying bounds (required by recursion)

29.2 Towers of Hanoi

- show problem with something visual (mixing bowls work well)
- develop algorithm and then show code

```
public static void moveTower (int height,int start,int finish)
{
  if (height == 1)
    System.out.println(start + " ---> " + finish);
  else
  {
    int other = 6 - (start + finish); // explain!
    moveTower(height-1,start,other);
    System.out.println(start + " ---> " + finish);
    moveTower(height-1,other,finish);
  }
}
```

- look at timing briefly
  universe is to end when monks finish their task
  show that one move per second gives about 5.8×10^{11} a
  compare to time since big bang — about 1.4×10^{10} a
- note that algorithm is exponential — such algorithms are totally impractical

29.3 Printing patterns recursively

- *e.g.* printing a row of stars

```
public static void printRow (int n)
{
  if (n < 1)
    System.out.println();
  else
  {
    System.out.print('*');
    printRow(n-1);
  }
}
```

- *e.g.* printing a triangle of stars of the form

```
*****
****
***
**
*
```

- develop method to print this
  note that base case (`n == 0`) requires that we do nothing

```
public static void printTriangle(int n)
{
  if (n > 0)
  {
    printRow(n);
    printTriangle(n-1)
  }
}
```

- perhaps also look at problem of printing an inverted triangle

```
*
**
***
****
*****
```

- show how we can adapt previous method

30 Recursion (cont.)

30.1 Quicksort

- discuss algorithm and illustrate it with cards
  do *not* develop the following code

```
private void quickSort (double[] list, int low, int high)
{
  double pivot = list[low];
  int left = low;
  int right = high;

  if (low < high)
  {
    while (left < right)
    {
      while (list[right] >= pivot && left < right)
        right--;
      list[left] = list[right];

      while (list[left] <= pivot && left < right)
        left++;
      list[right] = list[left];
    }
    list[left] = pivot;   // or right - they are equal

    quickSort(list,low,left-1);
    quickSort(list,right+1,high);
  }
}
```

- do, however, show header that would be required for recursive calls
- make recursive form of method a helper method
  with extra parameters for bounds
- user calls the following method

```
public static void quickSort (double[] list)
{
  quickSort(list,0,list.length - 1);
}
```

- helper method then works recursively, calling itself as necessary

30.2 Euclid's algorithm for gcd's

- a classic example of recursion
- develop basis of algorithm with class
  to find gcd(m, n)
    let gcd be g
      then $m = ga$, $n = gb$
      g will also be a factor of $m - n = ga - gb$
- do an example: gcd$(20, 8)$

- formalize algorithm

$$\gcd(m,n) = \begin{cases} m & \text{if } m = n \\ \gcd(n,m) & \text{if } m < n \\ \gcd(n, m-n) & \text{if } m > n \end{cases}$$

- show how rules are applied to determination of $\gcd(8, 20)$
- develop code

```
public static int gcd(int n)
{
  if (m == n)
    return m;
  if (m < n)
    return gcd(n,m);
  return gcd(n,m-n);
}
```

- see text (p. 435) for a more efficient version of gcd

30.3 Recursion with strings

- can think of a string as a character followed by a string
  (or a character preceded by a string)
  (or two characters enclosing a string)
- by thinking in these ways, we can develop recursive algorithms
- as an example, consider the problem of reversing a string
- outline an algorithm for this
- refer them to text (p 447) for details

30.4 Backtracking

- very brief outline of concept
- refer them to text (§11.6) for details and examples

30.5 Determining powers efficiently

- if there is time, examine problem of determining x^n efficiently
- start by looking at problem of minimizing number of multiplications to find x^n
  e.g. $x^8, x^{20}, \ldots$
- try to get them to see that

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ \left(x^{\frac{n-1}{2}}\right)^2 \times x & \text{if } n \text{ is odd} \end{cases}$$

- then develop Java code

```
public static double power (double x, int n)
{
  if (x == 0)
    if (n == 0)
      return NaN;
    else
```

```
      return 0.0;
  if (n < 0)
    return 1.0/power(x,-n);
  if (n == 0)
    return 1.0;
  double temp;
  if (n % 2 == 0)
  {
    // n is an even positive value
    temp = power(x,n/2);
    return temp * temp;   // spend some time explaining this
  }
  else
  {
    // n is an odd positive value
    temp = power(x,(n-1)/2)
    return x * temp * temp;
  }
}
```

31 ADT's and Linear Lists

31.1 ADT's

- often it is useful to separate the problems of
  what we want to do
  how we are going to do it
- to look at the first part, we can use ADT's
- define ADT informally
  a collection of data and abstract methods to manipulate the data
- an example: the `BigFraction` class
- the specification of the `BigFraction` class as an ADT was in the handout
  operations were addition, subtraction, etc.
- from a user's point of view, `BigFraction` objects and operations are part of Java

31.2 Separating ADT's and implementations

- allows us, when using the ADT, to treat it in a natural way
  add two fractions, etc.
  don't worry about how it is done
  we are simply working with fractions
- gives us flexibility
  if we decide that an implementation is flawed, we can replace it
  need not change anything else
- think of the ADT as a contract between the developer and the user
  the user only needs to know what external behaviour to expect
  this is described by the ADT
  developer can then proceed in any way that is compatible with the contract

31.3 The linear list ADT

- an example
  we could define an ADT for lists of integers
    structure
      a finite sequence of nodes
      since a sequence, order is implied
      a node is an undefined term — some collection of data
    possible operations
      start a new, empty list
      insert a new item into the list
      delete an item with a given value
      search for an item in the list
      print the list
      etc.
- to *implement* such an ADT, we could use
  individual cells
  arrays
  other structures — as we shall see shortly

31.4 Introduction to linked lists

- we have seen that we can create linear lists using arrays
- a problem with such an implementation
  insertions and deletions require a great deal of data movement
- a solution to this is to use linked allocation
- draw a diagram showing a linked list of nodes
- show insertion, deletion pictorially

31.5 Implementation of linked lists

- suppose that the items to be stored in the list are integers
- each node of the list will contain two fields
  `info` — an information field
  `link` — a reference to the next node
- we can create a class to define these objects

```
class Node
{
  int info;
  Node link;
}
```

- note — no public/private modifier — more soon
- to create nodes that contain initial values, we could add a constructor

```
class Node
{
  int info;
  Node link;

  Node (int item, Node next)
  {
    info = item;
    link = next;
  }
}
```

31.6 Adding nodes to a linked list

- suppose that we want to create such a list
- we could write, in our `main` method, for example,
  `Node first = new Node(7,null);`
  to give us a new element containing 7 in its `info` field
- illustrate result
- if we want to build a list. we could do so by writing
  `first.link = new Node(4,null);`
- show diagram
- we could continue with
  `first.link.link = new Node(8,null);`

- show new diagram
- again
  `first.link.link.link = new Node(2,null);`
- add this diagram
- this, clearly, gets silly
- we need some other solution

31.7 Avoiding excessive `link` references

- one solution to this problem:
  try to build the list at the head, rather than at the tail
- develop an example — with lots of diagrams
- start with
  `Node second = new Node(8,null);`
- then add a node in front of this
  `second = new Node(5,second);`
  lots of time explaining this
- then add a third node
  `second = new Node(2,second);`

32 Linked Lists (cont.)

32.1 Introduce a `List` class

- last day, we talked about a `Node` class
- we do not, however, want a user to be aware of the implementation
  could be an array, could be a linked list
- the user should deal with a `List` class
  public methods there should be only thing user sees
- last day we saw that the beginning of a linked list was a reference to the first node
  we put this reference into a `List` class
  call it `head` — refers to the beginning or head of the list
- we can now start to define a `List` class

```
class List
{
  private Node head;
}
```

- show diagram of a list
  a `List` object with `head` referring to first `Node` object
- perhaps add constructors

```
class List
{
  private Node head;

  public List ()
  {
    head = null;
  }

  public List (int item)
  {
    head = new Node(item,null);
  }
}
```

- show results of executing (in `main`) the statements

```
List list1 = new List();
List list2 = new List(4);
```

32.2 Integrating the `List` and `Node` classes

- we want a user to see only the interface methods, not the list structure
  this presents a visibility problem for the `Node` class
- if we make fields of `Node` class `private`,
  they cannot be seen from outside the `Node` class
  so how can they be seen from within the `List` class?
- if we make them `public`,
  everybody can see (and alter) them
- Java's solution is to allow us to create *inner classes*
  in our case, the class `Node` is defined within the class `List`
  fields and methods of inner class can only be seen from enclosing class

- result is:

```
class List
{
  private Node head;

  class Node
  {
    int info;
    Node link;

    Node(int i, Node l)
    {
      info = i;
      link = l;
    }
  } // end of inner class

  // constructors and methods of List class
     .
     .
     .
}
```

32.3 Moving along a linked list

- we cannot always avoid the problem of looking along the list
- as an example, suppose we have created a linked list
  with an arbitrary number of nodes
- show diagram
- suppose we want a method that prints the items in the list
- develop the following (carefully and slowly)

```
public void printList ()
{
  Node current = head;
  while (current != null)
  {
    System.out.println(current.info);
    current = current.link;
  }
}
```

32.4 Insertion into the tail of a linked list

- we can adapt this traversal technique to many situations
- as an example, a method to insert a new node as the last node of a list
- develop the following:

```
public void insertAtTail (int item)
{
  if (head == null)
    head = new Node(item,null);
  else
  {
```

```
      Node temp = head;
      while (temp.link != null)
        temp = temp.link;
      temp.link = new Node(item,null);
    }
  }
```

- note that we control the loop using `current.link != null`
  before we used `current != null`
  why the change?

33 Practice with Linked Lists

33.1 Deletion from front of a list

- draw diagrams: before and after
- develop code

```
public void deleteFirst ()
{
  if (head != null)
    head = head.link;
}
```

33.2 Deletion from rear of a list

- draw diagrams
- spend lots of time showing that traversal must stop at second last node
- note that there are now two special cases
  empty list
  list with a single node
- develop code

```
public void deleteLast ()
{
  if (head != null)
    if (head.link == null)
      head = null;
    else
    {
      Node temp = head;
      while (temp.link.link != null)
        temp = temp.link;
      temp.link = null;
    }
}
```

33.3 Searching for an element in a list

a method to determine whether or not an item is in a list

- ```
  public boolean contains (int item)
  {
    Node current = head;
    boolean found = false;
    while (current != null && !found)
      if (current.info == item)
        found = true;
      else
        current = current.link;
    return found;
  }
  ```
  ```

33.4 Insertion into an ordered list

- draw diagrams: before and after
- show need to look ahead in comparisons
  could use `temp.link.link` approach as in tail deletion
- as an alternative, use two references that traverse the list in tandem `current` and `previous`
- draw diagrams showing `previous` and `current` on either side of gap
  easy to change links once we have this situation
- sketch process
  note that lists with zero nodes or one node may be special cases
- refer them to code in text, more or less like the following

```
public void insert (int item)
{
  Node current, previous;
  boolean found = false;

  current = head;
  while (!found && current != null)
    if (item < current.info)
      found = true;
    else
    {
      previous = current;
      current = current.link;
    }

  Node newNode = new Node(item,current);
  if (current == head)
    head = newNode;
  else
    previous.link = newNode;
}
```

33.5 Deletion of all occurrences of an item from a list

- if there is time, develop the following
-

```
public void deleteAll (int item)
{
  while (head != null && head.info == item)
    head = head.link;
  if (head != null)
  {
    Node current = head.link;
    Node previous = head;
    while (current != null)
    {
      if (current.info == item)
        previous.link = current.link;
      else
        previous = current;
      current = current.link;
    }
  }
}
```

34 Stacks

34.1 Informal introduction

- a linear list in which all insertions/deletions take place at one end (the top)
- analogy with dishes in a cafeteria
- also called a LIFO (last in — first out) list
  also called a push-down store
- already encountered in discussion of recursion
  calls to friends pile up
  when we return from a call, we return to the last copy in the list
- also can be seen in many other contexts
  e.g. `Back` function in a web browser

34.2 The ADT stack

- a linear list with the following operations:
- `new` create a new, empty stack
- `push(item)` insert a new item onto the top of the stack
- `pop()` remove the item at the top of the stack and return it
  if stack is empty, signal in some way
- `peek()` return value of item at top — don't remove it
- `isEmpty()` return true/false if stack is empty/non-empty

34.3 Implementing a stack with an array

- intuitively, it should not be hard
- for implementation details, see text pp.332–334

34.4 Implementing a stack with a linked list

- again, suppose that each node is to store an integer
  nodes could be defined in the usual way — as an inner class
- show *some* of the following to implement the ADT stack

```
class Stack
{
  private Node top;

  public Stack ()
  {
    top = null;
  }

  class Node
  {
    int info;
    Node link;

    Node (int i, Node n)
    {
```

```
      info = i;
      link = n;
    }
  }

  public void push (int i)
  {
    Node temp = new Node(i,top);
    top = temp;
  }

  public int pop ()
  {
    if (top == null)
      throw new RuntimeException("Stack underflow");
    else
    {
      int i = top.info;
      top = top.link;
      return i;
    }
  }

  public int peek ()
  {
    if (top == null)
      throw new RuntimeException("...");
    else
      return top.info;
  }

  public boolean isEmpty ()
  {
    return top == null;
  }
}
```

34.5 Introduction to queues

- a linear list with
  all insertions made at one end (the *rear*)
  all deletions made at the other end (the *front*)
- also called a FIFO list
- term is used in English for a line of people at a bus, a bank, etc.
- a democratic regime — item waiting longest is served next
- an example — a LAN with a single printer
  requests for printing come at unpredictable times
  if server is free, request is processed
  if busy, put request in a queue (the print queue)

34.6 The ADT queue

- a finite sequence of items (first at *front*, last at *rear*)
  with the following operations:

- `new` create a new, empty queue
- `enqueue(item)` add a new item at the rear of the stack
- `dequeue` remove the item at the front of the queue and return it
  if queue is empty, signal in some way
- `peek` return value of item at front of the queue
  leave queue unchanged
- `isEmpty` return `true/false` if queue is empty/non-empty

34.7 Implementation using arrays

- chat very briefly about this
  they are *not* responsible for this implementation
- note problems — not too much detail
- problems can be solved
  often a useful representation

34.8 Implementation using linked lists

- similar to `Stack` class but now we need two references:
  to front and rear
- show how a queue could be represented using a linked list
- where should we put front and rear?
  convince them that `front` is at head of list, `rear` is at tail
  each node is "looking back" in the queue
- empty queue?
  either `front` or `rear` set to `null`
- then create a class `Queue` with *some* methods:

```
class Queue
{
  private Node front, rear;

  public Queue ()
  {
    front = rear = null;
  }

  class Node
  {
    int info;
    Node link;

    Node (int i, Node n)
    {
      info = i;
      link = n;
    }
  }

  public void enqueue (int item)
  {
```

```
    Node temp = new Node(item, null);
    if (rear != null)
      rear = rear.link = temp;
    else
      rear = front = temp;
  }

  public int dequeue ()
  {
    if (rear == null)
      throw new RuntimeException("...");
    else
    {
      int temp = front.item;
      if (front.link == null)
        rear = null;
      front = front.link;
    }
  }

  etc.

}
```

34.9 Applying queues

- if there is time, show how to implement quicksort using queues
  no recursion
- recall idea behind quicksort
- recursion helped us because we did not have to keep track of all the sub-intervals
  we can use queues and do the bookkeeping ourselves
- an example: 15 10 18 6 23 20 9 4 27 24 16 7
  carry this through three partitions
- then develop algorithm:
  enqueue boundaries of original unsorted list
  while queue is not empty
  dequeue boundaries of an unsorted list
  partition it (placing one item in its final position)
  enqueue both sub-lists (if they contain more than one item)
- develop code for method

```
public static void quickSort (int[] list)
{
  // Create a queue, initialized with original array's bounds
  Queue bounds = new Queue();
  bounds.enqueue(0);
  bounds.enqueue(list.length-1);

  // loop while there are still unsorted sub-lists
  while (!bounds.isEmpty())
  {
    // Get bounds of unsorted sub-list
    int left = bounds.dequeue();
    int right = bounds.dequeue();
```

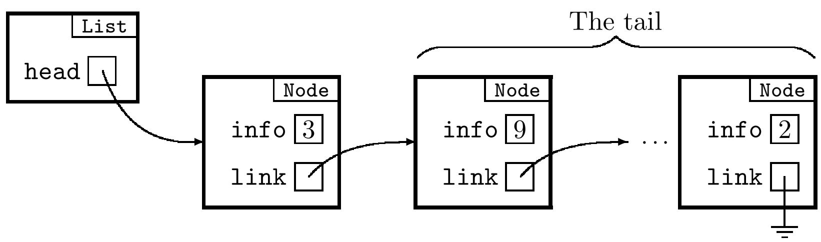
```
    // Partition it (method not shown)
    int mid = partition(list,left,right);

    // Save bounds of non-trivial sub-lists after partitioning
    if (left < mid-1)
    {
      bounds.enqueue(left);
      bounds.enqueue(mid-1);
    }
    if (mid+1 < right)
    {
      bounds.enqueue(mid+1);
      bounds.enqueue(right);
    }
  }
}
```

35 Linked lists and Recursion

35.1 Recursive structure of a linked list

- we can think of a linked list as being composed of two parts
  the first node — the head
  the other nodes — the tail
- show them a diagram



- thinking in this way, we can develop recursive algorithms to process a list
  take care of the head
  process the tail recursively
- simple case?
  the empty list

35.2 Printing a linked list recursively

- as a first try, consider

```
public void printList ()
{
  if (head != null)
  {
    System.out.println(head.info);
    head.link.printList();
  }
}
```

- we could then call this by writing

```
List myList = new List();
  .
  .
myList.printList();
```

- analyze this, with diagrams, to show them what is wrong with it
  `myList` of type `List`, `head.link` of type `Node`
- then show, slowly and carefully, how to fix it using inner classes
- in the `List` class
  a method to
    check if list is/is not empty
    call the recursive method if there is something to print

```
public void printList ()
{
  if (head != null)
    head.printList();
}
```

- in the `Node` class
  a recursive helper method to print a *non-empty* list

```
void printList ()
{
  System.out.println(info);
  if (link != null)
    link.printList();
}
```

35.3 Insertion into an ordered list

- develop, slowly and carefully
- in the `List` class
  a method to
  insert a node into an empty list
  insert a node at the front if appropriate
  call the recursive version otherwise

```
public void insert (int item)
{
  if (head == null || item < head.info)
    head = new Node(item,head);
  else
    head.insert(item);
}
```

- in the `Node` class
  a recursive helper method to insert a node
  into the *tail* of a *non-empty* list

```
void insert (int item)
{
  if (link == null || item < link.info)
    link = new Node(item,link);
  else
    link.insert(item);
}
```

35.4 Searching for an item in a list

- a boolean method that searches for an item
- in the `List` class

```
public boolean contains (int item)
{
  boolean result;
  if (head == null)
    result = false;
  else
    result = head.contains(item);
  return result;
}
```

- in the `Node` class

```
boolean contains (int item)
{
  boolean result;
```

```
  if (info == item)
    result = true;
  else if (link == null)
    result =  false;
  else
    result = link.contains(item);
  return result;
}
```

36 Binary Trees

36.1 Introduction

- linear lists are one-dimensional
- sometimes relationships among data have more dimensions
  e.g. airline networks 2D
  lattice structures — 3D
- a common 2D structure is a hierarchical one
  show a tree
- for many problems, we can restrict ourselves to a simpler structure
  show a binary tree
- introduce some nomenclature (informally, with an example)
  root, leaves, children (left and right), *parents, siblings, descendants, ancestors*

36.2 Formal definition of a binary tree

- give definition and note recursive aspects
- to obtain an ADT, we need operations
  could have
  print all nodes
  insert an item
  delete an item
  etc.
  depends on application

36.3 Traversals

- recall list traversals
- now many possible orderings
- introduce ideas of preorder, inorder, postorder traversals
  examples of traversals of all three kinds
  note that there are other possibilities — depends on application

36.4 Array implementation of binary trees

- fairly brief
- children of item at index i are located at $2i + 1$ and $2i + 2$
- fine for bushy trees — bad for stringy trees
  show why this is so

37 Binary Trees (cont.) and Binary Search Trees

37.1 Linked implementation of binary trees

- show diagram pointing out that
  the root is referenced by a field in a `BinaryTree` object
  the other nodes are referenced by values in `Node` objects
  similar to setup with linked lists

- show basics — fields and constructors of `BinaryTree` class and inner `Node` class

- show how to perform a preorder traversal, printing values in `info` fields

- first call is passed a `BinaryTree` object (implicitly — as an instance variable)

- if null, nothing to do
  otherwise, call traversal of non-null tree

```
public void prePrint () // in class BinaryTree
{
  if (root != null)
    root.prePrint(); // in class Node
}
```

- subsequent calls are passed references to `Node` objects
  implicitly — as an instance variable

- each call prints value at root (of sub-tree)
  and call traversals of non-null sub-trees

```
public void prePrint () // in class Node
{
  System.out.println(info);
  if (lChild != null)
    lChild.prePrint();
  if (rChild != null)
    rChild.prePrint();
}
```

37.2 Binary search trees

- give definition
  clarify difference between a binary tree and a BST
  a BST is a type of binary tree
  not all binary trees are BST's

- give an example
  draw a tree structure, put 50 at the root
  have them supply valid values for other nodes

37.3 Traversal of a binary search tree

- note that an inorder traversal visits nodes in ascending order

- do an example

38 Binary Search Trees (cont.)

38.1 Insertion in a binary search tree

- develop following code
- note the way the code uses an inner class
  to encapsulate the structure and methods

```
class BinSearchTree
{
  private Node root;

  public BinSearchTree ()
  {
    root = null;
  }

  // Initiate insertion of a new node into a tree
  public void insert (int item)
  {
    Node newNode = new Node(item,null,null);
    if (root == null)
      root = newNode;
    else
      root.insert(newNode);
  }

  class Node // an inner class
  {
    Node lChild;
    int info;
    Node rChild;

    Node (int i, Node l, Node r)
    {
      info = i;
      lChild = l;
      rChild = r;
    }

    // Insert a new node into its correct position
    // in a non-empty binary search tree
    void insert (Node newNode)
    {
      if (newNode.info < info)
        if (lChild == null)
          lChild = newNode;
        else
          lChild.insert(newNode);
      else
        if (rChild == null)
          rChild = newNode;
        else
          rChild.insert(newNode);
    }
  } // end of Node class
} // end of BinSearchTree class
```

38.2 Deletion from a binary search tree

- a nasty business — too hard for us
- sketch ideas
  note that they are not responsible for deletion

38.3 Notes on final examination

- old exams are posted on course website
  solutions will be posted about a week before the exam
- they are responsible for all material taught in lectures
  emphasis will be on material since midterm
- text references:
  Chapters 1–6 and 8–12
  omit: §§ 8.5, 9.4, 10.6, 10.7, 10.8, 11.6, 11.8
- exam is on April 20, at 2:00
  I will be available for extra help during the preceding week
  try to arrange some days and times that are convenient for them
    class agreed on
      Monday April 18, from 12 to 2
      Tuesday April 19, from 12 to 2
  if they need help at some other time, send me e-mail