CS 240 – Computer Science I

Acknowledgements: These slides were created by Dr. Travis Doom with information, graphics, materials, or aid kindly provided by Dr. Matt Rizki, Gaddis’s “Starting Out with Java” (Addison Wesley), Patt’s “Introduction to computers” and McConnell’s “Code Complete”.
An engineer’s introduction to abstraction and the digital computer

The modern general purpose digital computer

Abstraction/Encapsulation

Design decomposition
What is a computer?

• What is computation?

• There are many sorts of computing devices, they fall into two categories:
  – Analog: machines that produce an answer that measures some continuous physical property such as distance, light intensity, or voltage. Examples?
  – Digital: machines that perform computations by manipulating a fixed finite set of elements. Examples?
  – The difficulty with analog devices is that it is very hard to increase their accuracy.

• Before modern digital computers, the most common digital machines were adding machines.
  – Adding machines perform exactly one sort of operation.

• General purpose digital computers also perform one operation…
  – Modern computers accept a set of instructions that tell it how to do any sort of computation.

• We now make a clear distinction between “calculators” and “computers”
How *do* we get the electrons to do the work?

- We describe our problems in English or some other natural language. Computer problems are solved by electrons flowing around inside the computer.
- It is necessary to transform our problem from a natural language to the voltages that influence the flow of electrons.
- This transformation is really a sequence of systematic transformations, developed and improved over the last 50 years, which combine to give the computer the ability to carry out what may *appear* to be very complicated tasks. In reality, these tasks must decomposed into a number of simple and straightforward subtasks.

- Engineering design
  - Top down design
  - Decomposition into smaller problems (Divide and Conquer!)
  - Levels of Abstraction
The principle of design abstraction

- General model for Engineering (Byrne, 1992)
Levels of abstraction in digital computation

The Problem

Software level
Algorithm & Language

Hardware level
ISA & Microarchitecture

Logic level
Circuits & Devices

Computer Science
CS 24x, 4xx

Computer Engineering
CEG 320

Computer/Elect. Engineering
CEG 260/360
The statement of the problem

- We describe problems that we wish to solve with a computer in a “natural language.”
- Natural languages are fraught with a lot of things unacceptable for providing instructions to a computer.
- The most important of these unacceptable attributes is ambiguity. To infer the meaning of a sentence, a listener is often helped by context that the computer does not have.
  - “Mary had a little lamb and other nursery rhymes.”
- A computer cannot deal with any ambiguity, thus…
The algorithm

The first step in the sequence of transformations is to transform the natural language description of the problem to an algorithm.

An algorithm is a step-by-step procedure:
- That transforms an input (possibly NULL) into some output (or output action)
- That is guaranteed to terminate

**Definiteness**: Each step is precisely stated.

**Effective computability**: Each step must be something the computer can perform

**Finiteness**: The procedure must terminate/repeat

For any computable problem, there are an infinite number of algorithms to solve it.
- Which solution is best?
Exponential growth

- $10^1$
- $10^2$
- $10^3$ Number of students in the college of engineering
- $10^4$ Number of students enrolled at Wright State University
- $10^6$ Number of people in Dayton
- $10^8$ Number of people in Ohio
- $10^{10}$ Number of stars in the galaxy
- $10^{20}$ Total number of all stars in the universe
- $10^{80}$ Total number of particles in the universe
- $10^{100} \ll$ Number of possible solutions to traveling salesman (100)

- Travelling salesman (100) is *computable* but it is NOT feasible.
The programming language

The next step is to transform the algorithm into a computer program.

Programming languages are unambiguous “mechanical” languages.

There are two kinds of programming languages:

- High-level languages are machine independent. They are “far above” the (underlying) computer.
- Low-level languages are machine dependent. They are tied to the computer on which the program will execute. There is generally only one such language per machine (referred to as its ASSEMBLY language).
How do we specify the program?

- Contemporary languages
  - Java, C++, C, C#, Perl, Python, Ruby, and many more.
- Languages of yore
  - Fortran, COBOL, and many more.
- Specialty languages
  - VHDL, simulation languages, and many more.

- There are over 1,000 “standardized” programming languages today.

- The only goal of these languages is to help humans implement their algorithms in the instructions available for a particular ISA
  - This is what we mean when we say “programming”
Universal computing devices

- **Turing’s Thesis**: Computer scientists believe that ANYTHING that can be computed by a general purpose computer can be computed by any general purpose computer (provided that it has enough time and enough memory).

- What does this imply?
  - All computers (from the least expensive to the most expensive) are capable of computing EXACTLY the same things IF they are given enough time and enough memory.
  - Some computers can do things faster, but none can do more than any other computer.
  - All computers can do exactly the same things!

- Thus, any given problem is either computable or it is not computable
  - Problems may be computable, but still not feasible (NPC)
The instruction set architecture (ISA)

- The next step is to translate the program into the instruction set of the particular computer that will be used to carry out the work of the program.
- The Instruction Set Architecture (ISA) is the complete specification of the interface between programs that have been written and the underlying hardware that must carry out the work of those programs.
  - Examples: IA-32 (Intel, AMD, and others), PowerPC (Motorola)
- Programs are translated from high languages into the ISA of the computer on which they will be run by a program called a compiler (specific to the ISA).
  - Some languages are interpreted as they execute.
- Programs are translated from assembly to the ISA by an assembler.
Compilation vs. Interpretation

- Different ways of translating high-level language
  - **Interpretation**
    - interpreter = program that executes program statements
    - generally one line/command at a time
    - limited processing
    - easy to debug, make changes, view intermediate results
    - languages: Java, BASIC, LISP, Perl, Matlab, C-shell
  - **Compilation**
    - translates statements into machine language
      - does not execute, but creates executable program
    - performs optimization over multiple statements
    - change requires recompilation
      - can be harder to debug, since executed code may be different
    - languages: C++, C, Fortran, Pascal, etc.

Get W from the keyboard.

\[
X = W + W \\
Y = X + X \\
Z = Y + Y
\]

Print Z to screen.

How many arithmetic operations when interpreted? When compiled with optimization?
The von Neumann Model - Illustration

- **Memory**: holds both data and instructions
- **Processing Unit**: carries out the instructions
- **Control Unit**: sequences and interprets instructions (Fetch, Decode, Execute)
- **Input**: external information into the memory (keyboard, mouse, disk, NIC, etc.)
- **Output**: produces results for the user (monitor, printer, disk, NIC, etc.)
The instruction set

- **High-Level Language – Java/C++**
  - A = B + C;
  - Memory-Transfer Equivalent

- **Machine-Level Equivalent**
  - Assembly (human readable) ex: Machine (for a simple architecture)
    - Load R2, B
      - xE2EA08 1110 0010 1110 1010 0000 1000
    - Load R3, C
      - xE3EA10 1110 0011 1110 1010 0001 0000
    - R2 ← R2 + R3
      - x0223 0000 0010 0010 0011
    - Store A, R2
      - xF2EA00 1111 0010 1110 1010 0000 0000

- **Bits – Binary digits**
  - Two values (zero or one)
  - We can represent this as current flowing or not flowing
The microarchitecture

- The next step is to transform the ISA into an implementation. The detailed organization of an implementation is called its microarchitecture.
  - The IA-32 has been implemented by several different processors over the past twenty years: 8086 (Intel, 1979), 8286, 8386, 8486, Pentium (many generations), Athlon (many generations), etc.
  - Each implementation is an opportunity for computer designers to make different trade-offs between cost and performance. [Computer design is always an exercise in trade-offs.]

- Some ISA are implemented by a “virtual” microarchitecture which itself is implemented by an actual microarchitecture
  - Emulators
  - Java Virtual Machine
The logic circuit

- The next step is to implement each element of the microprocessor out of simple logic devices.
- Here there are also choices, as the logic designer decides how to best make the trade-offs between cost and performance.
- Even in the case of addition, there are several choices of logic circuits to perform this operation and differing speeds and corresponding costs.
The devices

- Finally, each basic logic circuit is implemented in accordance with the requirements of the particular device technology used.
- So, CMOS circuits are different from NMOS circuits, which are different, in turn, from gallium arsenide circuits.
Programming basics

Semantics, syntax, and style
Declarations, variables, and data types
Assignments
Operators
Parts of a program

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

- **Semantics** – the “meaning”
  - When programming we should focus on semantics.
- **Syntax** – the rules that must be followed when writing a program
  - In algebra, what is “5 + 3”? What is “5-”? 
  - Programming language syntax rules are like grammar rules in natural/spoken languages. Syntax differs by language! 
  - Syntax rules are required to avoid ambiguity. 
  - Compilers/IDEs help identify syntax errors but, like grammar, they must largely be memorized 
- **Style** – conventions that affect the readability of the program
The class header

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

- Semantics - Classes are “containers” that group together separate portions of a design that conceptually “belong” together.

- Syntax – Every java file must have exactly one “public” class. The name of that class is also the name of the text file in which the program is stored (e.g. HelloWorld.java)
  - Case sensitive keywords (public, class) and identifiers (HelloWorld)
  - Identifiers can use alpha, numeric (non-leading), and underscore
The main method header

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

- Semantically – Methods (aka functions, routines, subroutines, procedures) contain code to complete a task.
  - The main routine defines that “start” of the algorithm
- Syntax – Every java program must have exactly one “main” method. The main method’s is the start (and end) point of the program’s execution.
The main method body

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

- **Semantically** – Methods bodies consists of statements that describe an algorithm to be executed
- **Syntax** – Statements identify a specific instruction (using a key word) or a programmer/system defined method (with an identifier)
  - Statements must end with a semicolon
  - Whitespace (including indentation) is largely ignored
  - `System.out.println()` is a method to produce output. The details are hidden in the implementation of the System class.
How Java Compiles

```java
package HelloWorld;          // project/application name
public class HelloWorld {     // class/file name
    public static void main (String[] args) {
        System.out.println("Hello World");
    }
}
```

Java source code
Hello.java (Text file)

```
javac
```

Java byte code
Hello.class file (Java Bytecode)

```
java (JVM)
```

JVM is a Just In Time compiler!

Runs under XP on a PC
Runs under Linux on a PC
Runs under Unix on a mainframe
```java
package HelloWorld; // project/application name
public class HelloWorld { // class/file name
    public static void main (String[] args) {
        System.out.println("Hello World");
    }
}
```

```java
package HelloWorld; // HelloWorld = 7;
public class HelloWorld { public static void main (String[] args) { System.out.println ("Hello World"); }
}
```
Variable declarations

- Variables are used to hold data items.
- Each variable has a data type which tells the compiler what sort of data is to be stored (and how much space it needs, etc.).
- Variable Declarations take the form: `dataType variableName;`
  ```java
  int counter;
  ```
- Primitive data types are built into a programming language and have reserved/key words. Other (derived) data types are created by programmers and use identifiers.
- Java has 8 primitive data types:
  - byte
  - short
  - int
  - long
  - float
  - double
  - boolean
  - char
- Local variables have a lifetime (scope) (for now assume to method end)
Assignment statements

- Assignment statements take the following form:
  \[
  \text{variableName} = \text{value}; \quad // \text{read variable “gets” value}
  \]

- The Left Hand Side (LHS) of the assignment must be a variable
- The Right Hand Side (RHS) can be a literal, a variable, a method, an equation, or anything that can be interpreted as a value!
- Literals represent an unchanging (non variable or constant) value.
- Each literal has a \textit{data type} which can be specified or will be implied by the compiler.

  - \texttt{value} = 5; \quad // \text{Integer numbers default to type int}
  - \texttt{price} = 5.50; \quad // \text{Floating point numbers default to double}
  - \texttt{letter} = ‘a’; \quad // \text{Single character literals (use ‘’)}
  - \texttt{word} = “Dogma”; \quad // \text{Strings are a derived data type (use “”)}
### Variables in summary

<table>
<thead>
<tr>
<th>Variable declaration.</th>
<th>Assignment statement.</th>
<th>Initialized declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int value;</code></td>
<td><code>value = 5;</code></td>
<td><code>int value = 5;</code></td>
</tr>
</tbody>
</table>

```
0x000
0x001
0x002
0x003
```

This is a String *literal*. It will be printed as is.

```
System.out.print("The value is ");
System.out.println(value);
```

The integer 5 will be printed out here.

Notice no quote marks
Operators

- Programmers manipulate variables using the **operators** provided by the high-level language.
- You need to know the operators function, associativity, precedence, and the data type of the result.
- We’ve already seen the assignment operator =
- Java has 5 arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Binary</td>
<td>total = cost + tax;</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Binary</td>
<td>cost = total – tax;</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Binary</td>
<td>tax = cost * rate;</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Binary</td>
<td>salePrice = original / 2;</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>Binary</td>
<td>remainder = value % 5;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Right to left</td>
<td>x = -4 + 3;</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>(unary negation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* / %</td>
<td>Left to right</td>
<td>x = -4 + 4 % 3 * 13 + 2;</td>
<td>11</td>
</tr>
<tr>
<td>+ -</td>
<td>Left to right</td>
<td>x = 6 + 3 – 4 + 6 * 3;</td>
<td>23</td>
</tr>
</tbody>
</table>

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering
Calculate the reciprocal of a value (for example, 5)

```java
public class Reciprocal {

    /**
     * This program outputs \( f(x) = \frac{1}{x} \) for \( x = 5 \)
     */
    public static void main (String[] args) {

    }
}
```
Program design and control

- Top-down design
- Control flow: if and while
- Methods
- Debugging
Structured programming

- How do you design an algorithm to solve a complex problem?
  - Divide and conquer!
- Start with systematic decomposition of problem
  - “top-down” analysis
  - stepwise refinement
- The basic tools for decomposing a problem include:
  - Sequential execution
    - Do this and then do the next thing
  - Selection (Conditional)
    - Do this OR that
  - Iteration
    - Repeat that
  - Method calls
    - Do a task that has already been specified
Semantics of control flow

- Three control structures
  - Sequential
    - This is the default
  - Selective/Conditional
    - Branching or decision making
  - Iteration
    - Loops

- These concepts are universal to problem solving
  - There are many ways to specify these behaviors in programming languages
Syntax of control flow

- Sequential
  ...
  statement1;
  statement2;
  ...

- Selection
  ...
  if (condition) {
    statement(s);
  }
  ...

- Iterative
  ...
  while (condition) {
    statement(s);
  }
  ...

{ // a block of
  // statements are
  // treated as “one”
  // statement
  }

-or-

// a block of
// statements are
// treated as “one”
// statement

} else {
  statement(s);
}

}
Relational Operators

- Conditional tests must evaluate to TRUE or FALSE

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
</tr>
</tbody>
</table>
Stepwise refinement illustrated

Create a program that counts down from 100.

Sample output
100
99
98
...
1

Start
Count down from 100
Stop

Start
Starting number is 100
Count down from starting value one number at a time and displaying each number
Stop

Start
Starting number is 100
Done?
Display number
Get next number
Stop
Stepwise refinement illustrated

public class Main {

    /**
     * Display count down from 100.
     */
    public static void main (String[] args) {
        int currentCount = 100;

        while (currentCount > 0) {
            System.out.println(currentCount);
            currentCount = currentCount - 1;
        } // end while

    } // end method main

} // end class Main
Method calls

- Method is an OOP term
  - Commonly used terminology includes functions, procedures, or subroutines
- Used for
  - Decomposition
  - Frequently executed code segments
  - Library routines (pre-existing methods to perform common tasks)
- Requirements:
  - Pass parameters (inputs) and return values (if any)
  - Call from any point in the flow of the program and return control to the same point.
- Example:
  System.out.println(“Hello World”);
The Call / Return mechanism

- The figure illustrates the execution of a program comprising code fragments A, W, X, Y and Z.
  - Note that fragment A is repeated several times, and so is well suited for packaging as a subroutine:
Java API library

- Java sends information to the standard I/O devices by using a Java class stored in the standard Java library.
  - The console that starts a Java application is typically known as the standard output device.
  - The standard input device is typically the keyboard.
- Java classes in the standard Java library are accessed using the Java Applications Programming Interface (API).

```java
System.out.println("Hello World");
```

- This statement uses the System class from the Java API.
- The System class contains methods and objects that perform system level tasks.
- The out object, a member of the System class, contains the method println
A method for input

- A set of useful methods to get input are contained in the Scanner class.
- The Scanner class is defined in java.util, so we will use the following statement at the top of our programs:
  ```java
  import java.util.Scanner;
  ```

- Scanner objects work with System.in
- To create a Scanner object:
  ```java
  Scanner keyboard = new Scanner(System.in);
  ```

- Useful methods include
  - `nextInt()` - return the next input as an integer
  - `nextDouble()` - return the next input as a double
  - `nextLine()` - return the next line as a String
Example: input with nextInt()

```java
import java.util.Scanner;
public class Add {
    public static void main(String [] args){
        Scanner keyboard = new Scanner(System.in);
        int num1;
        int num2;
        int sum;
        System.out.println("Value 1? ");
        num1 = keyboard.nextInt();
        System.out.println("Value 2? ");
        num2 = keyboard.nextInt();
        sum = num1 + num2;
        System.out.println("Sum is ");
        System.out.println(sum);
    }
}
```
Debugging

- Compilers help find syntax errors (often with cryptic messages)
- Semantic errors require domain specific context to identify

- Most integrated development environments provide a debugging tool
- A debugging tool provides (at least) the ability to:
  - Stop execution when desired
    - Breakpoints allow the user to set points at which execution will halt & wait for a new “run” or “continue” instruction
  - Examine the contents of variables and memory locations
  - Execute instructions one at a time, or in small groups
    - Run will set the program running
    - Single step causes a single instruction to be executed
    - Step over causes the a called method to be executed
    - Step into follows the program into the called method
Style

- Programming languages have a variety of interesting features that, in general, novice programmers should avoid.
- These “shortcuts” are often the source of errors.
- Learn to avoid bad programming habits early!

Example

```c
int x, y;
x = (y = 5);
```

Vs.

```c
int numStudents = 5;
int numExams = numStudents;
```
Design/Debugging examples

- Input a number; count down from it to 0 then back
- Input two numbers; output their mean
- Input a rate of pay and a number of hours worked to calculate wage
Representing information

Bits, Bytes, and Memory
Encoding numbers
Encoding characters
The importance of type
Type casting
How do we represent information (data) in a form that is mutually comprehensible by human and machine?

- The devices that make up a computer are switches that can be on or off, i.e. at high or low voltage.
- Thus they naturally provide us with two symbols to work with: we can call them on & off, or (more usefully) 0 and 1.
  - Bit: Binary digIT

We will start by how to represent

- Integer numbers
- Floating point numbers
- Characters

Ultimately, we will have to develop schemes for representing all conceivable types of information - language, images, actions, etc.
Why do Computers use Base 2?

- Base 10 Number Representation
  - Natural representation for human transactions
    - $124.5 = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$
  - Even carries through in scientific notation
    - $1.5213 \times 10^4$
  - Hard to Implement Electronically
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
    - Hard to store/transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.

- Base 2 Number Representation
  - **Bit**: Binary digit
  - Easy to represent/store with electric current/bistable elements
  - Reliably transmitted on noisy and inaccurate wires
Main Memory

- Commonly implemented with random-access memory (RAM)
- Memory contains:
  - currently running programs
  - data used by those programs
- Memory is divided into addresses
- Each address contains:
  - A group of bits (each 1 or 0)
  - Often counted in bytes (groups of eight bits)
  - A word is often 32 or 64 bits
- How do we represent characters or numbers with bits?

Main memory can be visualized as a column or row of cells.

A byte-addressable memory
Unsigned Binary Integers

\[ Y = "\text{abc}\" = a \cdot 2^2 + b \cdot 2^1 + c \cdot 2^0 \]

(where the digits a, b, c can each take on the values of 0 or 1 only)

\[
\begin{align*}
N &= \text{number of bits} \\
\text{Range is: } 0 &\leq i < 2^N - 1 \\
\text{Umin} &= 0 \\
\text{Umax} &= 2^N - 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>4 bits</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x0</td>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>x1</td>
<td>0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>2</td>
<td>x2</td>
<td>0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td>3</td>
<td>x3</td>
<td>0011</td>
<td>0000 0011</td>
</tr>
<tr>
<td>4</td>
<td>x4</td>
<td>0100</td>
<td>0000 0100</td>
</tr>
<tr>
<td>5</td>
<td>x5</td>
<td>0101</td>
<td>0000 0101</td>
</tr>
<tr>
<td>6</td>
<td>x6</td>
<td>0110</td>
<td>0000 0110</td>
</tr>
<tr>
<td>7</td>
<td>x7</td>
<td>0111</td>
<td>0000 0111</td>
</tr>
<tr>
<td>8</td>
<td>x8</td>
<td>1000</td>
<td>0000 1000</td>
</tr>
<tr>
<td>9</td>
<td>x9</td>
<td>1001</td>
<td>0000 1001</td>
</tr>
<tr>
<td>10</td>
<td>xA</td>
<td>1010</td>
<td>0000 1010</td>
</tr>
<tr>
<td>11</td>
<td>xB</td>
<td>1011</td>
<td>0000 1011</td>
</tr>
<tr>
<td>12</td>
<td>xC</td>
<td>1100</td>
<td>0000 1100</td>
</tr>
<tr>
<td>13</td>
<td>xD</td>
<td>1101</td>
<td>0000 1101</td>
</tr>
<tr>
<td>14</td>
<td>xE</td>
<td>1110</td>
<td>0000 1110</td>
</tr>
<tr>
<td>15</td>
<td>xF</td>
<td>1111</td>
<td>0000 1111</td>
</tr>
</tbody>
</table>

Problem:
- How do we represent negative numbers?
Signed Magnitude

- Leading bit is the sign bit

\[ Y = \text{“abc”} = (-1)^a (b \cdot 2^1 + c \cdot 2^0) \]

Range is: \[-2^{N-1} + 1 < i < 2^{N-1} - 1\]

- \[ S_{\text{min}} = -2^{N-1} + 1 \]
- \[ S_{\text{max}} = 2^{N-1} - 1 \]

Problems:
- How do we do addition/subtraction?
- We have two numbers for zero (+/-)!

<table>
<thead>
<tr>
<th>Dec</th>
<th>4-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1111</td>
</tr>
<tr>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>-5</td>
<td>1101</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>+0</td>
<td>0000</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
</tr>
<tr>
<td>+2</td>
<td>0010</td>
</tr>
<tr>
<td>+3</td>
<td>0011</td>
</tr>
<tr>
<td>+4</td>
<td>0100</td>
</tr>
<tr>
<td>+5</td>
<td>0101</td>
</tr>
<tr>
<td>+6</td>
<td>0110</td>
</tr>
<tr>
<td>+7</td>
<td>0111</td>
</tr>
</tbody>
</table>
Two’s Complement

- Transformation
  - To transform a into -a, invert all bits in a and add 1 to the result

\[
\begin{align*}
\text{Range is: } & -2^{N-1} < i < 2^{N-1} - 1 \\
T_{\text{min}} & = -2^{N-1} \\
T_{\text{max}} & = 2^{N-1} - 1
\end{align*}
\]

Advantages:
- Operations need not check the sign
- Only one representation for zero
- Efficient use of all the bits
Manipulating Binary numbers

- Binary to Decimal conversion & vice-versa
  - A decimal number can be broken down by iteratively determining the highest power of two that “fits” in the number:
    - e.g. $(4)_{10} =>$
    - e.g. $(13)_{10} =>$ 
    - e.g. $(0.75)_{10} =>$ 

- Binary mathematics

\[
\begin{array}{c}
1101 \\
+ 1011 \\
\hline
1000 \\
\end{array}
\]

\[
\\Rightarrow 8 \text{ (as expected!)}
\]

- What if the value of the answer can not be represented?
  - Overflow!
Dangers of abstraction!

- Assume machine with 32 bit word size, two’s complement integers
- For each of the following expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

**Initialization**

```c
int x = foo();
int y = bar();
int z = foobar();
```

- Is \((x*x >= 0)\) always true?
- If \(x<0\), is \((2*x) < 0\) true?
- Is \((x+(y+z) == (x+y)+z)\) true?
- What if \(x\), \(y\), and \(z\) are of floating point numbers?
Real numbers

- Most numbers in the “real” world are not integers!
- Say you have space to represent a ten digit decimal number. Where would _you_ put the decimal place?
  - Range: The magnitude of the numbers we can represent (determined by # of bits):
    - e.g. with 32 bits the largest representable number is \( \sim +/- 2 \text{ billion} \), too small for many purposes!
  - Precision: The exactness with which we can specify a number (determined by # of bits):
    - e.g. a 32 bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal representation

- Our decimal system handles non-integer real numbers by adding yet another symbol - the decimal point (.) to make a fixed point notation:
  - e.g. \( 3,456.78 = 3.10^3 + 5.10^1 + 4.10^0 + 6.10^{-1} + 7.10^{-2} \) or \( 3.45678 \times 10^3 \)

- The floating point, or scientific, notation allows us to represent very large and very small numbers (integer or real), with as much or as little precision as needed.
  - e.g. \( 25.75_{10} = 1.2_4^4 + 1.1_2^3 + 1.1_2^1 + 1.2_2^{-1} + 1.2_2^{-2} = 11001.110_2 \) or \( 1.1001110 \times 2^4 \)
  - As a literal \( 2.75E1 \) (E-notation)
IEEE-754 fp numbers

- **Double** precision (64 bit) floating point

  \[
  N = (-1)^s \times 1.fraction \times 2^{(\text{biased exp.} - 1023)}
  \]

- **Range & Precision:**
  - 32 bit:
    - mantissa of 23 bits + 1 => approx. 7 digits decimal
    - \(2^{+/127}\) => approx. \(10^{+/38}\)
  - 64 bit:
    - mantissa of 52 bits + 1 => approx. 15 digits decimal
    - \(2^{+/1023}\) => approx. \(10^{+/308}\)
# Numeric data types in JAVA

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>Integers in the range -128 to +127</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>Integers in the range of -32,768 to +32,767</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>Integers in the range of -2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>Floating-point numbers in the range of ±3.4×10^{-38} to ±3.4×10^{38}, with 7 digits of accuracy</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>Floating-point numbers in the range of ±1.7×10^{-308} to ±1.7×10^{308}, with 15 digits of accuracy</td>
</tr>
</tbody>
</table>
Data type of result

- Problems arise when you attempt to mix two different data types
- Be wary of compiler assumptions!
  - In Java integer literals are “cast” as int
  - In Java real number literals are “cast” as double
  - Results of mixed types are *promoted* to the larger type
- Examples
  - 2/3 is cast as an integer (specifically 0)
  - 2.0/3 is cast as a double (specifically 0.666…)
- Java automatically converts lower precision types to higher precision types but not visa-versa!
  - byte -> short -> int -> long -> float -> double

What about?
int x = 2.0/3;

found : double
required: int
  int x = 2.0/3;
1 error
BUILD FAILED (total time: 0 seconds)

What about?
float x = 2.0/3;

found : double
required: float
  float x = 2.0/3;
1 error
BUILD FAILED (total time: 0 seconds)
Shortcuts and Pitfalls: Typecasting

- Addition/Subtraction: If mixed types, smaller type is "promoted" to larger.
  - \( x + 4.3 \) // answer is implicitly promoted (type cast) to double

- Division: If both operands are of integer type, the default result is a truncated integer
  - \( \text{int} \ x = 5; \text{double} \ d = 5.0; \)
  - ( \( x / 3 = = 1 \) ) // true (Not 1.6, not 2, but 1!)
  - ( \( d / 3 = = 1 \) ) // false (mixed operands, result promoted)
  - ( (double) \( x / 3 = = 1 \) ) // false (explicit typecasting)

- The rules can be overridden by typecasting the operands or result
  - Put the name of the type in parentheses before the RHS variable or result
    - float \( x = (\text{float}) (5/3.0) \);

- Typecasting tells that the programmer KNOWs what they are doing.
  - Use type casting with care!
Representing text input (characters)

- Each character encoded in a fixed number of bits
  - C/C++ uses standard 7-bit ASCII encoding
  - Java uses 16-bit UNICODE
    - UNICODE is a superset of ASCII that includes characters for non-English alphabets
- Character ‘A’ has code x0065
- Character ‘0’ has code x0030 (48 in decimal)
- Note that
  - ‘0’ != 0
  - (int) ‘0’ == 48
  - ‘0’ + 2 == 50
- Escape sequences used for special characters
- Java supports the primitive data type char
- Strings are not a primitive data type
  - but strings are a built in class in Java

```java
STRING number = "10213";
```

```
number

\[\begin{array}{|c|c|}
\hline
x00A0 & x00F0 \\
\hline
\end{array}\]

\[\begin{array}{|c|c|}
\hline
x00F0 & x0031 \\
\hline
x00F1 & x0030 \\
\hline
x00F2 & x0032 \\
\hline
x00F3 & x0031 \\
\hline
x00F4 & x0033 \\
\hline
x00F5 & x0000 \\
\hline
\end{array}\]
```

Hex
### Escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n newline</td>
<td>Advances the cursor to the next line for subsequent printing</td>
</tr>
<tr>
<td>\t tab</td>
<td>Causes the cursor to skip over to the next tab stop</td>
</tr>
<tr>
<td>\b backspace</td>
<td>Causes the cursor to back up, or move left, one position</td>
</tr>
<tr>
<td>\r carriage</td>
<td>Causes the cursor to go to the beginning of the current line, not the next line</td>
</tr>
<tr>
<td>\ backslash</td>
<td>Causes a backslash to be printed</td>
</tr>
<tr>
<td>' single quote</td>
<td>Causes a single quotation mark to be printed</td>
</tr>
<tr>
<td>&quot; double quote</td>
<td>Causes a double quotation mark to be printed</td>
</tr>
</tbody>
</table>
Boolean data types

- Two valued: true or false
- The boolean data type is often used to score the result of a conditional test
- The boolean literals are the keywords true and false (no quotes).
  - e.g. boolean moreWorkToDo = false;

- A predicate method is one that returns a boolean value
- The Character class has several predicate methods, including:
  - Character.isDigit(char)
  - Character.isLetter(char)
  - Character.isLowerCase(char)
  - Character.isUpperCase(char)
  - Character.isWhitespace(char)
String: A simple object

- String is a derived data type (an object)
- String is such a useful data type that it is built into the language (java.lang)
  - String has its own context for +
    - Concatenate
  - String objects are built automatically (without the need to invoke new).
  - String objects come with useful methods like length(), toLowerCase(), and charAt()

String message1 = “Hello” + “ ” + “World” + “\n”;
String message2 = “Goodbye”;
char letter = message2.charAt(0);

Primitive data types store values

```
int x = 10; int y = 5;
x = y;
x = 5; y = 5
```

Objects use references/pointers

```
message1 = Hello World
message2 = Goodbye
message1 = message2;
message1
message2
```

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering

CS 240
Computer Science I
Enumerated Types

- Enumerated types allow each value of a user-defined type to be described in English. This is a major point of style!
- Enumerated types are a powerful alternative to schemes in which you explicitly say “1 stands for red, 2 stands for green, 3 stands for blue,…”
- if (chosenColor == 1) Vs. if (chosenColor = Color.RED)
- int result = retrievePayrollData ( data, true, false, false)
  Vs.
  int result = retrievePayrollData (data,
  EmployeeStatus.CURRENT_EMPLOYEE, MarriageStatus.MARRIED,
  PayrollType.SALARIED)
- enum TypeName { One or more comma separated labels }
Enumerated Types: Syntax

- Each value of an enumerated type is an object of type Day and should be treated as such.
- Each value is ordered (ordinal value) from 0 to last value.
- ```
    enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
    Day today;
    today = Day.MONDAY;
    if (today == Day.MONDAY) ...
    if (today.equals(Day.MONDAY) ... // Better
    If (today.compareTo(Day.MONDAY) // -1 <, 0 =, +1 >
  ```

- System.out.println(today) produces MONDAY
- today.ordinal() produces 1
- Day.SUNDAY.ordinal() produces 0
Scope: Global and Local

- Where is the variable accessible?
- All most programming languages variables are defined as being in one of two storage classes
  - Automatic storage class (on the stack, may be uninitialized in some languages)
  - Static storage class (in memory, initialized to 0)
- Compiler infers scope from where variable is declared unless specified
  - programmer doesn't have to explicitly state (but can in many languages!)
    - automatic int x;
    - static int y;
- Global: accessed anywhere in program (default static)
  - Global variable is declared outside all blocks
- Local: only accessible in a particular region (default automatic)
  - Variable is local to the method/block after the point it is declared
  - block defined by open and closed braces { } in most languages
  - can access variable declared in any "containing" block
public class Main {
    static int itsGlobal = 0;

    public static void main (String[] args) {
        System.out.println (itsGlobal);
        do {
            int itsGlobal = 1; /* local to loop */
            System.out.println (itsGlobal);
        } while (false);
        System.out.println (itsGlobal);
        int itsGlobal = 2;
        System.out.println (itsGlobal);
    } // end method main
} // end class Main

• Output
  0
  1
  0
  2
Summary I: Representing information

- Declarations allocate an area of main memory to hold information of the specified data type
  - The value stored in memory can be changed (variable / mutable)
  - The “final” keyword can fix an initial value (constant / unmutable)
    - e.g. final double PI = 3.14; // Style hint: Use ALL_CAPS for named constants
- This named area of memory is available from the time it is declared until the end of the method
  - These are “local” variables
  - After the variable falls out of scope memory it is reclaimed
    - “cannot resolve symbol” error
- Carefully decide what data types to use to store information
  - Consider your result values
  - Use typecasting with caution
    - Beware of exceeding the precision of your data type
Summary II: Representing information

- Semantics – abstract place to hold information
  - However, we must be aware of the realities/limitations of the implementation!
- Syntax – type identifier (or cast)
  - Identifier characters (a-z, A-Z, 0-9, _, $)
  - Leading character must not be a digit
  - Case sensitive
  - Scope
- Style
  - Naming conventions
    - numOfStudents Vs. num, n, students, loopCheckValue, tempInt
    - NUM_OF_STUDENTS (for unchanging constant values)
  - Declaration conventions
    - Declaration block
    - Just in Time
  - Initialization conventions
Real world realities: Here be dragons!

```java
System.out.println
(3.14 * 3.0 * 3.0);
System.out.println
(3.0 * 3.0 * 3.14);

// count from 0.0 to 1.5 by 0.1
double count = 0.0;
while ( count <= 1.5 ) {
    System.out.println(count);
    count = count + 0.1;
}
```

28.259999999999998
28.26
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
1.4000000000000001
Pitfall: Ariane 5

- **Danger!**
  - Computed horizontal velocity as floating point number
  - Converted to 16-bit integer
  - Worked OK for Ariane 4
  - Overflowed for Ariane 5
    - Used same software

- **Result**
  - Exploded 37 seconds after liftoff
  - Cargo worth $500 million
Introduction to methods

- Semantics of methods
- Using methods
- Writing methods
- Commonly used Java library methods
Classes, objects, methods, and fields

- Objects: Software entities that contain attributes (fields) and methods (functions)
- Class: The code/blueprint that describes related fields/methods
  - The “data type” of an object
  - Objects are *instances* of a class (created using “new className”).
    - Zero or more objects may be created
    - Some class methods can be used without an object/instances
- Fields: Typed variables contained “in” the object
  - Object.field (note that the field may, itself, be an object)
- Methods: Functions that the object/class is capable of performing
  - Class.method
  - Object.method
Semantics of Methods

- Smaller, simpler, subcomponent of program
- Provides abstraction
  - hide low-level details
  - give high-level structure to program,
    easier to understand overall program flow
  - enables separable, independent development
- Method calls
  - zero or multiple arguments passed in
  - single result returned (optional)
    - Void
    - Data type
  - by convention, only one method named main (this defines starting point)
- Methods must be declared/defined
- In other languages, called functions, procedures, subroutines, ...
Methods in Java

- A method consists of
  - Declaration
    - includes return value, function name, and the order and data type of all arguments
    - names of argument are optional
  - Definition
    - Names of variables do not need to match prototype, but must match order/type
    - Defines functionality (source code) and returns control (and a value) to caller
    - May produce side-effects

- Declaration
  public static int sum (int num1, int num2)

- Definition
  public static int sum (int num1, int num2) {
    return (num1 + num2);
  }

- Call
  totalGrade = sum (exam1, exam2);
The Method call

- A `void` method is one that simply performs a task and then terminates.
  
  ```java
  System.out.println("Hi!");
  ```
- A value-returning method not only performs a task, but also sends a value back to the code that called it.
  
  ```java
  int number = keyboard.nextInt();
  ```
- Values that are sent into a method are called **arguments**.
  
  ```java
  System.out.println("Hello");
  int length = Math.sqrt(area);
  ```
- The data type of an argument (a.k.a. actual parameter) in a method call must correspond to the method declaration.
- Parameters (a.k.a. formal parameters) are variables in the called method that holds the value/arguments being passed.
A method call illustrated

displayValue(5);   // Method Call

The argument 5 is copied into the parameter variable num.

// Method Declaration
public static void displayValue(int num){
    // Method Definition
    System.out.println("The value is " + num);
}

Argument (or actual parameter): 5  What is displayed?
Parameter (or formal parameter): num  What is returned?
Passing multiple arguments

The argument 5 is copied into the \texttt{num1} parameter.

\begin{verbatim}
showSum(5,10);
\end{verbatim}

The argument 10 is copied into the \texttt{num2} parameter.

\begin{verbatim}
public static void showSum(double num1, double num2){
    double sum;     // to hold the sum
    sum = num1 + num2;
    System.out.println("The sum is "+sum);
}
\end{verbatim}

\textbf{Arguments? Parameters?}
Calling a value-returning method

total = sum(value1, value2);

public static int sum(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
Methods and your data

- A local variable is declared inside a method and is not accessible to statements outside the method.
  - **Scope**
  - Different methods can have local variables with the same names because the methods cannot see each other’s local variables.
  - A method’s local variables exist only while the method is executing.
- The values/arguments passed to the parameters of the method become local variables to that method.
  - **Call by value**
    - Changes to the local copies of variables do not change the original
    - Use returned value to make changes to primitive data types
- If references to an object are passed to an method, then the method makes a copy of the reference and can use that reference to make changes to the aliased object.
  - **Call by reference**
public class Main {

    static int increment(int x) {
        x = x + 1;
        return x;
    } // end method increment

    public static void main(String[] args) {
        int x = 0;
        int y = increment(4);
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    } // end method main

} // end class Main
Under the hood

- Making a method call involves three basic steps
  - The arguments of the call are passed from the caller to the callee
  - The callee does its task
  - A return value is returned to the caller
- The run time-stack

**Before call**

**During call**

**After call**
Library methods

- Java comes complete with an extensive library of generally useful methods
  - `java.lang` (default)
  - `java.util` (must import)

- Methods are associated with a class or an object
  - Object = an *instance* of a class

```java
import java.util.Scanner;
...
Scanner keyboard;
keyboard = new Scanner(System.in);
System.out.println("Hi");
int number = keyboard.nextInt();
class. object.method();
object.method();
```
Math class methods

- The Java API library provides a class named Math
  - java.lang.Math

- The Math class has a large number of useful methods and a couple of useful constant fields (PI, E).

- Example: Calculate the area of a circle

```
  double y = 3.14 * x * x;

  double area = Math.PI *
                Math.pow(radius, 2);
```

floor(x)
ceil(x)
round(x)
sin(x)
cos(x)
tan(x)
atan(x)
asin(x)
acos(x)
log(x)
exp(x)
pow(x, y)
sqrt(x)
min(x, y)
max(x, y)
...
Learn how to find methods

- The IDE will help you find the right method and explain what it returns
- You have to know where to start looking!

- Note that most methods allow multiple different argument types
- This is called overloading
String class methods

- The Java language provides a class named String
- Example: Find the character at the third position of “dogma”;
  
  ```java
  String word = "dogma";
  char letter = word.charAt(2);
  ```

- String S1 = new String ("ABC");
- String S2 = new String ("ABC");

  ```java
  (S1 == S2) is false  // PITFALL!
  S1.equals(S2) is true
  (S1.compareTo(S2) == 0) is true
    - Zero if equal, neg if S1 < S2, pos if S1 > S2
  ```
## I/O methods

- We’ve seen the **Scanner** class for input
  - `byte nextByte()`
  - `double nextDouble()`
  - `float nextFloat`
  - `int nextInt()`
  - `string nextLine()`
  - `long nextLong()`
  - `short nextShort()`

- We’ve seen the **System** class for output
  - `system.out.println( value )`
    - `println (int)`
    - `println (double)`
    - `println (string)`
  - `system.out.print (value)`
    - `println (value);`
    - `print (value);`
    - `print ("\n");`
System.out.printf()

system.out.printf( format, v1, v2, .. vn)

Format string:
   %[flags][width][.precision]type

System.out.printf("UID=%d\n", uid);
System.out.printf("UID=%d\tlogin=%s\n", uid, loginName);

printf("%+-5d  %+5d", 10, -20)
+1 0 _ _ _ _ - 2 0
printf("%8.3f  %7.2f",1.234567,1.234567)
_ _ _ 1 . 2 3 4 _ _ _ 1 . 2 3

<table>
<thead>
<tr>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left justified</td>
</tr>
<tr>
<td>+</td>
<td>prefix with +/-</td>
</tr>
<tr>
<td>0</td>
<td>pad with zeros</td>
</tr>
<tr>
<td></td>
<td>separate by thousands</td>
</tr>
<tr>
<td></td>
<td>negatives in parens</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>integer (digits)</td>
</tr>
<tr>
<td>%f</td>
<td>floating point</td>
</tr>
<tr>
<td>%e</td>
<td>exponential /scientific notation (floatingpoint)</td>
</tr>
<tr>
<td>%b</td>
<td>boolean</td>
</tr>
<tr>
<td>%c</td>
<td>character</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
</tr>
</tbody>
</table>
Buffered I/O

- In many systems, characters are buffered in memory during an I/O operation.
  - Conceptually, each I/O stream has its own buffer.
- Keyboard input stream
  - Characters are added to the buffer only when the newline character (i.e., the "Enter" key) is pressed.
  - This allows user to correct input before confirming with Enter.
- Output stream
  - Characters are not flushed to the output device until the newline character is added.
- Advantages/Disadvantages?
In-class examples

- Subsequence finder
  - Input a sequence of characters, a starting position, and a length.
  - Output the subsequence
  - Package I/O in main and functionality as a method

- Pattern finder
  - Input a sequence of characters and a three character subsequence
  - Output every starting position of the subsequence in the sequence
  - Package functionality as a method, return “next” position found
Control flow: Selection

if-then-else
nested if statements
Logical operators
Switch
Conditional operator
Semantics of control flow

- Three control structures
  - Sequential
    - This is the default
  - Selection/Conditional
    - Branching or decision making
  - Iteration
    - Loops

- These concepts are universal to problem solving
  - There are many ways to specify these behaviors in programming languages
Syntax of control flow

- **Sequential**
  
  ... 
  
  statement1;
  
  statement2;
  
  ...
  
  {
  // a block of
  // statements are
  // treated as “one”
  // statement
  }

- **Selection**
  
  ... 
  
  if (condition) {
    statement(s);
  }
  
  ...
  
  -or-
  
  if (condition) {
    statement(s);
  } else {
    statement(s);
  }

- **Iteration**
  
  ... 
  
  while (condition) {
    statement(s);
  }
  
  ...
Selection control: if statements

if (BooleanExpression)
    statement;

if (BooleanExpression)
    block of statements;
else
    statement2;

if (BooleanExpression)
    block of statements1;
else {
    block of statements2;
}

if (ageInYears < 0) {
    handleError();
} else {
    employee.age = ageInYears;
}
Nested ifs

```java
if (BooleanExpression1){
    if (BooleanExpression2) {
        block of statements;
    }
}

if ( student.InCourse(CS240) ) {
    if (date < lab1.dueDate()) {
        lab1.allowTurnIn();
    } else { // lab is late
        printErrorMessage("This lab will" +
        " not be accepted late.\n");
    } else { // student is not in CS240
        printErrorMessage("Only CS240 students " +
        " may turn in this lab.\n");
    }
}
```
## Predicate logic and logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>Connects two boolean expressions into one. Both expressions must be true for the overall expression to be true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>The ! operator reverses the truth of a boolean expression.</td>
</tr>
</tbody>
</table>

| Expression A | Expression B | !A | !B | A && B | A || B | !A && !B | !(A && B) |
|--------------|--------------|----|----|--------|--------|----------|----------|
| false        | false        | true | true | false  | false  | true     | true     |
| false        | true         | true | false | false  | true   | false    | true     |
| true         | false        | false | true | false  | true   | false    | true     |
| true         | true         | false | false | true   | true   | false    | false    |
Best programming practices: Flags

- It is generally a bad practice to repeat a test in a program
  - Someone updating the test may only see it in one place!

- A flag is a boolean variable that monitors some condition in a program.
- When a condition is true, the flag is set to true.
- The flag can be tested to see if the condition has changed.

  \[
  \text{highScore} = (\text{average} > 95);
  \]

- Later, this condition can be tested:

  ```java
  if (highScore) {
      System.out.println("That's a high score!" );
  }
  ```
# Order of Precedence

<table>
<thead>
<tr>
<th>Order of Precedence</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!</td>
<td>Unary negation, logical NOT</td>
</tr>
<tr>
<td>2</td>
<td>* / %</td>
<td>Multiplication, Division, Modulus</td>
</tr>
<tr>
<td>3</td>
<td>+ -</td>
<td>Addition, Subtraction</td>
</tr>
<tr>
<td>4</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Less-than, Greater-than, Less-than or equal to, Greater-than or equal to</td>
</tr>
<tr>
<td>5</td>
<td>== !=</td>
<td>Is equal to, Is not equal to</td>
</tr>
<tr>
<td>6</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>=</td>
<td>Assignment operator.</td>
</tr>
</tbody>
</table>
Logical operator

```java
if(x > y && y > z) {
    System.out.println("true");
}

((x > y) && (y > z))
((x > y) || (y > z))
!(x == 4)
!(!!(!(x)))
!(x<y)

Short circuiting
Does test get invoked if x = 2?  x = 0?

if( (x > 1) || test() ) {
    // do something
}

!(x<y)

if( (x > 1) && test() ) {
    // do something
}

Boolean laws
Distribution  A&&(B||C) == AB || AC
DeMorgan’s    !(A||B) == !A&&!B
             !(A&&B) == !A||!B
```
Shortcuts and pitfalls

- **Order of Precedence**
  - The `!` operator has a higher order of precedence than the `&&` and `||` operators.
  - The `&&` and `||` operators have a lower precedence than relational operators like `<` and `>`.  
  - Style hint: Always fully parenthesize

- **Short Circuiting - Logical AND and logical OR operations perform short-circuit evaluation of expressions.**
  - Logical AND will evaluate to false as soon as it sees that one of its operands is a false expression.
  - Logical OR will evaluate to true as soon as it sees that one of its operands is a true expression.
  - Style hint: Avoid state changing behavior in logical expressions
Shortcuts and pitfalls

- if and {}
  - The if statement does not require curly braces for single statement blocks
  - Style hint: ALWAYS include the braces.

```java
if (numStudents > 5) {
    numExams = numStudents;
    giveExams();
}
```

```java
if (numStudents > 5) {
    numExams = numStudents;
} giveExams();
```
Shortcuts and pitfalls

- Beware of using the assignment operator (=) when you want to use the Boolean equality operator (==)
  - This will result in an incompatible type error
  - required: boolean, found: int

```java
int y = 5;
if (y=5) {
    System.out.println("Howdy");
}
```

- Beware of using single | or single & instead of || and &&
  - | is the bitwise OR operator
  - & is the bitwise AND operator
  - We’ll cover these later!
Switch

- Expression must be non-floating point primitive (usually integer or char)
- Cases are literal values

```java
char gender;
// ...
switch ( gender ){
    case 'f':
        System.out.println("Female");
        break;
    case 'm':
        System.out.println("Male");
        break;
    default:
        System.out.println("Invalid gender");
} // end switch
```

- keyword break
  - Exits from local control block
- Style hint: Except for in switch statements we will avoid for now
Conditional operator

- variable name = boolean condition ? value if true : value if false

String legalClassification = (age > 18) ? “Adult” : “Child”;

Vs.

String legalclassification;
if (age > 18) {
    legalClassification = “Adult”;
} else {
    legalClassification = “Child”;
}

- Style hint: Avoid the use of this unnecessary (albeit cool) operator.
Control flow: Iteration
Semantics of control flow

- Three control structures
  - Sequential
    - This is the default
  - Selection/Conditional
    - Branching or decision making
  - Iteration
    - Loops

- These concepts are universal to problem solving
  - There are many ways to specify these behaviors in programming languages
## Syntax of control flow

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Selection</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>statement1;</td>
<td>if (condition) {</td>
<td>while (condition) {</td>
</tr>
<tr>
<td>statement2;</td>
<td>statement(s);</td>
<td>statement(s);</td>
</tr>
<tr>
<td>...</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>{</td>
<td>}</td>
<td>...</td>
</tr>
<tr>
<td>// a block of</td>
<td>-or-</td>
<td></td>
</tr>
<tr>
<td>// statements are</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// treated as “one”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

// a block of statements are treated as “one” statement

Statement
while statements semantics

- Sound iteration structures require
  - (1) Initialization of values
  - (2) Test expression
  - (3) Increment/update expression values

- What do you want to test?
- How does that value change?
- What should its initial value be?
- When should the test occur?

- Infinite loops have a static test (true)
while statement syntax

```java
while (BooleanExpression)
    statement;

while (BooleanExpression) {
    block of statements;
}
```

```java
int i = 0;
while (true) {
    System.out.println(i);
    i = i + 1;
} // end infinite loop
```

```java
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i = i + 1;
} // end while
```
while statement style

```java
int i = 1;
while (true) {
    System.out.println(i);
    i = i + 1;
} // end infinite loop

while (true) {
    doTask();
} // end infinite loop

int i = 1;
while (i <= 5) {
    System.out.println(i);
    i = i + 1;
} // end while

boolean atMaxValue = false;
int i = 1;
while (!atMaxValue) {
    System.out.println(i);
    i = i + 1;
    atMaxValue = (i >= 5);
} // end while

final int MAX_RECORD_NUMBER = 5;
boolean moreRecords = true;
int recordNumber = 1;
while (moreRecords) {
    System.out.println(recordNumber);
    recordNumber = recordNumber + 1;
    if (recordNumber >= MAX_RECORD_NUMBER) {
        moreRecords = false;
    }
} // end if
} // end while
```
pitfalls

● Off by one errors

```java
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i = i + 1;
} // end while
```

```java
int i = 0;
while (i <= 5) {
    i = i + 1;
    System.out.println(i);
} // end while
```

● Accidental infinite loops

```java
int i;
// ... what if i = 6?
while (i != 5) {
    System.out.println(i);
    i = i + 1;
} // end while
```

```java
int i;
// ... what if i = 6?
while (i < 5) {
    System.out.println(i);
    i = i + 1;
} // end while
```

● Do not monkey with loop indices!
do
  statement;
while (BooleanExpression);

do {
  block of statements;
} while (BooleanExpression);

int i = 0;
do {
  System.out.println(i);
  i = i + 1;
} while (i <= 5);

int i = 0;
do {
  i = i + 1;
  System.out.println(i);
} while (i <= 5);
For loop statement

- **Syntax**
  ```java
  for (initialization; test; update) {
      Block of statements;
  } // end for
  ```

- **Semantics**
  - **Pre-test**
  - Equivalent to while

- **Style**
  - Use to count through an exact number of iterations
  - Always use braces
  - Best practices dictate that variables in a for loop not be used outside of the loop body
  - Declaring the loop variable in the for statement limits the scope

```java
int i;
for (i = 1; i <= 5; i = i+1) {
    System.out.println(i);
} // end for
```

```java
for (int i = 1; i <= 5; i = i+1) {
    System.out.println(i);
} // end for
```
Nested loops

- A loop can contain any statement, including another loop
  - Each iteration of the outer loops causes an inner loop to iterate to completion

```java
for (int i = 1; i <= 5; i = i+1) {
    for (int j = 1; j <= 5; j = j+1){
        if (i == j) {
            System.out.print("\n");
        } else {
            System.out.print("*")
        } // end if
    } // end for j
} // end for i
```
Jump statements: break, continue, return, try/finally

- **Return statement:** passes control of a method back to its caller
- **Break statement:** Exits current substatement block of a loop or switch statement
  - Passes control to the next line after the current (innermost) iteration
  - Labels can be used to break from outer loops
- **Continue statement:** Skips all remaining statements and goes immediately to the next iteration of the innermost loop
  - Labels can be used
- **Finally statement**
  - Later!
- **Style hints:**
  - Avoid using break/continue for flow control.

```java
myLoop: while (true) {
    i = input.nextInt();
    doSomething(i);
    if (i==0) break myLoop;
} //end while
```
Using loops to gather input

- Loops are used to perform repetitive tasks
  - Getting input is often repetitive
  - What do you do if you don’t know how much input you have?

- Standard test conditions include:
  - Input the count of the size of the input as the first input
    - E.g. 5, 2, 4, 2, 1, 7
  - Use a user specified sentinel value
    - E.g. 1, 5, 6, 3, -1
  - Test for the end of an input stream using standard sentinels
    - The EOF control character is embedded as a sentinel value by every input device to signal the end of the transmission
Nested loops example

- Say that we need to find the longest common subsequence shared by two strings
  - Very common problem
  - Biological sequence search
- Dot-plot: a visual representation

```
ATGGCATTATGGA
C .
A . . . .
T . . . .
G . . . .
G . . . .
A . . . .
```
More about operators

Combined assignment operators = += -= etc
  Increment/Decrement operators
  Bitwise operators
  Shift operators
  Operator associatively
  Operator precedence
  Implicit/Explicit Type conversions
Combined assignment operators

- Many expressions modify an existing variable
  - Ex: \( x = x + 1; \)

- Combined assignment operators implicitly include the Left Hand Side variable on the Right Hand Side as well
  - Ex: \( x += 1; \)

- A combined assignment operator exists for each of the 5 arithmetic operators: +=, -=, *=, /=, %= 

- Style hint: Use (or do no use) these operators consistently!
Increment and decrement operators

- Incrementing/Decrementing a variable by one is a very common task
  - Ex: \( x = x + 1; \quad y = y - 1; \)
- The increment/decrement operators allow a shorthand notation
  - Ex: \( x++; \quad y--; \)
- \( ++x \) is a post-increment operator
  - Ex: \( y = x++; \quad y = x; \quad x = x + 1; \)
- \( ++x \) is a pre-increment operator
  - Ex: \( y = ++x; \quad x = x + 1; \quad y = x; \)
- Style hints: Use (or do not use) these operators consistently!
  - \( x++ \) is frequently used as the increment expression in for-loops
Bitwise operators

- Bitwise operators allow you to directly manipulate the bits in a variable
  - Generally this is not something that you will need to do as a novice programmer
  - Systems level programmers need to do this often
  - Generally you can **not** perform these tasks with any other operator

- Bitwise AND:  &
  - $11309 \& 798 = 0000 0000 0000 1100$
  - $798 \& 1111 1100 1110 0001$

- Bitwise OR:  |
  - $11309 | 798 = 0010 1111 0011 1111$
  - $798 | 1111 1100 1110 0001$

- Bitwise Exclusive OR (XOR):  ^
  - $11309 ^ 798 = 0010 1111 0011 0011$
  - $798 ^ 1111 1100 1110 0001$

- Bitwise negation:  ~
  - $\sim 798 = 1111 1100 1110 0001$
  - $\sim 11309 = 0000 0000 0000 1100$

- Literal values for hexadecimal values:  int x = 0xFF;

- These operators also work on boolean typed data (0 is false, 1 is true)

- Pitfall: what does \texttt{if ( x \& y )} do?
Shift operators

- Shift operators move (shift) all the bits in a number to the left (towards the Most Significant Bit) or to the right (towards the Least Significant Bit)

- There are three bitwise shift operators
  - Left shift: `<<`
  - Signed right shift: `>>`
  - Unsigned right shift: `>>>`

- Assume x is a 8-bit variable (for ease of illustration)

<table>
<thead>
<tr>
<th>x</th>
<th>x &lt;&lt; 2</th>
<th>x &gt;&gt; 2</th>
<th>x &gt;&gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0001 1111</td>
<td>0111 1100</td>
<td>0000 0111</td>
</tr>
<tr>
<td>-17</td>
<td>1110 1111</td>
<td>1011 1100</td>
<td>1111 1011</td>
</tr>
</tbody>
</table>
Bit masking

- In the real world, mask is another term for a stencil
- In computer science, a **mask** is some data that, along with an operation, is used in order to extract information stored elsewhere.
- The most common mask used, also known as a **bitmask**, extracts the status of certain bits in a binary string or number (a bit field or bit array).
- For example, if we have the binary string 10011101 and we want to extract the status of the fifth bit counting along from the most significant bit, then we would use a bitmask such as 00001000 and use the bitwise AND operator.
  - Recalling that 1 AND 1 = 1, with 0 otherwise, we find the status of the fifth bit, since 10011101 AND 00001000 = 00001000
- Likewise we can set the fifth bit by applying the mask to the data using the OR operator.
Example: Networking

- Internet protocol addresses (IP addresses) are stored as a 32-bit value.
- They are communicated as a set of four octets
- Each octet is specified in a network class (A-D)

Consider: 129.122.5.16
- In Hex: 0x817A050F
- In Decimal: -2122709745

Given an integer containing a 32-bit value, how would you
- Determine the class A subnet for routing?
- Output the address in IP address format?
- Change the class D address to broadcast (255)?
24-bit RGB colors

- A common way to encode color is to encode the intensity of Red, Green, and Blue contained in the color.
- 24-bit colors represent each color on a scale from 0-255
- 8 bits x 3 colors = 24 bits

What bitwise instruction extracts just the red component?
What bitwise instruction sets the green component to 15 without changing the other components?
## Complete Java Order of Precedence

<table>
<thead>
<tr>
<th>Order</th>
<th>Operator</th>
<th>Operation</th>
<th>Associates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( )</td>
<td>parenthesis</td>
<td>L to R</td>
</tr>
<tr>
<td>1</td>
<td>[ ] . ,</td>
<td>array subscript, member selection, comma delimiter, post increment, post decrement</td>
<td>L to R</td>
</tr>
<tr>
<td></td>
<td>++ --</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>++ -- + - !</td>
<td>prefix increment, prefix decrement, positive, negative, NOT</td>
<td>R to L</td>
</tr>
<tr>
<td>3</td>
<td>(type) new</td>
<td>type cast, object instantiation</td>
<td>R to L</td>
</tr>
<tr>
<td>4</td>
<td>* / %</td>
<td>multiplication, division, modulo</td>
<td>L to R</td>
</tr>
<tr>
<td>5</td>
<td>+ - +</td>
<td>addition, subtraction, string concatenation</td>
<td>L to R</td>
</tr>
<tr>
<td>6</td>
<td>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</td>
<td>left shift, right shift, unsigned right shift</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&gt;= &lt;= &gt; &lt; instanceof</td>
<td>greaterThanOrEqual, lessThanOrEqual, greaterThan, lessThan, type comparison</td>
<td>L to R</td>
</tr>
<tr>
<td>8</td>
<td>== !=</td>
<td>equalTo, notEqualTo</td>
<td>L to R</td>
</tr>
<tr>
<td>9</td>
<td>&amp;</td>
<td>bitwise/boolean AND</td>
<td>L to R</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
<td>bitwise/boolean XOR</td>
<td>L to R</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>bitwise/boolean OR</td>
</tr>
<tr>
<td>12</td>
<td>&amp;&amp;</td>
<td>logical AND (short circuits)</td>
<td>L to R</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>?:</td>
<td>Conditional operator (boolean statement)?(expression if true):(expression if false)</td>
<td>R to L</td>
</tr>
<tr>
<td>15</td>
<td>=, +=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, &gt;&gt;&gt;&gt;&gt;, &amp;=, ^=,</td>
<td>=</td>
<td>assignment, plusAssgn, minusAssgn, timesAssgn, dividesAssgn, moduloAssgn, leftShiftAssgn, rightShiftAssgn, rightShiftAssgnUnsigned, ANDAssgn, XORAssgn, ORAssgn</td>
</tr>
</tbody>
</table>
Dealing with files

File I/O
Buffering
Exceptions
Introduction to Files

- A **computer file** is a block of arbitrary information or resource for storing information
  - Available to a computer program
  - Based on some kind of durable storage
  - Given some identifier for future ease of reference (a **filename**)
  - Avoids having to reenter or print out tedious data
- A file is just a sequence of binary digits.
  - These bits may represent integer values, text characters, or anything else.
  - The program using the file must “understand” the layout of the information to present it to the user as a text document, image, song, or program.
    - Text file: ASCII/UNICODE characters
    - Binary file: Pretty much everything else
The Operating System handles all file operations for programs
- The OS knows how to interact with the durable storage device
- The OS has policies on who/what/when can access a file
- File properties are handled by the OS
  - Size, normally expressed in bytes
  - Permissions
  - Name

The Operating System (OS) must **open** files for programs
- Programs use method calls to invoke OS routines
- New files can be created and opened to **write** (output)
- Existing files can be opened to **read** (input)
- Existing files can be opened to write/append (output)
- Existing files can be destroyed and opened to write/clobber (output)

If the OS routine fails it complains by generating/throwing an **exception**
File Buffers

- A program invokes a method to ask the OS to open the file.
  - The OS creates a memory area (a buffer) that the program has access to.
  - The OS gives the program a reference to this area (a file handle)
  - OS policy decides if a program will be provided a file handle
    - Security/permissions
    - What if someone attempts to open a file that is already opened?

- The use of buffers improves performance
  - Memory is much faster than durable storage devices
  - If a file is opened to read, the OS copies the contents of the file into the buffer
  - If a file write occurs, the change occurs in the write buffer.
  - Eventually, the OS copies the information from the write buffer into the file on the durable storage device.
File Buffers

- After data is read/written from the file, the OS must **flush** the buffer and **close** the file.
  - All open files will be closed when the program exits
  - You should always explicitly close every file that you open
  - In the future, your routine may NOT be the “main” routine!
File pointers

- Open files have a “pointer” which indicates where the next read or write operation will take place.
- A file is treated as a sequential one-dimentional sequence of characters.
- Recall: Line breaks are represented by non-printing characters.
- The **Read position** indicates what characters will be returned on the next read operation.
  - The Read position is updated/moved to the first character following the last character read.
  - Ex: 1501 245
  - If 1501 was read with nextInt(), then the read pointer would be on the space following the digit 1.
  - If the next read was via nextLine(), the returned String would be “245”;
- Most languages also provide method calls to move the Read pointer.
Filenames

- Filenames are specified differently in each OS (policy decision)
- Filenames can be specified from a default working directory (relative reference)
  - Netbeans – Set in project properties
    ```java
    String filename = "Data.txt";
    ```
- Filenames can be specified from a root directory (absolute reference)
  - C:\Documents and Settings\w001ted\Desktop\Data.txt
    ```java
    String filename = "C:\Documents and Settings" + "\w001ted\Desktop\Data.txt";
    ```
- Pitfall: Windows evolved from DOS, which used the ‘\’ character to indicate directory structure. This is the escape sequence in ASCII/UNICODE. Thus we must use `\`
- UNIX OSs use a forward slash “/”
import java.io.FileWriter;
import java.io.PrintWriter;
public class Main {
    public static void main (String[] args) throws Exception {
        final boolean APPEND_EXISTING_FILE = true;
        final boolean CLOBBER_EXISTING_FILE = false;
        String filename = "Data.txt";  // uses working directory
        String filename2 = "C:\Documents and Settings" + "\\w001ted\\Desktop\\Data.txt ";
        FileWriter fileHandle =
            new FileWriter(filename,CLOBBER_EXISTING_FILE);
        PrintWriter outputFile =
            new PrintWriter(fileHandle);
        outputFile.println("Hello file!");
        outputFile.close();
    } // end method main
} // end class Main
The PrintWriter Class

- The PrintWriter class allows you to write data to a file using the print and println methods, as we have used to output to the console.
- The constructor for PrintWriter can take a filename (String) or a filehandle (FileWriter).
  - WARNING: If you give PrintWriter a filename, it will overwrite (destroy) any existing file of that name.

```
import java.io.PrintWriter;

PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.println("Chris");
outputFile.println("Kathryn");
outputFile.println("Jean");
outputFile.close();
```

Open the file.

Write data to the file.

Close the file.
The FileWriter class

- The FileWriter class is in the java.io library
  
  ```java
  import java.io.FileWriter;
  -or-
  import java.io.*;  // import all java.io classes
  ```

- To avoid erasing a file that already exists:
  - Create a FileWriter object using an optional boolean argument that tells the object to append data to the file.
    ```java
    FileWriter fwriter = new FileWriter("filename", true);
    ```

- The buffer will be created in such a manner that any output will be appended to the end of the existing file.
- The FileWriter object can be passed to PrintWriter
Exceptions

- **Exception**: When something unexpected happens during execution
  - The program is informed (by the OS) so that it can respond appropriately
    - Common File I/O exceptions include: Can not create file (lack of permissions, disk space, etc), Can not find file (file does not exist)
  - The method making the OS method call that generated the exception must either handle the exception or pass it up the line.
    - Default action is usually to halt the program (**crash**)
    - Handling the exception will be discussed later.
- To pass it up the line, add a **throws** clause in the method header.
  
  ```java
  public static void main (String[] args) throws Exception {
  ```
- File I/O is a **checked exception**
  - the exception **must** be handled or passed up
- A method with file I/O will generate a compile-time error if the exception is not handled or passed up.
import java.io.File;
import java.util.Scanner;
public class Main {
    public static void main (String[] args) throws Exception {
        String filename = "Data.txt";  // uses working directory
        File fileHandle = new File(filename);
        Scanner inputFile = new Scanner (fileHandle);

        String line;
        while (inputFile.hasNextLine()) {
            line = inputFile.nextLine();
            System.out.println(line);
        }
        inputFile.close();
    } // end method main
} // end class Main
The File class

- The File class is in the java.io library
  
  ```java
  import java.io.File;
  ```

- Use the File class create a file handle
  
  ```java
  File fileHandle = new File ("filename");
  ```

- A Scanner object can then be made to parse the associated buffer
  
  ```java
  Scanner inputFile = new Scanner (fileHandle);
  ```

- Once an instance of Scanner is created, data can be read using the same methods that you have used to read keyboard input (nextLine, nextInt, nextDouble, etc).

- Use the Scanner predicate methods (hasNextLine(), hasNextInt(), hasNextDouble(), etc) to test for the existence of data and the end of file.
If you attempt to create a Scanner with a file that doesn’t exist:

```java
Exception in thread "main" java.io.FileNotFoundException: Data.txt (The system cannot find the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:106)
  at java.util.Scanner.<init>(Scanner.java:636)
  at fileio.Main.main(Main.java:21)
```

- The File class has predicate methods for doing basic sanity checking:
  - `exists()`, `canWrite()`, `canRead()`, `canExecute()`
- The File class has other useful methods for performing common tasks:
  - `createNewFile()`, `delete()`, `getPath()`, `getAbsolutePath()`
- The File class can also set OS properties:
  - `setExecutable()`, `setReadable()`, `setWritable()`, `setLastModificationDate()`
Using multiple Scanner objects

```java
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();
File file = new File(filename);
Scanner inputFile = new Scanner(file);
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file
Handling exceptions

- A program indicates that it will handle an exception (rather than throw it up to the next level) by surrounding one or more potentially exception generating methods with a *try block*.
- If a statement in a *Try blocks* generates an exception, then the try block’s sequential execution halts and it immediately executes the code in a following *catch block*.
- Multiple catch blocks may be used if behavior differs by Exception type.
- A try statement may (optionally) be followed by a *finally block* that is executed after statements in the try block and/or any statements in the catch block (if an exception occurs)
  - Statements in the finally block are executed even if an exception does not occur.
Try/Catch/Finally statements

```java
try {
    // statements that can cause exceptions
}
catch (Exception e) {
    // what to do if an exception happens
}
finally {
    // what to do no matter what happens above
}
```
public static void main (String[] args) {
    try {
        Scanner input =
            new Scanner (new File ("Data.dat"));
        // more code here
    } catch (Exception e) {
        System.out.println ("Error: " + e);
        // recovery code here
        // System.exit(2);
    }
    finally {
        System.out.println("Program complete.");
        System.exit (0);
    }
    System.exit (1);
} // end method main
Exercise

- Create a text file named Inventory.txt that contains the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Inventory</th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprocket</td>
<td>10</td>
<td>$1.45</td>
<td>$14.50</td>
</tr>
<tr>
<td>Widget</td>
<td>3</td>
<td>$3.00</td>
<td>$9.00</td>
</tr>
</tbody>
</table>

- Write a program that produces an output file Accounting.txt that contains the name, number, cost, and total value of each item as well as the total value of the inventory.

<table>
<thead>
<tr>
<th>Item name</th>
<th>Inventory</th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprocket</td>
<td>10</td>
<td>$1.45</td>
<td>$14.50</td>
</tr>
<tr>
<td>Widget</td>
<td>3</td>
<td>$3.00</td>
<td>$9.00</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>$23.50</td>
</tr>
</tbody>
</table>
Introduction to arrays

The array data structure
Array subscript notation
Creating new array objects
Array usage style
Arrays: simple data structures (Semantics)

- Fundamental data types:
  - Integers, Floating-point numbers, Characters and Strings, Boolean variables

- Data structures contain groups of related items
  - Thus data structures are containers or “objects”

- Fundamental data structure: the array (or list)
  - Arrays are the simplest and most common type of data structure.
  - In some languages arrays are the only data structure

- An array contains a group of items that are all of the same type and that are directly indexed using their order in this list.
  - The elements of the array can be fundamental data types or derived types (including other arrays)

- We’ve already seen this concept in String (an “array” of char)
Creating Arrays (Syntax)

- An array is an object so it needs an object reference.
- This reference can be stored in a variable.
  ```java
  int[] testScores;
  ```
- Array objects must be created to be a certain (static) size.
  ```java
  testScores = new int[6];
  ```
- Combined declaration/initializer
  ```java
  int[] testScores = new int[6];
  ```
- New array elements values are initialized to 0 in Java

<table>
<thead>
<tr>
<th>reference</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>testScores</td>
<td>index 0</td>
<td>index 1</td>
<td>index 2</td>
<td>index 3</td>
<td>index 4</td>
<td>index 5</td>
<td></td>
</tr>
</tbody>
</table>

- Array indices start at 0 and are of type int
  - Using an “out of bounds” index causes an error (exception)
Accessing Arrays (Syntax)

- An *element* of an array is accessed by the array name and an index in [ ]
  
  ```
  testScores[0] = 95;
  testScores[1] = testScores[0] - 5;
  int currentTest = 2;
  testScores[currentTest] = 75;
  ```

- Array elements can be treated as any other variable of the element type.
- Array subscripts can be accessed using integer literals or variables.
Array Initialization (Syntax)

- By default, array elements are initialized to 0 in Java
- An initialization list can be used to initialize the array to values known at implementation time
  ```java
  int[] daysPerMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  ```
- The numbers in the list are stored in the array in order:
  - daysPerMonth[0] is assigned 31,
  - daysPerMonth[1] is assigned 28,
  - etc.
- An array variable contains a reference to an array object!
  What does this initializer do?
  ```java
  int[] tempSwap = daysPerMonth;
  ```
Arrays are objects

```java
String[] monthNames = {"Jan", "Feb", "Mar"};
```

- Objects hold data (fields) and methods
- In addition to your data, all arrays in Java provide some functionality
- The length of an array can be obtained via its length field.
  ```java
  int size = monthNames.length;   // 3
  ```
- Useful array method include toString(), clone(), equals()
- Also note that arrays can hold objects and array elements that are objects have access to all of the normal methods for that object
  ```java
  String monthNameInCaps = monthName[0].toUpperCase;
  ```
Bounds Checking

- Array indexes always start at zero and continue to (array length - 1).
  
  ```java
  int values = new int[10];
  ```

- This array would have indexes 0 through 9.

- Java throws an *exception* if you access an array outside of its legal indices
  - Not all programming languages perform bounds checking!
  - What does this imply?
Pitfall: Off-by-One Error

- It is very easy to be off by one when accessing arrays.
  ```java
  // This code has an off-by-one error.
  int[] numbers = new int[100];
  for (int i = 1; i <= 100; i++){
      numbers[i] = i;
  }
  ```

- Here, the equal sign allows the loop to continue on to index 100, where 99 is the last index in the array.
- This code would throw an `ArrayIndexOutOfBoundsException`.
Some of the brightest people in Computer Science suggest that arrays should never be accessed randomly but only sequentially
- Random accesses in arrays tend to be undisciplined, error prone, and hard to prove correct.
- They suggest other (more advanced) data structures for random access.

The enhanced for loop (a type of for-each loop) provides a means to implement this stylistic constraint.

```
For (datatype elementVariable : array){ statements;}
```

Compare:

```java
double[] numberList = new double[20];
...
for(int i=0; i< numberList.length; i++) {
   System.out.println(numberList[i]);
}
```

```java
double[] numberList = new double[20];
...
for( double number : numberList) {
   System.out.println(number);
}
```
Arrays can be declared using two different syntax styles

```java
song[] songList, albumList, playList;
song songList[], albumList[], playList[];
```

- The first usage is preferred as it more accurately indicates the variable type (ex: the variable songList is an array of type song)
- `int[]`, `double[]`, `String[]`, `objectName[]` can also be used as method argument and return types

**Best practices:**

```java
int MAX_NUM_OF_SONGS = 700;
song[] songList = new song[MAX_NUM_OF_SONGS];
```

**Arrays elements are usually accessed sequentially (in order) with a loop**

- `i`, `j`, and `k` are conventionally accepted names for loop index variables when the index has no domain specific context
int numWidgets; // Get initial inventory
int numCogs;
int numSprockets;

System.out.println ("Initial Inventory");
System.out.println ("----------------");
System.out.println ("Please enter initial number of Widgets");
numWidgets = keyboard.nextInt();
System.out.println ("Please enter initial number of Cogs");
numCogs = keyboard.nextInt();
System.out.println ("Please enter initial number of Sprockets");
numSprockets = keyboard.nextInt();

Scanner keyboard =
   new Scanner
   (System.in);

// Process inventory, 1 Cog + 1 Sprocket become a Widget
while ( numCogs >= 1) && (numSprockets >= 1) {
   numCogs--;
   numSprockets--;
   numWidgets++;
}

// Output final inventory
System.out.println ("Final Inventory");
System.out.println ("----------------");
System.out.println ("Number of Widgets: " + numWidgets );
System.out.println ("Number of Cogs: " + numCogs );
System.out.println ("Number of Sprockets: " + numSprockets );
Style: Usage of an array instead of multiple name variables

```java
int[] inventory = new int[3];

Scanner keyboard = new Scanner(System.in);

// Get initial inventory
System.out.println("Initial Inventory");
System.out.println("----------------

for (int i = 0; i <= 2; i++) {
    System.out.println("Please enter initial " +
                        "number of items of type " +
                        i + ": ");
    inventory[i] = keyboard.nextInt();
}

// Process inventory, 1 Cog + 1 Sprocket become a Widget
while ( (inventory[1] >= 1) &&
        (inventory[2] >= 1) ) {
    inventory[1]--;
    inventory[2]--;
    inventory[0]++;
}

// Output final inventory
System.out.println("Final Inventory");
System.out.println("----------------

for (int i = 0; i <= 2; i++) {
    System.out.print("Number of items of type " +
                     i + ": ");
    System.out.println(inventory[i]);
}
```
Style: Usage of named indices

```java
final int FIRST_ITEM = 0; // Get initial inventory
final int WIDGET = 0;
final int COG = 1;
final int SPROCKET = 2;
final int LAST_ITEM = 2;
String[] itemName = {
    "Widget", "Cog", "Sprocket"};
int[] inventory =
    new int [LAST_ITEM+1];

Scanner keyboard =
    new Scanner
    (System.in);

// Get initial inventory
System.out.println ();
System.out.println ("Initial Inventory");
System.out.println ("-----------------");
for (int i = FIRST_ITEM; i <= LAST_ITEM; i++) {
    System.out.println ("Please enter initial " +
        "number of " + itemName[i] +"s: ");
    inventory[i] = keyboard.nextInt ();
}

// Process inventory, 1 Cog + 1 Sprocket become a Widget
while ( (inventory[COG] >= 1) &&
    (inventory[SPROCKET] >= 1) ) {  
    inventory[COG]--;
    inventory[SPROCKET]--;
    inventory[WIDGET]++;
}

// Output final inventory
System.out.println ("Final Inventory");
System.out.println ("---------------");
for (int i = FIRST_ITEM; i <= LAST_ITEM; i++) {
    System.out.print ("Number of " + itemName[i] +"s: ");
    System.out.println (inventory[i]);
}
```
Exercises

- Calculate maximum (minimum) value in an array
- Sum the values in an array
- Average the values in an array
- Find (sequentially search for) the first location of a given value in an array
Objects and References

Arrays as objects
Object references/pointers
Garbage collection
Copying objects
Comparing objects
Call by reference
Multidimensional arrays
Arrays are objects

String[] monthNames = {"Jan", "Feb", "Mar"};

- Objects hold data (fields) and methods
- In addition to your data, all arrays in Java provide some functionality
- The length of an array can be obtained via its length field.
  ```java
  int size = monthNames.length;  // 3
  ```
- Useful array method include toString(), clone(), equals()

- Also note that arrays can hold objects and array elements that are objects have access to all of the normal methods for that object
  ```java
  String monthNameInCaps = monthName[0].toUpperCase;
  ```
Assigning object references

- A variable can be assigned to any object of the appropriate type.
  ```java
  // Create an array referenced numbers.
  int[] numbers = new int[10];
  int[] digits = new int[500];
  ...
  numbers = digits;
  
  // Diagram
  numbers [reference]——> [object]
  digits [reference]——> [object]
  ```

- Both variables reference the same object (NOT a copy)
- If an object has no references then the object is lost
  - it is “garbage” and will be “collected”
Copying objects

- You cannot copy an object by merely assigning one reference variable to another.
- For an array, you need to copy the individual elements of one array to another.

```java
int[] xList = {5, 10, 15, 20, 25};
int[] yList = new int[5];
for (int i=0; i < xList.length; i++)
    xList[i] = yList[i];
```
- Or

```java
xList = yList.clone();
```
- To change the size of an array you must make a new array of the appropriate size and copy the appropriate elements!
How not to test equality

- Consider:
  ```c
  int[] xList = {1, 2, 3};
  int[] yList = {1, 2, 3};
  ```

- What is \((xList == yList)\)?

- Consider
  ```c
  xList = yList;
  xList[0] = 5;
  ```
  What is?
  ```c
  xList == yList?
  yList[0]?
  ```
Testing equality

- You cannot test equality simply by determining if the references are the same!
- You need to compare the individual elements of one array to another.

```java
int[] xList = {5, 10, 15, 20, 25};
int[] yList = {5, 10, 15, 20, 25};

boolean listsEqual = (xList.length == yList.length);
for (int i=0; i < xList.length; i++) {
    if (xList[i] != yList[i]) {
        listsEqual = false;
    }
}

-OR- xList.equals(yList);
```
Passing objects as arguments

- Object references can be passed to methods like any data type.

```java
int[] numbers = {5, 10, 15};
printList(numbers);
...

public static void printList(int[] list){
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
}
} // end method printList
```

- What does this *call by reference* imply to changes made to list[0]?
public static void swapFirstTwoElements (int[] array) {
    int firstValue = array[0];
    array[0] = array[1];
    array[1] = firstValue;
} //end method swapFirstTwoElements

swapFirstTwoElements(numbers); // method call

public static int[] swapFirstTwoElements (int[] array) {
    int firstValue = array[0];
    array[0] = array[1];
    array[1] = firstValue;
    return array;
} //end method swapFirstTwoElements

numbers = swapFirstTwoElements(numbers); // method call
Arrays of references

- Array elements can contain *primitive data types* or *references* to objects.
  - We’ve seen this so far as arrays of String
    
    ```java
    String[] nameList = {"Sam", "Bobbie", "Pat", "Kim", "Teri"};
    ```
Multi-dimensional arrays

- Arrays are objects that contain references to objects.
- Thus arrays can contain references to other arrays (which can contain other arrays, ad nauseam)

```java
boolean[][] itRained = new boolean[NUM_MONTHS][NUM_DAYS]
```

![Diagram of multi-dimensional array]

itRained

Address

itRained[0]

Address

itRained[0][0]

True

itRained[0][1]

False

itRained[0][2]

False

...

itRained[11]

Address

itRained[0][30]

True
Two-Dimensional Arrays

- A two-dimensional array is an array of arrays.
- It can be thought of as having rows and columns.
- Declaring a two-dimensional array requires two sets of brackets and two size declarators (each in its own brackets)
- Each element requires two subscripts to access!

```java
double[][] scores = new double[4][4];
```

<table>
<thead>
<tr>
<th>row 0</th>
<th>row 1</th>
<th>row 2</th>
<th>row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scores[0][0]</td>
<td>Scores[0][1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scores[1][0]</td>
<td>Scores[1][1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scores[3][3]</td>
</tr>
</tbody>
</table>
Accessing elements in multi-dimensional arrays

- Operations on multi-dimensional use multiple array indices
  - Each responses with the appropriate type (for instance int or int []).
    
    ```java
    int[][] numList = { {1,2,3},          // numList[0]
                       {3,4,5},          // numList[1]
                       {4,5,6} };       // numList[2]
    
    System.out.print(numList[0][0]);    // 1
    System.out.print(numList[0][1]);    // 2
    System.out.print(numList[1][2]);    // 5
    
    System.out.print(numList[0]);       // [I@3e25a5
    System.out.print(numList[1]);       // [I@3e25a5
    System.out.print(numList[3]);       // ArrayIndexOutOfBoundsException
    ```
Iterating through array values

- Nested loops are useful tools for dealing with multi-dimensional arrays.

```java
public static void print2DList (int[][] list) {

    for (int i=0; i < list.length; i++) {
        for (int j=0; j < list[i].length; j++) {
            System.out.print (list[i][j] + " ");
        }
        System.out.println();
    }

} // end method print2DList

- Write sumRow() or sumColumn()
```
public static void main(String[] args) {
    int[][] numList;
    int[] xList = {1, 2, 3, 4};
    yList = {9, 8};
    numList = new int[3][];

    numList[0] = xList;
    numList[1] = yList;
    xList = yList;

    printList (numList[0]);
    printList (numList[1]);
    printList (xList);
    printList (yList);
    printList (numList[2]);
} // end method main

public static void printList (int[] list) {
    for (int i=0; i < list.length; i++) {
        System.out.print (list[i] + " ");
    }
    System.out.println();
} // end method printList
Variable-length argument lists

- What if you wish to make a method that calculates the average for any number of parameters.
  
  ```java
  x = mean(10);  
  -or-  x = mean(10,15);  
  -or-  x = mean(10,15,45,2);
  
  Many contemporary programming languages provide a mechanism for variable-length argument lists (vararg parameters)
  ```

  ```java
  public static double mean(int... numList) {
  double sum = 0.0;
  for (int i = 0; i < numList.length; i++) {
    sum += numList[i];
  }
  return (sum/numList.length);
  }
  ```

- The argument numList is treated as an array of integers
Command-line arguments

- The main method header
  
  ```java
  public static void main (String[] args)
  ```

- The array that is passed into the args parameter comes from the operating system command-line.

- `java program -f test.txt`
  - `arg[0] = "-f"`
  - `arg[1] = "test.txt"`

- In Netbeans, you can set command line arguments in project/properties
Style: Parallel Arrays Vs. Objects

- Don’t create parallel arrays (two arrays that contain values “in parallel”):
  ```java
  String[] nameList; // Inventory item name
  int[] numList;    // Number of item in inventory
  double[] costList; // Cost per item
  ```

- By parallel we mean that `nameList[0]`, `numList[0]`, and `costList[0]` all contain data for the same inventory item.

- Instead, use objects to group related data items (CS 241!)
  ```java
  public class InventoryItem {
      String name;
      int number;
      double cost;
      // ...
  } // end class Inventory Item
  ```

```java
InventoryItem[] inventoryList;
```
Array of Objects Diagram Revisited

inventoryList

```
InventoryItem[]

  name = "Widget"
  number = 10
  cost = 10.20

  name = "Cog"
  number = 123
  cost = 99.99

  etc
```
Exercises

- Create a copy of a 2-dimensional array
- Sort a 1-D and 2-D arrays using selection sort
  - Search for the minimum value in the data structure
  - Swap this value with the value in the “lowest” unsorted position
  - Move to the next position and repeat

- Selection/“Bubble” sort
  - 10 5 2 7 search for position 0, min value 2 swap with 10
  - 2 5 10 7 search for position 1, min value 5, swap with 5
  - 2 5 10 7 search for position 2, min value 7, swap with 10
  - 2 5 7 10 final position reached.
Using objects / simple data structures

Operations on array data
  Wrapper classes
  The ArrayList class
  Hashes
Copying array elements

- Suppose we want to move a section of an array:
  - To make room for a new element
  - To delete an existing element
  - Or copy just a section to a new array
- We could write the code ourselves, using a for loop
- Instead, we can use a method in the System class:

  System.arraycopy( fromList, fromStart, toList, toStart, count );

array name  index
Growing an array

- Suppose your array is full, but you need room for one more element
- 1.  Create a new, larger array
   - int[] tempList = new int[origList.length + 1];
- 2.  Copy existing elements to this new array:
   - System.arraycopy( origList, 0, tempList, 0, origList.length );
- 3.  Make the old variable refer to the new data
   - origList = tempList;
Array Lists

- The ArrayList class in the Java API is similar to an array, but it does not store primitive data types as elements.
  - ArrayList data structures store any object using a generic data type
  - An ArrayList can hold objects of different types!
- ArrayList data structures provide the following abstractions/features:
  - Add an element: The ArrayList object automatically expands as items are added to it
  - Remove an element: The ArrayList object automatically reduces as items are removed from it

```java
import java.util.ArrayList;               //…
        ArrayList nameList = new ArrayList(); // NOTE NO SIZE
nameList.add( new String( "Bob" ) );      // ADDING OBJECTS
nameList.add( new String( "Pat" ) );
nameList.add( new String ("April") );
String name = (String) accounts.get( 0 ); // NOTE TYPECASTING!
```
ArrayList methods

- **Adding elements**
  - Method `add( object )` adds the object reference to the end
  - Method `add(index, object)` inserts an object reference before the into the specified position index
    - all subsequent objects have the indices updated accordingly
  - Method `set(index, object)` overwrites an existing object

- **Retrieving elements**
  - Method `get(index)` returns the object reference.
    - Note this is a generic object reference and must be type cast to be used

- **Removing elements**
  - Method `remove(index)` removes an object reference from the ArrayList

- **Checking size**
  - Method `size()` returns the size of the ArrayList
Traversing an ArrayList

```java
for ( int i = 0; i < nameList.size(); i++ ) {
    String name = (String) nameList.get( i );
    System.out.println( name.toUpperCase() );
}

- OR -

for ( int i = 0; i < shapeList.size(); i++ ) {
    Shape thisShape = (Shape) shapeList.get( i );
    Shape.draw();
}
```
If an ArrayList only hold objects of a single object type then it should be strongly typed.

- Strongly typed data structures help reduce errors!
- ArrayList can be used as a *parameterized class*

```java
ArrayList<String> nameList = new ArrayList<String>();
```

If an ArrayList has an expected size, provide a clue to its size by setting its initial capacity

- The size is still dynamic
- Helps the reader understand the scope of the ArrayList
- Increases memory efficiency
- Default initial capacity is 10

```java
ArrayList<String> nameList = new ArrayList<String>(50);
```
Wrapper Classes

- Java is not “completely” object-oriented.
- It has primitive data types as well as objects.
- To treat primitive types as objects, Java provides “wrappers” which creates objects instead of variables.
- There are wrapper classes for all 8 primitive types.
- All have the same name (with capital letter) fully spelled out (note Integer and Character).

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Double d = new Double( 15.78 );
System.out.println( d ); // 15.78
double dval = d.doubleValue();
Wrapping (boxing/unboxing)

// Create the ArrayList
ArrayList data = new ArrayList();
data.add( new Double( 3.14 ) ); // boxing 3.14
data.add( new Double( 2.72 ) ); // boxing 2.72

// Retrieve element 0 into a primitive type in 2 steps
Double dobj = (Double) data.get( 0 );
double dval = dobj.doubleValue(); // unboxing object

// Or, you can replace the last 2 lines above with:
double dval = ( (Double) data.get( 0 ) ).doubleValue();
With current versions of Java (5.0+), the compiler will automatically convert (box/unbox) from primitive data types to wrapped objects.

```java
// Create the ArrayList<Double>
ArrayList<Double> data = new ArrayList<Double>();

// Add elements as doubles via auto-boxing
data.add(3.14);
data.add(2.72);

// Retrieve element 0 as double via auto-unboxing
double x = data.get(0);
```
Wrapper class methods

- The wrapper classes have a number of highly useful methods for dealing with their subject data types.
- We’ve already introduced some of the wrapper class predicate methods, such as Character.isDigit(), Character.isUppercase(), etc.
- One of the most useful is are the parse methods which turn a String representation of a primitive value into the primitive value.
- Example:
  
  ```java
  String inputValue = "123.56";
  double cost = Double.parseDouble( inputValue );
  ```

- Wrapper classes are a good place to look for methods to perform “common” tasks
StringBuilder Class

- The StringBuilder class is to Strings what the ArrayList class is to Arrays
- StringBuilder is part of the default API, no import statement is needed

```java
int index;
StringBuilder name = new StringBuilder("travis");
name.append(" Doom, Ph.D.");
index = name.indexOf("Doom"); // index = 7
name.insert(index, "E.W. ");
name.replace(0,1,"T");
index = name.indexOf(" Ph.D."); // index = 16
name.delete(index, index + ", Ph.D".length () + 1);
System.out.println (name);
```
Hashmaps

- A map is a collection of keys and values
- It is much like an array, but with without the requirement of using integer indices from 0 to size-1.
- A hash is a one way to organize/implement a map

```java
import java.util.HashMap;
//...

HashMap<String, String> phoneBook =
    new HashMap<String, String>();
phoneBook.put("Travis Doom", "(937) 775-5105");
phoneBook.put("CSE Office", "(937) 775-5133");
// ...
System.out.println(phoneBook.get("Travis Doom"));
```
Tokenizing Strings

- Splitting a string into its components (tokens separated by delimiters) is a necessary step in the processing of most data.

```java
import java.util.StringTokenizer;

//...

String date = "8-8-2007-7";
StringTokenizer dateTokens =
    new StringTokenizer(date,"-" );
System.out.println(dateTokens.countTokens ( ) );
String month = dateTokens.nextToken ( );
String day = dateTokens.nextToken ( );
String year = dateTokens.nextToken ( );
day = Integer.toString (Integer.parseInt (day) + 1);
date = month + "-" + day + "-" + year;
System.out.println(date);
```
Wait, there’s more!

- In CS241 look forward to:
  - Object-oriented design
  - Dialog boxes and other GUI features
  - Use of asserts, unit testing, and other error detection features
  - Advanced debugging and the stack

  - Many, many new classes, methods, and object types that will help you deal with complexity
System.exit()

- This method call immediately ends the execution of the program with the return call 0
  - 0 implies all is well
  - Other values represent possible error codes

- Hopefully we’ve reached
  System.exit(0);