

David M. Keil, Framingham State University
CSCI 252 Computer Science II

Background

1. Java compilation and syntax
2. Standard data types
3. Loops and debugging
4. Classes and objects
5. Precalculus concepts

1. Java compilation and syntax

- 0.1a Explain the Java virtual machine*
- 0.1b Explain the fetch-execute cycle*
- 0.1c Describe Java syntax*
- 0.1d Identify the steps in system development*
- 0.1e Explain code documentation*




The Java virtual machine

- The Java compiler translates Java code to an assembler-like language called “byte code”
- The JVM is a program that interprets byte-code instructions, simulating a real processor
- The *java* program at the command line, and any Internet browser, contain JVMs
- The *class loader* in *java* allows program statements from different *.class* files to invoke each other

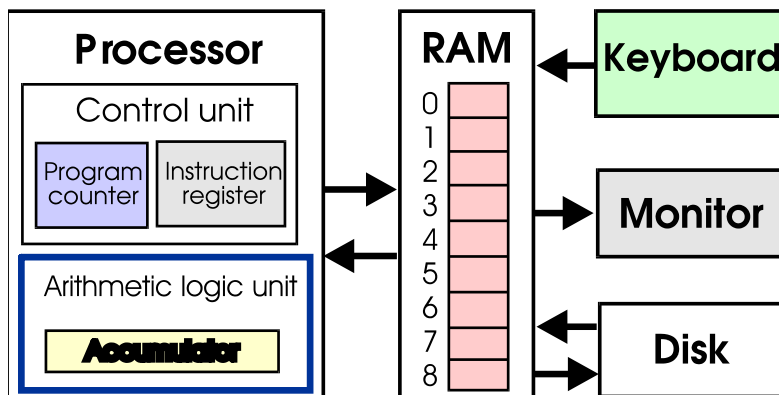
Virtual machines

- Any interactive program or operating system is an *interpreter of commands* that lets computer hardware emulate a specialized machine
- *Examples:*
 - UNIX command interpreter is platform independent;
 - Java VM in Web browsers executes downloaded platform-independent *byte code*
- *Issue:* Java VM’s security – Does VM permit byte code to write to disk, send email, etc.?

Gates

- Used to manipulate binary data
- 1 or 2 bit input, 1-bit output
- Specified using truth tables
- NOT (negation) 
- AND (conjunction) 
- OR (disjunction) 
- Used as components of combinational circuits:
NAND, NOR, XOR, adders, etc.

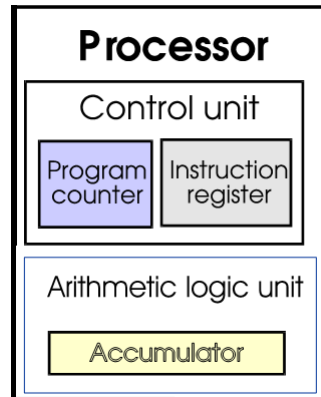
A simplified model computer



- RAM (random-access memory)
contains programs and data

The processor

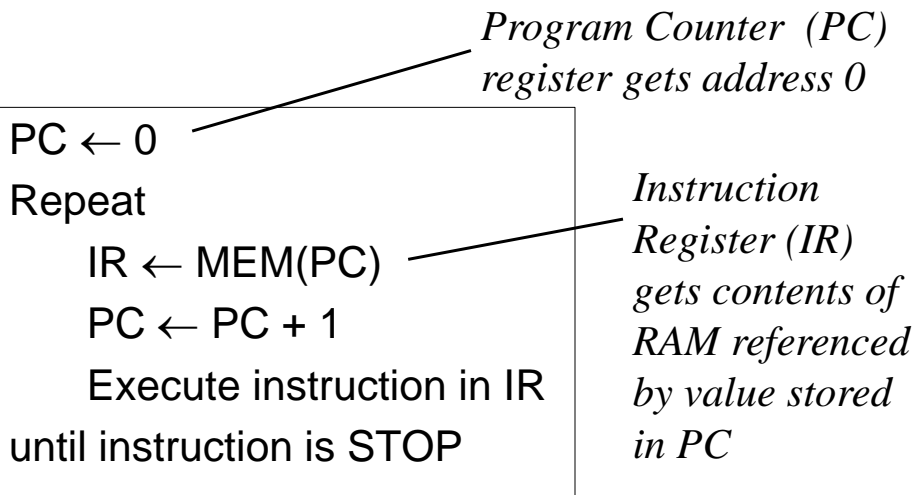
- Initiates all actions
- Has two units:
 - *Control*: Determines order of operations
 - *Arithmetic logic*: Executes operations on data
- Communicates with memory
- Has three registers: program counter, instruction register, accumulator



Machine-language programs

- *Instructions* are represented in binary *operation codes*
- These may have *operands*, which specify *address* of data to be operated on
- All operations are simple
- Instructions and data are stored in consecutive RAM locations

The fetch-execute cycle



Programming languages

- Syntax:
 - rules for forming *tokens* (e.g., delimiters, operators, IDs, numerals)
 - rules for putting tokens together (e.g., statement, expression, program)
- Semantics: Meaning; i.e., a mapping from *structure of program to machine actions* (machine-code statements)

Object-oriented languages

- Data are *objects*, defined by *attributes* and *behaviors* (“methods”)
- Objects send and receive *messages*
- Classes are templates for objects
- Features:
 - *encapsulation* (data hiding);
 - *inheritance* (derived classes inherit attributes and methods from base classes);
 - *polymorphism* (meaning of a message depends on class of recipient)
- *Examples*: Smalltalk, Simula, C++, Java

Higher-level languages

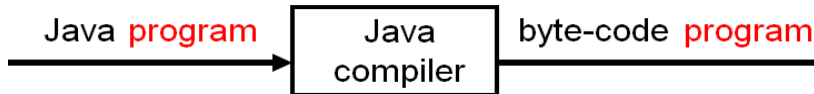
- Support I/O, control structures, and modularity
- Shield programmer from hardware and operating-system details
- Are *portable* (compilable to any runtime environment)
- Are translated to machine language or byte code by compilers or interpreters
- *Examples*: COBOL, Fortran, Pascal, C, C++, Java

Interpreting vs. compiling

- *Interpreted code* is executed one instruction at a time from input stream (machine code, Java byte code, command line)



- *Compilers* translate code from high-level languages like Java to low-level form that can be interpreted



Integrated development environments

- *Editor* enables code entry and modification, with syntax highlighting
- *Compiler* translates Java to machine code or byte code; provides *warnings*; *error diagnostics*
- *Debugger* enables trace of variables
- *Help systems* provide reference
- *Examples*: NetBeans, BlueJ, Eclipse
- *Java Development Kit* (Sun) provides compiler, debugger

Projects

- Most IDEs organize Java programs as *projects* consisting of multiple source-code files
- *Examples:* BlueJ, Eclipse, NetBeans
- Often developers create one source file per Java class, compile source files separately, link compiled *.class* files
- If *.jar* file is produced, then it is executable alone if the Java runtime environment is on the computer

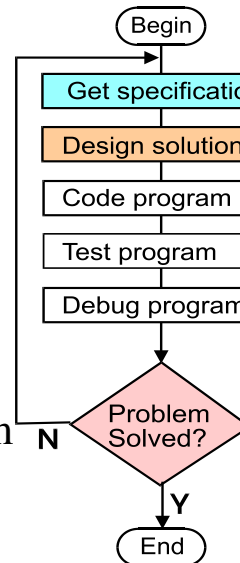
Projects and object files (C, C++)

- *Project:* a set of source files ready to link together after compilation
- Programmer selects files to include based on class and function dependencies
- Compilation produces intermediate linkable object files (*.obj* or *.o*), which may or may not define *main*; some function addresses may not be resolved
- *Build* command automatically compiles any source files modified since last compile

The system development process

Phases (may repeat):

- *Analysis:* specifies input and output
- *Program design:* prepares algorithms, data structures
- *Coding:* implements design as a *program* in a language
- *Testing:* evaluates working program
- *Maintenance:* addresses errors and needs not found in previous stages



Standards for system specification

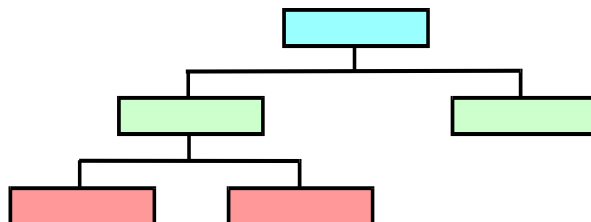
- *Specification* (requirements document) gives:
 - input
 - output
 - correspondence between the two
- User interface should be specified
- If input is via *file* or *port*, specify this
- Any *repeated* steps should be stated

Design concerns

- *Simplicity* (via abstraction, structure)
- *Performance* (throughput, response time)
- *Reliability* (redundancy, recovery, integrity)
- *Evolvability* (adaptation to changes in function and scale)
- *Security* (access control privacy, authentication)
- ***Design principles***: abstraction, information hiding, modularity, packages, version control, divide and conquer, layering, hierarchy, reuse, interfaces, encapsulation, virtual machines

Modular decomposition

- *One strategy*: divide and conquer
- All programming languages support modularity
- An *organization* is modular



- Modular design may be *top-down*
- *Subprograms* implement modular designs

Documentation guidelines

- State *purpose* of every program and every component of large programs at top
- Give meaningful names to *variables*
- Use well-named *constants*
- Use *comments* to clarify intention
- *Format* source code for clarity
- To be debugged or maintained, a program must be understandable

Formatting source code

- *Example:*

```
void main()  
{  
    System.out.println("Hello");  
}
```
- Leave an empty line before a method definition such as *main*
- Align pairs of braces vertically
- Indent statements 2-3 spaces
- *Readability* is a major concern

Documenting exercises

- In this course, all programs are to be documented by comments
- Purpose of program is to be at top
- Each block of code is to have a comment stating its purpose
- Comments include your name, the date, and reference to course objective

Specifying grammar rules

- A language is a set of strings, e.g., the set of all possible C++ programs
- A grammar is a set of rules for what is permitted in a language
- Java *tokens* are formed by simple rules; e.g., an integer literal is a series of digits
- Higher-level (*nonterminal*) components (*program, statement, expression, etc.*) are built from tokens or other nonterminals

Ways to specify syntax

- Plain English (e.g., “A compound statement is a series of statements, in braces”)

- List of alternatives; e.g.:

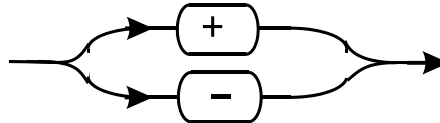
statement-list:

statement

statement statement-list

- Diagram;

e.g., *sign:*



Syntax rules and diagrams

compound-statement:

{ statement-list }

**Diagram for
*statement-list***



statement-list:

statement

statement statement-list

statement:

nothing

declaration

assignment

IO-statement

compound-statement

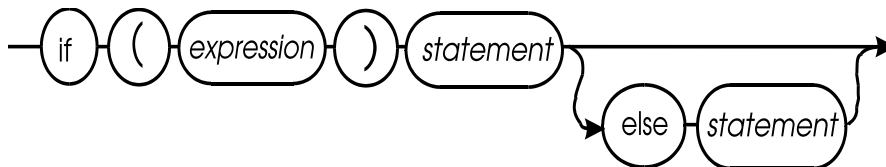
Java syntax for *if*

branch-statement :

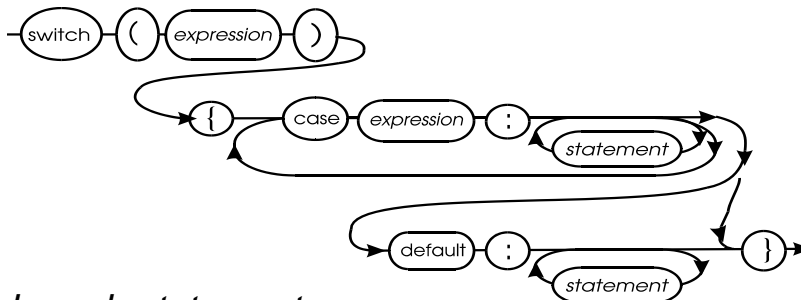
if (*expr*) *statement*

if (*expr*) *statement* **else** *statement*

- *Statement* may be assignment, compound statement, *if* statement, loop statement, method call, etc.



Syntax for *switch*



branch-statement :

switch (*expr*) *compound-statement*

The subordinate *statement* in a *switch* is normally compound, with case labels, alternative statements, *breaks*

Loop syntax

loop-statement:

while (*expression*) *statement*

do *statement* **while** (*expression*) ;

for (*expr* ; *expr* ; *expr*) *statement*

Loop semantics

- The *expression* in the *while* and *do...while* statements is the exit test.
- The *expressions* in the *for* statement are for initialization, exit test, and updating.

Lexical analysis in programming-language translation

- Compiler translates from higher-level language to assembler or machine code
- Lexical analysis
 - Finds *tokens*, indivisible items of code
 - Tokens are formed by simple rules
 - Examples: literals, operators, keywords, delimiters, identifiers
 - lexer stores tokens in sequence
- Parsing applies grammar rules to build tokens into a structure

Kinds of tokens (lexical elements)

- keyword (*void, main, int, ...*)
- identifier (letter or ‘_’ followed by a series of letters, digits, ‘_’s)
- constant literal (numeral, double-quoted string, single quoted character)
- operator (=, +, *, -)
- punctuator (semicolon, comma, paren, brace)

Not tokens:

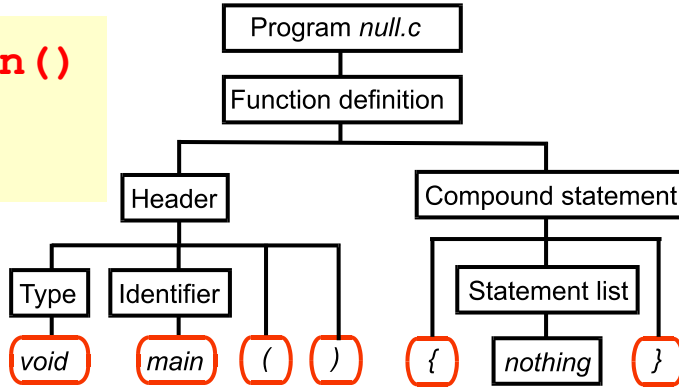
- The compiler ignores white space (space characters, tabs, newlines)
- Compiler ignores comments (*//..., /*...*/*)

Context-free grammars

- Defined by *productions* using \rightarrow to denote “is defined as”
- Symbols:
 - *terminal*: self-defining or defined by a lexical grammar;
 - *non-terminal*: defined by a production
- *Examples*:
 - $method-defn \rightarrow type\ ID\ (\)\ comp-stmt$
 - $comp-stmt \rightarrow \{ stmt-list \}$
 - $stmt-list \rightarrow \lambda \mid stmt\ stmt-list$

The parser generates a parse tree of a program

```
void main()
{
}
```



Each syntax rule is applied by putting a defined element's components under the name of the element

2. Standard data types

0.2a Describe standard Java types and classes*

0.2b Correct a type error*

0.2c Use logical operators*

0.2d Use bitwise operators

Classes and methods

- A *class* is an abstract specification for *objects*, which are *instances* of the class; i.e., *data items*
- *Example*: the informal concept, *students*, is a *class*, while a particular student is an *object*
- Classes have *members* that are *methods* (operations) or *properties* (data items, instance variables)
- *Methods* are invoked by writing an object's or class's name, a dot, and the method name, followed by parameter(s) in parentheses

A Java program defines a class

- Program must define one *public class* and may define any number of private classes
- This public class defines the *application*
- Each class may define one or more *method* (subprogram) and *attribute* (data item)
- A public class must define a method called *main*, which executes automatically
- A method definition contains *executable statements*

Numeric data types

- *Definition*: a specification that defines the *storage* and *meaning* of patterns of bits
- Java stores two kinds of numeric values: *integer* and *floating-point*
- *int* is a signed 32-bit integer type, with range of values $-2G \dots +2G$ (32 bits: $2^{32} = 4G$)
- After *int* $x = 9$;, x is an expression of type *int*
- Other integer types: *byte* (8 bits), *short* (16 bits), *long* (64 bits)
- Floating-point (fractional) types: *float*, *double*

Standard numeric types

Type	Storage (bits)	Range
Integer		
<i>byte</i>	8	$-128 \dots 127$
<i>short</i>	16	$-32,768 \dots 32,767$
<i>int</i>	32	$-2.1 \times 10^9 \dots 2.1 \times 10^9$
<i>long</i>	64	$-2^{63} \dots 2^{63}$
Floating-point		
<i>float</i>	32	$-3.4 \times 10^{38} \dots 3.4 \times 10^{38}$
<i>double</i>	64	$-1.8 \times 10^{308} \dots 1.8 \times 10^{308}$

Operator precedence

- Parenthesized operations come first
- Unary minus has high precedence
- Multiplication and division precede addition and subtraction
- Operations of the same precedence proceed from left to right

- *Examples:*

```
8 - 2 + 5    8 - (2 + 5)
3 * 2 + 4    3 * (2 + 4)  3 + 2 * 4
3 + 6 / 2    (3 + 6) / 2  -2 + 3
1 + 3 % 2    25 % 5 * 2
```

Overflow

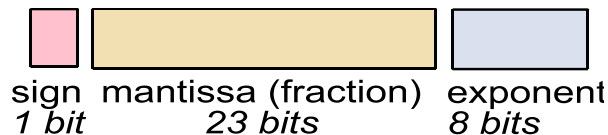
- When a value is assigned that exceeds the capacity of a variable, *overflow* occurs and incorrect value is stored
- In overflow, the number of bits in the value is greater than the capacity of storage of the variable
- *Example:*

```
byte a = 100, b = 180;  
System.out.println(a+b);
```

Output: ____

Data types *float* and *double*

- May store fractional values of great range; *float* 32 bits, *double* 64 bits
- *Storage*, based on scientific-notation concept: *sign bit*, *fraction*, *exponent*
- The numeral “floats” left or right because the exponent part compensates for a shift to eliminate 0’s on left



A Boolean variable (flag) stores a truth value

may be replaced by int *a flag*

- `boolean invalid = (age < 0);`
 ...
`if (invalid)`
`out.println("Invalid age");`
- *boolean* is a standard Java data type with a range of values {*false*, *true*} (0, 1)
- Boolean variables hold values for later use

Character-based types

- *char* is a standard basic type
- *String* objects are sequences of characters in memory
- *String* is a *class* defined by the standard library package *String*.
- *Streams* are sequences of characters going to output devices or coming from input devices
- *Files* are stream objects

The standard data type *char*

- *char* data items are 16 bits, representing up to 65,536 symbols in Unicode
- To declare a variable that stores *one* character:
char c;
- The *ASCII table* maps a character set to integers 0...127 (7 bits)
- *Character literals* use single quotes
- *Escape sequences* express special characters:
newline `'\n'`, tab `'\t'`, backspace `'\b'`, single quote `'\''`, null character `'\0'`, backslash `'\\'`

Java strings

- *String* class declares string objects
- String literals use double quotes:

```
String stu_id = "ab12";
String greeting = "Hi"+stu_id;
```
- Concatenation operator '+' converts any type to *String*, allowing grouping of output items of different types in *println* calls
- Methods of *String* class: *length()*, *parseInt()*, *substring()*, *charAt()*

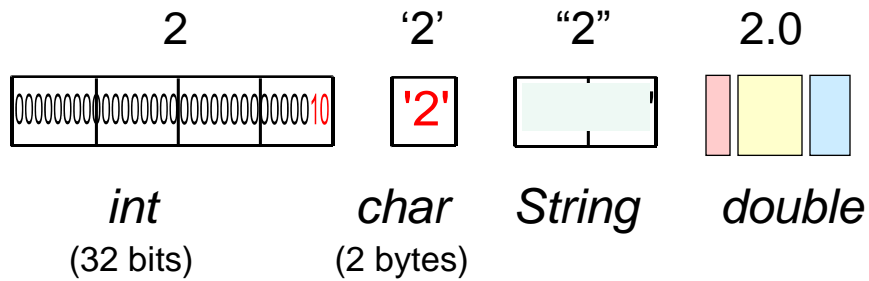
String methods

Expression	Value
<code>String s1 = toupper("abcd")</code>	ABCD
<code>char c = s1.charAt(2);</code>	C
<code>String s2 = s1.substring(2,2)</code>	CD
<code>int i = s1.indexOf('B')</code>	1

Other methods:

<code>length()</code>	<code>compareTo(t)</code>	<code>equals(t)</code>
<code>startsWith(t)</code>	<code>startsWith(t, i)</code>	<code>endsWith(t)</code>
<code>contains(cs)</code>	<code>indexOf(t)</code>	<code>trim()</code>
<code>indexOf(c)</code>	<code>indexOf(c, i)</code>	<code>indexOf(t, i)</code>
<code>s.substring(i)</code>	<code>substring(i, j)</code>	<code>charAt(i),</code>
<code>toUpperCase()</code>	<code>replace(c1, c2)</code>	<code>toLowerCase()</code>
<code>compareToIgnoreCase(t)</code>	<code>equalsIgnoreCase(t)</code>	<code>replace(cs2, cs3)</code>

Storage for four standard types



Stream input/output

- A *stream* is a sequence of characters moving from or to a device



- Stored data occupies finite space; streams are infinite
- Destinations/sources for streams:
 - keyboard
 - screen
 - ports
 - disk files

Formatting output

- *System.out.printf()* enables specification of width, precision, and type of output elements
- *Example:*
`System.out.printf("%-6s%5.2f%n", "Tax:", tot);`
displays a *String* and a *float* value, with the string left-aligned across width of six, and the *float* of width five and precision to two decimal places
- *%n* denotes end-of-line character
- Other type specifiers: *%d* (decimal), *%x* (hexadecimal), *%o* (octal), *%e* (exponential),

Output objects and method calls

- Methods (subprograms) *print* and *println* cause output of their parameter values
`System.out.print("Sum");`
`System.out.print("Sum="+y);`
- *System* is a predefined class; *out* is an output stream object, a member of *System*
- *println* is a method of the class of *out*
- “Sum” is a parameter to *println*
- Values of different types may be concatenated with “+” operator

Formatted strings

- *String.format(fmt_spec, parm_list)*
returns a string formatted using
 - %s string
 - %n end of line
 - %8d decimal integer right-flush
across 8 columns
 - %8.2f floating-point with two
digits precision
 - %-8s left-flush across 8 columns

Classes and methods for input

- Standard Java class: *Scanner*
- Input requires creation of a *Scanner* object and call to a *next...* method

```
Scanner in =
    new Scanner(System.in);
x = in.nextInt();
```
- The method call, *in.nextInt()*, fetches the next space-delimited sequence of characters from the stream *in* and returns it as an integer value

Inputting numbers and strings

```
double d = in.nextDouble();  
String s = in.next();  
    // reads until white-space char  
String s = in.nextLine();  
    // reads until end of line  
Char x = in.nextDouble() is a type error
```

The code above assumes that *in* has been initialized as a *Scanner* object

Java packages

- I/O and many other features are not part of the Java language
- They are made part of a Java program by using code *packages*
- The packages are *imported* to the *.java* file using *import* at the top of the file:

```
import java.awt;
```
- The standard package, which defines *System*, is imported *implicitly*

Some standard Java packages

- *java.lang* (automatically imported classes)
 - *System*
 - *String*
- *java.awt*: Abstract Windowing Toolkit, graphics classes
- *java.applet*: classes for Web applets

The standard Java library has thousands of classes. The Application Programming Interface documentation (java.sun.com/javase/6/docs/app) explains each class and how to use it in Java programs

Relational operators

equal-to	==	not-equal-to	!=
greater	>	less-than-or-equal	<=
less-than	<	greater-or-equal	>=

- Expressions with relational operators have *Boolean* values
- Each operator has a complement
- *Tip*: don't compare *doubles* or *Strings* for equality

Logical operators

Operation	Eng.	Logic	Java	Example
Negation	not	\neg	!	! (price > cost)
Conjunction	and	\wedge	&&	a > b && b > c
Disjunction	or	\vee		x == 1 x == 2

- Nested *ifs* may express conjunction too:

```
if (age > 0)
    if (age < 120)
        out.print("Valid age");
```

- The above is equivalent to

```
if (age > 0 && age < 120)
    out.print("Valid age");
```

Operator precedence in boolean expressions

- Parenthesized operations come first
- Logical negation (\sim) has high precedence
- Arithmetic operators precede relational operators
- Relational operators ($==$, $!=$, $<$, $<=$, $>$, $?=$) precede binary logical operators ($||$, $&&$)
- $&&$ (AND) precedes $||$ (OR)

Using logical operators

```
out.print("Enter 3 integers: ");
int a = in.nextInt(), |
    b = in.nextInt(),
    c = in.nextInt();
if (a == b && b == c)
    out.print("They're the same");
out.print("Enter your age: ");
int age;
age = in.nextInt(); Boolean variable
boolean impossible =
    (age < 0 || age > 120);
if (! impossible)
    out.print("Thank you");
```

[logops.cpp]

Bitwise operations

- Operations: AND (&), OR (|), XOR (^), complement (~), shift (<<, >>)
- Each operator performs a logical operation on each bit of operand(s)
- *Applications:*
 - Compact storage of status data
 - Storage of sets
 - Arithmetic at hardware level
 - Systems programming

Shift-left operator multiplies by a power of 2

`input << 3`

*Shift-left-3
multiplies by 8
because $2^3 = 8$*

Bitwise examples

Java

<i>Operation</i>	<i>operator</i>	<i>Example</i>
Complement	<code>~</code>	$\sim 10000000_2 = 01111111_2$
OR	<code> </code>	$1100_2 1001_2 = 1101_2$
AND	<code>&</code>	$1100_2 \& 1001_2 = 1000_2$
Left shift	<code><<</code>	$1101_2 \ll 1 = 11010_2$
Right shift	<code>>></code>	$11000_2 \gg 2 = 110_2$
XOR	<code>^</code>	$1001_2 \wedge 1010_2 = 0011_2$

Setting a bit to 1 with OR

- To set n th bit of a number, create a mask by shifting a 1 left $(n - 1)$ times, and then bitwise-OR the number with the mask.
- E.g., **0001_2 OR $1010_2 = 1011_2$**
- So to set 1st (rightmost) bit of 4-bit value ten (1010_2), OR it with 0001:

10 | 1

To set the n th bit of a number

```
int n = in.next(),
    x = in.next();
int mask = 1 << (n - 1);
int result = x | mask;
```

The OR operation makes sure that the mask value's single 1 bit sets to 1 the corresponding bit in the result

Clearing a bit to 0 with AND

- To clear n th bit of a number, create a mask by shifting a 1 left $(n - 1)$ times and complement this; then bitwise-AND the number with the mask
- E.g., (1) $\sim 0001_2 = 1110_2$
(2) $1110_2 \text{ AND } 1011_2 = 1010_2$
(First bit in 1011 is cleared in 1010)
- To clear first (from right) bit of 4-bit value 11_{10} (1011_2), AND it with complement of 1 shifted left 0 places:
`cout << (11 & ~(1 << 0));`

To clear the n th bit

```
int n, x;
cin >> n >> x;
int mask = ~(1 << (n - 1));
int result = x & mask;
```

The AND operation makes sure that the mask value's single 0 bit clears to 0 the corresponding bit in the result

3. Loops and debugging

0.3a Trace a branch or loop**

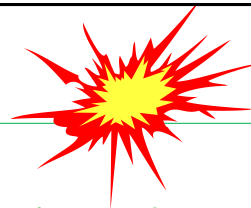
0.3b Solve a numeric loop problem**

0.3c Solve a loop problem
with strings**†

0.3d Debug a defective loop**†

Algorithm:

*A precise plan to transform
input to output in a finite number of steps*



- Program designs use algorithms
- Most computation is algorithmic
- Flowcharts and pseudocode can represent algorithms



Debugging a flawed design

- Suppose we try to find the largest of three numbers as follows:

input a, b, c

$y \leftarrow a$

if $b > a$

$y \leftarrow b$

if $c > a$

$y \leftarrow c$

a	b	c	y
2	4	3	2
			4
			3

- Trace of this algorithm for $(a, b, c) = (2, 4, 3)$ is above; do you see the error?

Tracing an algorithm or process

- Allows designer to check result of algorithm, including internal (undisplayed) values
- Use one column per variable; one row per iteration.
- Example* (See prev slide), assuming input 3, 2, 1, 0:

<i>quantity</i>	<i>total</i>	<i>output</i>
	0	
3	3	
2	5	
1	6	
0	6	6

Why trace?

- Computer programs don't display all their internal workings
- To find and fix a car problem, the mechanic must look under the hood
- A trace displays the values of all variables as they change
- Tracing is crucial in debugging programs and systems

Loops and strings

- String manipulation is a major application for loops
- The parameter of *charAt* is called an *index* or *subscript* to the string

```
// Finds first character after
// first space in 'name'
String name = in.nextLine();
for (int i=0; i < name.length; i++)
    if (name.charAt(i) == ' ')
        out.println(name.charAt(i+1));
```

Extracting substrings from strings

```
String s = in.nextLine();
int num_spcs = 0, spc_1_loc = 0,
    spc_2_loc = 0;
for (int i=1; i < s.length(); i++)
    if (s.charAt(i) == ' ')
    {
        num_spcs++;
        if (num_spcs == 2)
            spc_2_loc = i;
        if (num_spcs == 3)
            spc_3_loc = i;
    }
int word_len = spc_3_loc - spc_2_loc;
String third_word =
    s.substring(spc_2_loc, word_len);
```

*Example: read
a line of text
and extract the
third word*

Uncontrolled infinite loops

- The hardest-to-find infinite loop is one that *may* exit sometimes:

```
int power;
while (power < 1000)
{
    out.print(power + " ");
    power = power * 2;
}
```

**HANGS
sometimes**

- To terminate, a loop must change a value that is tested in the exit condition

```
int count = 0;
while (count < 100)
    out.print(count + " ");
    ++count;
```

**HANGS
always**

Boolean variables for loop control

```
final int sentinel = 0;
boolean done = false;
int total = 0;
while (! done)
{
    // <total> stores sum of input so far
    int input;
    out.print("Enter a number, 0 to exit: ");
    input = in.nextInt();
    if (input != sentinel)
        total += input;
    else
        done = true;
}
out.print(total);
```

• Boolean loop-control variable (flag)

Convergence of exit-test value assures termination

```
Input  $n$ 
 $count \leftarrow 0$ 
while  $n > 0$ 
     $n \leftarrow \lfloor n / 2 \rfloor$ 
     $count \leftarrow count + 1$ 
```

- The value n converges on 0
- This guarantees that the loop will exit

Kinds of errors

- *Syntax*: violation of grammar rule; caught by compiler
- *Specification*: Program solves the wrong problem
- *Logic*: Programmer's chief concern; program produces incorrect or unpredictable output
- *Runtime*: Cause abnormal termination due to invalid operations, illegal memory access, etc. Preventable.

Errors to watch for

- Uninitialized variables
- Unused variables or parameters
- Poorly indented code
- Variables, functions, or types with vague names (“value”, “process”, “data”...)
- Loops not *provably* terminating
- Loops that iterate once too many or too few times
- Array subscripts not *provably* in bounds

Common pitfalls with loops

- Declaring inside a loop a value updated by the loop
- Iterating one too few times
- Iterating one too many times
- Impossible exit conditions
 - value tested not changed in loop body
 - value changed may fail to move toward exit value
- Exit condition that is never met

Tracing a loop

- When a loop produces bad results, *tracing hidden values* helps in debugging
- **Trace statement** below shows garbage values

```
int count, x, total;
x = in.nextInt();
while (input > 0)
{
    out.print("x=" + x + " total=" + total);
    x = in.nextInt();
    total += x;
}
[trace.java]
```


Testing and correctness

- *Testing* can only prove the existence of faults, not their absence
- *Correcting a fault* late in development is expensive
- *Regression testing* finds errors introduced by maintenance process
- *Integration testing* determines whether separately developed modules work together
- *Engineering* applies teamwork, standards, science, and math to problem solving

4. Classes and objects

0.4a Write a UML class diagram*

0.4b Describe memory allocation for objects**

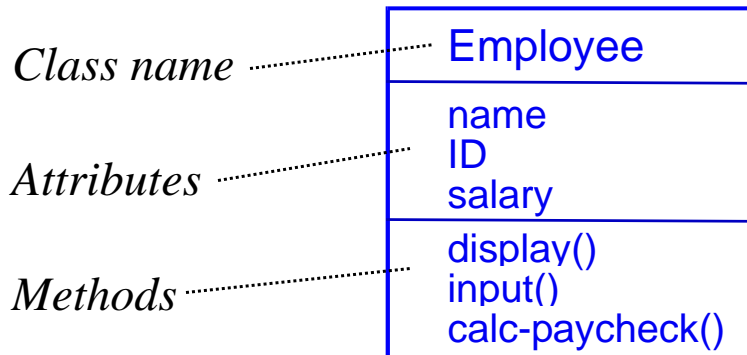
Data abstraction

- *Defn*: the creation of new data types
- Some data types, whose instances have components, are called *compound*
- *Objects* are defined by their *attributes* and *operations*
- Objects are *instances of classes*
- Encapsulation separates classes' *interfaces* from their *implementation*

Object-oriented design

- Any concept is a candidate for a class: persons, things, places, transactions
- Relationships among classes include
 - *containment* (an address object is part of a customer object)
 - *inheritance* (scrollers and dialogs are two kinds of views)
- A class implements an abstraction; it may be instantiated by one or more objects

UML class diagrams



- Classes model *state* and *behavior*

Why program with classes?

- We may model the *state* and *behavior* of what we want to represent: e.g., persons, events, collections, displayed objects
- Classes let us model the *interactions* found in the environments we work with
- Class libraries may be *reused* conveniently
- Bookseller examples: *books*, *customers*, *transactions* (*challenge*: define classes for them)

Defining data types

- Enumerated types are a simple example
- Keyword *enum* enables definition of a named integer type whose possible values are specified by constant identifiers

- *Example:*

```
public enum Seasons
    {Spring, Summer, Fall, Winter};
Seasons this_season =
    Seasons.Winter;
```

A class associates attributes

- *Class design guidelines:*
 - Use data members that are *attributes* of an object whose values may persist
 - Use *local variables* for values that are only of use during execution of a method
- *Example:* a value like *paycheckAmount* that is computed from *wageRate* and *hours*, should be a local variable

Methods and objects

- A call to a method names an instance of the class, using a dot

```
Employee emp = new Employee();  
emp.display();
```

- Methods of a class have access to the members of that class's instances

```
public void display()  
{  
    out.print(name + hours);  
}
```

- A method that returns an object name returns a *reference* to the object, not a *copy*

Objects vs. object references

- When a variable is declared with a class as its type, it refers to an *object* or is a *reference*

- If *Purchase* is a class, then

```
Purchase p = new Purchase();
```

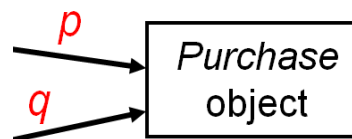
declares an *object*

- If *p* is a *Purchase*, then

```
Purchase q = p;
```

declares a *reference* to *p*

- References store *addresses* of objects, not *copies*



Null objects

- For any class C , the following is acceptable:

```
C x = null;
```

- In that case, x is a null object reference, denoting no object
- Later the variable may be given a non-null value:

```
x = new C();
```

- Object references may be tested for nullness:

```
if (x == null)...
```

5. Precalculus concepts

0.5 Explain precalculus concepts*

Functions

- *Function*: a relation of which each left-hand member of a tuple is in not more than one tuple (maps to a unique value)
- *Examples*: *EVEN* is a function;
> is a relation but not a function
- Every function has a
 - *Domain*: set of values mapped from
 - *Range*: set of values mapped to
- Computation of a function takes parameter as input, return value as output

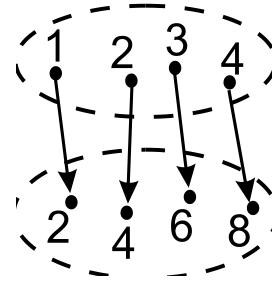
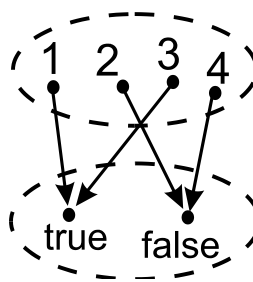
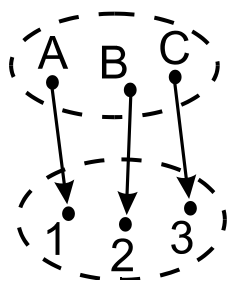
Mathematical functions

- *Function*: A set that is a mapping from one set to itself or to another set
- *Examples*:

$Index('B') = 2$

$Odd(3) = true$

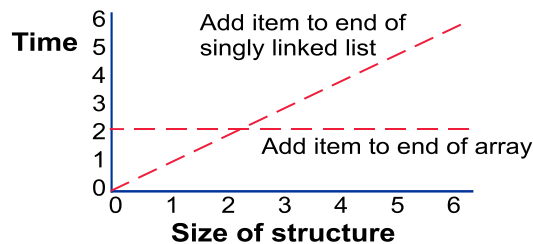
$Twice(1) = 2$



Examples of functions

- *Identity* (for a set): $(\forall x \in A) I_A(x) = x$
- *Sequence* as a function: A sequence A may be defined as a function f_A where $f_A(x) = A_x$
- *Arithmetic operators*: the operators $+$, $-$, \times , \div , may be defined as functions $f: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ where, for example, $+(a, b) = a + b$
- *Other functions*: predicates (Boolean functions); encoding and decoding functions; logarithm; factorial; exponent

Linear functions



- A linear function may be expressed $y = ax + b$
- Its graph is a line; e.g., the two red broken lines above

Polynomial functions

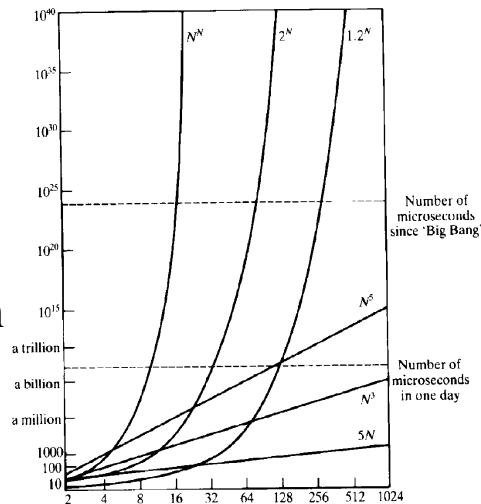
- A quadratic function (polynomial of degree 2) may be expressed

$$y = ax^2 + bx + c$$
- Its graph is a parabola
- A polynomial function is of degree 1 or higher: $y = ax^3 + bx^2 + cx + d$

Exponential functions

- May be expressed

$$y = a2^x + f(x),$$
 where f is a polynomial
- The value of any exponential function grows faster than any polynomial function



Polynomials and power functions

- A *power function with exponent a* is
 $p_a : \mathbf{R}^+ \rightarrow \mathbf{R}^+$ s.t. $y = p_a(x) = x^a$
- *Examples:* $f(x) = 1$ $f(x) = x^{0.5} = \sqrt{x}$
 $f(x) = x$ $f(x) = x^2$ $f(x) = x^{10}$
- *Polynomials* extend the power functions by including sums of power functions
- Polynomials may be defined on sets of natural numbers, in which case they are discontinuous (have breaks in their graphs)

Logarithmic functions

- Function $\log_b(x)$ is the inverse of function b^x
- Logarithmic functions grow extremely slowly, more slowly than $y = \text{sqrt}(x)$
- Processes of decay are described by logarithmic functions

Exponential and logarithmic functions

- Many processes of growth and decay are described by exponential and logarithmic functions
- Function $\log_b(x)$ is the inverse of function b^x
- These functions grow extremely slowly and extremely quickly, respectively
- These functions grow proportional to the base; i.e., the big-O analysis is independent of base

The factorial function

- **Factorial of n ($n!$):**
 $n \times (n - 1) \times (n - 2) \times \dots \times 1$
- ***factorial* (n) =**
$$\begin{cases} 1 & \text{if } n \leq 1 \\ n \times \text{factorial}(n - 1) & \text{otherwise} \end{cases}$$
- ($n!$) expresses the number of permutations of a set of size n

Sequences

- A *sequence over A* is a function $s: \mathbb{N} \rightarrow A$, written s_1, s_2, s_3, \dots
- Each element of s is called a *term* and the subscripts are called *indexes*
- *General formula*: a rule that shows how for a sequence s , values of s_k depend on k
- *Arithmetic sequences*' terms differ by a constant amount
- *Geometric sequences*' terms differ by a constant factor

Strings

- *Alphabet*: a finite set of symbols, e.g., $\{0,1\}$, $\{0, 1, \dots, 9\}$, $\{a, b, c, \dots, z\}$
- A *string* is a sequence of symbols over a finite alphabet; Σ^* is the set of strings over Σ
Recursive definition:
 - *Base*: $\lambda \in \Sigma^*$ ($\lambda = \varepsilon = \text{null string}$)
 - *Recursive*: $a \in \Sigma \wedge s \in \Sigma^* \Rightarrow sa \in \Sigma^*$
 - *Restriction*: *only* objects defined as above are strings
- In set notation, $\Sigma^* = \{\lambda\} \cup \{ax \mid a \in \Sigma, x \in \Sigma^*\}$

The binary system

- Appropriate for two-state devices
- *Binary* is the form in which all information is represented (numeric, text, graphical, sound)
- Uses two digits (1 and 0) rather than ten
- Like decimal, uses *place values*
- We distinguish *numerals* (representations) from *numbers* (abstractions)

Relations

- *Definition:* a subset of a Cartesian product of sets
- A relation R is thus a set of ordered pairs, e.g., $\{ (2,1), (3,1), (3,2), (4,1), (4,2), (4,3) \}$
- If R is a relation and $(x, y) \in R$, we write $x R y$
- *Example:* $>$ is a relation, because $2 > 1$; $3 > 1$, etc.

$>$	1	2	3	4
1	F	F	F	F
2	T	F	F	F
3	T	T	F	F
4	T	T	T	F

Relations and databases

- A database table is a relation
- *Examples:*
 - (names \times IDs \times addresses)
 - The Cartesian product of *students* and *courses* is the set of all possible pairings of *course*, *student*
 - A table of student registrations is a relation $R \subseteq (\textit{students} \times \textit{courses})$

References

- Cay Horstmann. *Big Java*, 3rd ed. Wiley, 2008.
- D. Keil. Java file input/output. Classroom handout, 2008.
- D. Keil. Reading and displaying file records. Classroom handout, 2008.