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?
    - Before modern digital computers, the most common digital machines were adding machines.
    - 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.
- **What 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.
  - 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)
  - Reverse Engineering Abstraction
    - Re-design
    - Re-specify
    - Rebuild
  - Existing System
  - Design
  - Implementation
  - Forward Engineering Refinement
    - Conceptual Requirements
    - Conceptual Design
    - Conceptual Implementation
    - Software level
    - Computer Science
    - ISA & Microarchitecture
    - Logic level
    - Computer/Elect. Engineering
  - Computer Engineering
  - CEG 320
  - CEG 260/360

---

**Levels of abstraction in digital computation**

---

**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 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 can not deal with any ambiguity, thus…

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
- \(10^{30}\) Total number of particles in the universe
- \(10^{100} \approx\) Number of possible solutions to traveling salesman (100)
- Travelling salesman (100) is computable but it is NOT feasible.

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.”

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?

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).

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 Problem**
- 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.
- Example: IA-32 (Intel, AMD, and others), PowerPC (Motorola).

**Algorithm & Language**
- Programs are translated from high languages in to 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.

**ISA & Microarchitecture**
- Machine-Level Equivalent
- High-Level Language – Java/C++
  - A = B + C;

**Circuits & Devices**
- 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.)

### Compilation vs. Interpretation

**Different ways of translating high-level language**
- **Interpretation**
  - compiler – 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 executable code may be different
  - languages: C++, C, Fortran, Pascal, etc.

**The Problem**
- 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
- Program is executed by the processing unit

**Processing Unit**
- ALU (Arithmetic Logic Unit)
- TEMP (Temporarily holds data)

**Control Unit**
- Carries out the instructions

**Input (keyboard)**
- ES[EA00] = ES[EA08] + ES[EA10]

**Output (monitor)**
- Print Z to screen.

### The Instruction Set

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

**Memory-Transfer Equivalent**
- \( \text{Mem}(A) = \text{Mem}(B) + \text{Mem}(C) \)
- \( \text{Mem}(\text{EA08}) = \text{Mem}(\text{EA08}) + \text{Mem}(\text{EA10}) \)

**Machine-Level Equivalent**
- Assembly (human readable)
  - Load R2, B 12EA08 1110 0010 1110 1010 0000 0000
  - Load R2, C 12EA10 1110 0010 1110 1010 0000 0000
  - R2 ← R2 + R3 0221 0000 0010 0000 0011
  - Store A, R2 22EA08 1110 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 Problem**
- The next step is to transform the ISA into a 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 Problem**
- 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
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
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

- Semantically – 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
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
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.
Primitive data types are built into a programming language and have literals represent an unchanging (non variable or constant) value.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.out.println(&quot;value&quot;);</td>
</tr>
<tr>
<td>System.out.print(&quot;The value is &quot;);</td>
</tr>
</tbody>
</table>

Java has 8 primitive data types:

- byte
- short
- int
- long
- float
- double
- boolean
- char

Variables in summary

<table>
<thead>
<tr>
<th>Variable declaration.</th>
<th>Assignment statement.</th>
<th>Initialized declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value;</td>
<td>value = 5;</td>
<td>int value = 5;</td>
</tr>
<tr>
<td>0x000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0x001</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0x002</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0x003</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

This is a String literal. It will be printed as "5".

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:
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Modulus

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Type</th>
<th>Example</th>
<th>Operator</th>
<th>Precedence</th>
<th>Example</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>numeric</td>
<td>value + value</td>
<td>-</td>
<td>Subtract</td>
<td>-value - value</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>numeric</td>
<td>value / value</td>
<td>%</td>
<td>Modulus</td>
<td>value % value</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>numeric</td>
<td>value * value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>numeric</td>
<td>value - value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>numeric</td>
<td>value % value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```
public class Reciprocal {
    /**
     * This program outputs f(x) = 1/x for x = 5
     */
    public static void main (String[] args) {
        // code goes here
    }
}
```

Program design and control

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);
  }

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

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");

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).
  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.

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:

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:
    import java.util.Scanner;
  - Scanner objects work with System.in
  - To create a Scanner object:
    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
  - Use breakpoints to halt execution at specific points
  - Examine the contents of variables and memory locations
  - Execute instructions one at a time, or in small groups
  - Step into followed by 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

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

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

Encoding

- 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.
- library stuff
- 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
  - Even carries through in scientific notation
  - Hard to implement electronically
  - ENIAC (First electronic computer) used 10 vacuum tubes/digital
  - Need high precision to encode 10 signals levels on single wire
  - Messy to implement digital logic functions
  - Reliably transmitted on noisy and inaccurate wires

- Base 2 Number Representation
  - Bit: Binary digit
  - Easy to represent/store with electric current/bistable elements
  - Reliably transmitted on noisy and inaccurate wires
  - Natural representation for human transactions
  - Hard to implement electronically
  - Even carries through in scientific notation

Commonly implemented with random-access memory (RAM)
- Memory contains:
  - Currently running programs
  - Data used by those programs
  - Memory is divided into address boundaries
  - Each address contains:
    - A group of bits (either 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
- Main memory can be visualized as a column or row of cells.

Unsigned Binary Integers

Y = "abc" = a·2^2 + b·2^1 + c·2^0
(where the digits a, b, c can each take on the values of 0 or 1 only)

N = number of bits
Range is: 0 ≤ i < 2^N - 1
Umin = 0
Umax = 2^N - 1

Problems:
- How do we represent negative numbers?

Signed Magnitude

Y = "abc" = (-1)^c·(b·2^1 + c·2^0)

Range is: -2^N - 1 ≤ i < 2^N - 1
Smin = -2^N - 1
Smax = 2^N - 1

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

Two’s Complement

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

Range is: -2^N - 1 ≤ i < 2^N - 1
Tmin = -2^N - 1
Tmax = 2^N - 1

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 iterating 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
  - 1101_2 13
  + 1011 + 5
  1000 = 8 (as expected)
- What if the value of the answer cannot 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

```
int x = foo();
int y = bar();
int z = foorbar();
```

What if x, y, and z are of floating point numbers?

```
1 (x*3 == 1) // true (Not 1 6 not 2 but 1!)
```

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 ~± 2 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 as 31 bits of precision, or roughly 9 digits proficient 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. 3.45678 = 3.10^1 + 4.10^-1 + 5.10^-2 + 6.10^-3 + 7.10^-4 + 8.10^-5
  - 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. 2.5710^-12 = 1.010^3 + 2.10^2 + 5.10^1 + 2.10^0 = 1.001101 or 1.000110 x 2^n

As a literal 2.75E1 (E-notation)

IEEE-745 fp numbers

- Double precision (64 bit) floating point
  - 52 bits: mantissa of 52 bits + 1 => approx. 15 digits decimal
  - 11 bits: biased exp.
  - 1 bit: sign

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

Range & Precision:

- 52 bit:
  - ±1.2 x 10^-38 to ±1.2 x 10^38 with 15 digits of accuracy
- 23 bit:
  - ±1.2 x 10^-9 to ±1.2 x 10^9 with 7 digits of accuracy
- 8 bit:
  - ±1.2 x 10^-3 to ±1.2 x 10^3 with 4 digits of accuracy

Dangers of abstraction!

- 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!

Java automatically converts lower precision types

```
int z = foobar();
int y = bar();
```

Previously Integer data types in JAVA

```
byte 1 byte Integers in the range -128 to +127
short 2 bytes Integers in the range -32,768 to +32,767
int 4 bytes Integers in the range -2,147,483,648 to +2,147,483,647
long 8 bytes Integers in the range -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float 4 bytes Floating-point numbers in the range ±3.4028234663852886e+38 with 7 digits of accuracy
double 8 bytes Floating-point numbers in the range ±1.70141183460469231e+308, with 15 digits of accuracy
```

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
  - (int x = 5j; double d = 5.0j;
    - x / 3 = 1j // true (Not 1.6, not 2, but 1!)
    - 1d / 3 = 0j // false (mixed operands, result promoted)
    - 1d (double) / 3 = 1j // 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 = (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 as 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)
- Escape sequences used for special characters
- Java supports the primitive data type char

Escape sequences

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

Boolean data types

- Two valued: true or false
- The boolean type is often used to score the result of a conditional test
- The boolean literals are the keywords true and false (no quotes).
- A predicate method is one that returns a boolean value
- Each value is ordered (ordinal value) from 0 to last value.
- A boolean variable has a value that is one of the two truth values true or false.
- A boolean variable can be treated as a number: true is 1 and false is 0,
  or treated as such.

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 +,
  - String objects are built automatically
- String objects come with useful methods like length(), toLowerCase(), and charAt()
- Use the special string constants "Hello" and "word" to illustrate string concatenation:

```java
String message1 = "Hello" + " " + "world";
String message2 = "Goodbye";
char letter = message2.charAt(0);
```

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,…”
  - enum { 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.
  - enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRI
    DAY, SATURDAY};
- String is such a useful data type that it is built into the language (java.lang)
  - String has its own context for +,
  - String objects are built automatically
- String objects come with useful methods like length(), toLowerCase(), and charAt()
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
  - 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

Scoping Example

```java
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);
    }
}
```

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)
- 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
- 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;
}
```

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: Typified 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
    - Definitions functionally (source code) and return 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);
    }

The Method call

- A void method is one that simply performs a task and then terminates.
  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.
  int number = keyboard.nextInt();
- Values that are sent into a method are called arguments.
  System.out.println("Hello");
  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
int number = keyboard.nextInt();
System.out.println("Hello");
length = Math.sqrt(area);

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

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

The argument 5 is copied into the num1 parameter.

showSum(5,10);
The argument 10 is copied into the num2 parameter.

NOTE: Order matters!

```java
public static void showSum(double num1, double num2) {
    double sum; // to hold the sum
    sum = num1 + num2;
    System.out.println("The sum is "+ sum);
}
```
Arguments? Parameters?

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

Writing your own method

```java
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

```
<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller</td>
<td>Caller</td>
<td>Caller</td>
</tr>
</tbody>
</table>
```
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
  - 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).

- `floor(x)`, `ceil(x)`, `round(x)`
- `sin(x)`, `cos(x)`, `tan(x)`
- `atan(x)`
- `i()`
- `85`

Example: Calculate the area of a circle
- `double y = 3.14 * x * x;`
- `double area = Math.PI * Math.pow(radius,2);`

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"
- `String word = "dogma";`
- `char letter = word.charAt(2);`

- `length()`, `charAt()`, `toUpperCase()`, `toLowerCase()`, `compareTo()`, `compareToIgnoreCase()`, `concat()`, `endsWith()`, `Equals()`, `equalsIgnoreCase()`, `getChars()`, `indexOf()`, `lastIndexOf()`, `regionMatches()`, `Replace()`, `Split()`, `startsWith()`, `Substring()`, `Trim()`, `valueOf()` ...

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

I/O methods

- We've seen the Scanner class for input
  - `byte nextByte()`
  - `double nextDouble()`
  - `float nextFloat`
- We've seen the System class for output
  - `system.out.println( value )`
  - `println (int)`
  - `println (double)`
  - `nextInt()`
  - `string nextLine()`
  - `long nextLong()`
  - `short nextShort()`
- `system.out.print (value)` implemented as `print (value);`
- `println (value);`

System.out.printf() (format, v1, v2, .. vn)
- `System.out.printf("UID=%d\n", uid, loginName);`
- `printf("%+-5d  %+5d", 10, -20)`
- `printf("%8.3f  %7.2f",1.234567,1.234567)`

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;
  ...

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

Selection control: if statements

if (BooleanExpression) {
  statement1;
  else {
  statement2;
  }
}
if (BooleanExpression) {
  block of statements;
  }
else {
  block of statements;
  }

Nested ifs

if (BooleanExpression1) {
  if (BooleanExpression2) {
  block of statements;
  }
  else {
  block of statements1;
  }
  
  if (BooleanExpression3) {
  block of statements2;
  }
  
  else {
  block of statements3;
  }
  }

if (student.InCourse(CS240) ) {
  if (date < lab1.dueDate()) {
    lab1.allowTurnIn();
  } else { // lab is late
    printErrorMessage("This lab will" +
    "not be accepted late.");
  } 
  else { // student is not in CS240
    printErrorMessage("Only CS240 students" +
    "may turn in this lab.");
  }

  
  if (ageInYears < 0) {
    handleError();
  } else {
    employee.age = ageInYears;
  }
}
Predicate logic and logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndAlso</td>
<td>Connects two boolean expressions into one. Both expressions must be true for the overall expression to be true.</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>Connects two boolean expressions into one. Either one or both expressions must be true for the overall expression to be true.</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>The '!' operator reverses the truth of a boolean expression.</td>
<td></td>
</tr>
</tbody>
</table>

### Expressions A and B

| Expression A | Expression B | A \& B | A && B | A || B | A \& \& B | A \& \& B |
|--------------|--------------|--------|-------|-------|-----------|-----------|
| false        | false        | true   | false | false | true       | false     |
| false        | true         | true   | false | false | false      | true      |
| true         | false        | false  | true  | true  | false      | true      |
| true         | true         | true   | true  | true  | true       | true      |

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.
  ```java
  highScore = {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");
}
```

Short circuiting

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

Boolean laws

- Distribution A && (B || C) == AB || AC
- DeMorgan's
  1. !(A && B) == !A || !B
  2. !(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
- 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();
}
```
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
- Cases are literal values

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

Conditional operator

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

```java
String legalClassification = (age > 18) ? "Adult" : "Child";
```

Vs.

```java
String legalClassification;
if (age > 18) {
    legalClassification = "Adult";
} else {
    legalClassification = "Child";
}
```

- Style hint: Avoid the use of this unnecessary (albeit cool) operator.

Control flow: Iteration

- Three control structures
  - Sequential
  - Selection/Conditional
  - Iteration

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

Syntax of control flow

- Sequential
- Selection
- Iterative
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

\[
\text{while (BooleanExpression) ; } \\
\quad \text{statement;} \\
\text{while (BooleanExpression) ; } \\
\quad \text{block of statements;} \\
\text{while (BooleanExpression) ; } \\
\quad \text{statement(s);} \\
\text{while (BooleanExpression) ; } \\
\quad \text{false}
\]

while statement style

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

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

pitfalls

- Off by one errors
- Accidental infinite loops
- Do not monkey with loop indices!

for loop statement

```java
for (int 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
```

- Syntax
- 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
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++) {
    for (int j = 1; j <= 5; j++) {
        if (i == j) {
            System.out.print("*");
        } else {
            System.out.print(" ");
        }
    }
    System.out.print("n");
}
```

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
- Continue statement: Skips all remaining statements and goes immediately to the next iteration of the innermost loop
- Style hints: Avoid using break/continue for flow control.

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
  - Use a user specified sentinel value
  - Test for the end of an input stream using standard sentinels
- Example: 5, 2, 4, 2, 1, 7
- 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 . . . . . . 
```

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 not use) these operators consistently!

More about operators

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

- Incrementing/Decrementing a variable by one is a very common task
  - Ex: \( x = x + 1; \), \( y = y - 1; \)
- The increment/decrement operators allow a shorthand notation
  - Ex: \( x++ \) \( y--; \)
  - \( x++ \) is a post-increment operator
  - \( y--; \) is a pre-increment operator
- \( x++ \) 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
  - \( 11309 | 798 \) \( 0010 1100 0010 1101 \)
  - Bitwise OR: |  
  - Bitwise AND: & 
  - Bitwise Exclusive OR (XOR): ^
  - Bitwise negation: ~
- Literal values for hexadecimal values
  - \( \text{int } x = 0xFF; \)
- These operators also work on boolean typed data (0 is false, 1 is true)
- Pitfall: what does \( | x \ y \) 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)
  - \( x \), \( x << 2 \), \( x \gg 2 \), \( x >>> 2 \)
  - Assume \( x \) is a 32-bit variable
    - \( x \), \( x << 2 \), \( x \gg 2 \), \( x >>> 2 \)

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)?

Example: RGB colors

- 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
- 0000 1111 0000 0000 0000 1111 is what color?
- 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>Annotation</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>prefix increment, prefix decrement</td>
<td>L to R</td>
</tr>
<tr>
<td>2</td>
<td>+ -</td>
<td>addition, subtraction</td>
<td>L to R</td>
</tr>
<tr>
<td>3</td>
<td>type</td>
<td>new type cast, object instantiation</td>
<td>R to L</td>
</tr>
<tr>
<td>4</td>
<td>instanceof</td>
<td>downcasting, type comparison</td>
<td>L to R</td>
</tr>
<tr>
<td>5</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>greaterThanOrEqual, lessThanOrEqual</td>
<td>L to R</td>
</tr>
<tr>
<td>6</td>
<td>&amp; ^</td>
<td>bitwise/boolean AND, bitwise/boolean XOR</td>
<td>L to R</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>logical OR (short circuits)</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></td>
<td>logical AND (short circuits)</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>bitwise/boolean OR</td>
<td>L to R</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>Conditional operator</td>
<td>R to L</td>
</tr>
<tr>
<td>14</td>
<td>+= -= *= /= %= &lt;&lt;= &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 Input and Output (I/O)**

- 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/clobber (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 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

**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-dimensional 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 nextInt(), the returned String would be “245”.
- Most languages also provide method calls to move the Read pointer.

Writing to a file

```java
import java.io.FileWriter;
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) throws Exception {
        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 = true;
        PrintWriter outputFile = new PrintWriter(fileHandle);
        outputFile.println("Hello file!");
        outputFile.close();  
    } // end method main
}
```

File I/O is a checked exception. When something unexpected happens during execution, the program is informed (by the OS) that it can respond appropriately.

- The FileWriter class is in the java.io library

```java
import java.io.FileWriter;
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) throws Exception {
        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
}
```

File I/O is a checked exception. When something unexpected happens during execution, the program is informed (by the OS) that it can respond appropriately.

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
    - String filename = "Data.txt";
  - Filenames can be specified from a root directory (absolute reference).
    - `\Documents and Settings\w001ted\Desktop\Data.txt`
    - String filename = "\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 OS uses a forward slash ‘/’.

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.

```java
public class Main {
    public static void main(String[] args) throws Exception {
        String filename = "Names.txt";
        PrintWriter outputFile = new PrintWriter(filename, CLOBBER_EXISTING_FILE);
        outputFile.println("Jean");
        outputFile.println("Kathryn");
        outputFile.println("Chris");
        outputFile.close();
    }
}
```

Exceptions

- Exception: When something unexpected happens during execution.
  - The program is informed (by the OS) so that it can respond appropriately.
- 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.
  - public static void main(String[] args) throws Exception {

```java
public class Main {
    public static void main(String[] args) throws Exception {
        String filename = "Names.txt";
        PrintWriter outputFile = new PrintWriter(filename, CLOBBER_EXISTING_FILE);
        outputFile.println("Jean");
        outputFile.println("Kathryn");
        outputFile.println("Chris");
        outputFile.close();
    }
}
```

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.
- 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.

```java
import java.io.FileWriter;
import java.io.PrintWriter;

public class Main {
    public static void main(String[] args) throws Exception {
        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
}
```

To pass it up the line, add a throws clause in the method header.
  - public static void main(String[] args) throws Exception {

```java
public class Main {
    public static void main(String[] args) throws Exception {
        String filename = "Names.txt";
        PrintWriter outputFile = new PrintWriter(filename, CLOBBER_EXISTING_FILE);
        outputFile.println("Jean");
        outputFile.println("Kathryn");
        outputFile.println("Chris");
        outputFile.close();
    }
}
```
Reading from a file

```
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();
    }
}
```

The File class

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

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

- A Scanner object can then be made to parse the associated buffer
  ```
  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 (nextInt, nextDouble, etc).
- Use the Scanner predicate methods (hasNextLine(), hasNextInt(), hasNextDouble(), etc) to test for the existence of data and the end of file.

File class methods

- If you attempt to create a Scanner with a file that doesn’t exist:
  ```
  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 other useful methods for performing common tasks
  - `createNewFile()`, `delete()`, `getAbsolutePath()`
  - `setExecutable()`, `setReadable()`, `setWritable()`, `setLastModificationDate();`

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.

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);
```}

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
}
```
Try/Catch/Finally statements

```java
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:
  - Item: Inventory Cost
  - Sprocket: 10 $1.45
  - Widget: 3 $3.00

- 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>

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.
- Int[] testScores;
- Array objects must be created to be a certain (static) size.
  - testScores = new int[6];
- Combined declaration/initializer
  - int[] testScores = new int[6];
- New array elements values are initialized to 0 in Java
  - 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.
Arrays are objects

- 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;
  ```

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, 30, 31, 30};
  ```
- 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!
  ```java
  int[] tempSwap = daysPerMonth;
  ```
- What does this initializer do?
  ```java
  int[] daysPerMonth = {31, 28, 31, 30, 31, 30, 31, 30, 31, 30};
  ```

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?

for / for-each loops

- 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 (a type of for-each loop) provides a means to implement this stylistic constraint.
  ```java
  for (datatype elementVariable : array){ statements;}
  ```
- Compare:
  ```java
  double[] numberList = new double[20];
  ```
  ```java
  for(int i = 0; i < numberList.length; i++) {
      System.out.println(numberList[i]);
  }
  ```

Array usage (Style)

- Arrays can be declared using two different syntax styles:
  ```java
  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;
  ```
  ```java
  song[] songList = new song[MAX_NUM_OF_SONGS];
  ```
- Arrays elements are usually accessed sequentially (in order) with a loop
  - k, and l are conventionally accepted names for loop index variables when the index has no domain specific context

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.
Style: No data structure

```java
int[] inventory = new int[LAST_ITEM+1];
Scanner keyboard = new Scanner(System.in);

// Get initial inventory
System.out.println ("Please enter initial number of Sprockets");
numSprockets = keyboard.nextInt();
System.out.println ("Please enter initial number of Cogs");
numCogs = keyboard.nextInt();
System.out.println ("Please enter initial number of Widgets");
numWidgets = keyboard.nextInt();
System.out.println ("-----------------" );
System.out.println ("Initial Inventory");
System.out.println ("Number of Sprockets: ", numSprockets );
System.out.println ("Number of Cogs: ", numCogs );
System.out.println ("Number of Widgets: ", numWidgets );
System.out.println ("---------------" );
// Process inventory, 1 Cog + 1 Sprocket become a Widget
while ( numCogs >= 1 ) && (numSprockets >= 1) {
    numWidgets++;
    numSprockets--;
    numCogs--;
}
System.out.println ("-----------------" );
System.out.println ("Final Inventory");
System.out.println ();
System.out.println ("Number of Sprockets: ", numSprockets );
System.out.println ("Number of Cogs: ", numCogs );
System.out.println ("Number of Widgets: ", numWidgets );
System.out.println ("---------------" );
```

Style: Usage of an array instead of multiple name variables

```java
int[] inventory = new int[LAST_ITEM+1];
Scanner keyboard = new Scanner(System.in);

// Get initial inventory
System.out.println ("Please enter initial number of Sprockets");
inventory[0] = keyboard.nextInt();
System.out.println ("Please enter initial number of Cogs");
inventory[1] = keyboard.nextInt();
System.out.println ("Please enter initial number of Widgets");
inventory[2] = keyboard.nextInt();
System.out.println ("-----------------" );
System.out.println ("Initial Inventory");
System.out.println ("Number of Sprockets: ", inventory[0]);
System.out.println ("Number of Cogs: ", inventory[1]);
System.out.println ("Number of Widgets: ", inventory[2]);
System.out.println ("---------------" );
// Process inventory, 1 Cog + 1 Sprocket become a Widget
while ( (inventory[0] >= 1) && (inventory[1] >= 1) ) {
    inventory[2]++;
    inventory[0]--;
    inventory[1]--;
}
System.out.println ("-----------------" );
System.out.println ("Final Inventory");
System.out.println ();
System.out.println ("Number of Sprockets: ", inventory[0]);
System.out.println ("Number of Cogs: ", inventory[1]);
System.out.println ("Number of Widgets: ", inventory[2]);
System.out.println ("---------------" );
```

Style: Usage of named indices

```java
int[] inventory = new int[LAST_ITEM+1];
Scanner keyboard = new Scanner(System.in);

// Get initial inventory
System.out.println ("Please enter initial number of Sprockets");
i = keyboard.nextInt();
System.out.println ("Please enter initial number of Cogs");
i = keyboard.nextInt();
System.out.println ("Please enter initial number of Widgets");
i = keyboard.nextInt();
System.out.println ("-----------------" );
System.out.println ("Initial Inventory");
System.out.println ("Number of Sprockets: ", inventory[0]);
System.out.println ("Number of Cogs: ", inventory[1]);
System.out.println ("Number of Widgets: ", inventory[2]);
System.out.println ("---------------" );
// Process inventory, 1 Cog + 1 Sprocket become a Widget
for (int i = FIRST_ITEM; i <= LAST_ITEM; i++) {
    inventory[i] = keyboard.nextInt();
    if ( inventory[i] >= 1 ) {
        numItemsOfType[i]++;
        numSprockets--;
        numCogs--;
    }
}
System.out.println ("-----------------" );
System.out.println ("Final Inventory");
System.out.println ();
System.out.println ("Number of Sprockets: ", inventory[0]);
System.out.println ("Number of Cogs: ", inventory[1]);
System.out.println ("Number of Widgets: ", inventory[2]);
System.out.println ("---------------" );
```

Objects and References

- Arrays are objects
- 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
- 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
  String monthNameInCaps = monthName[0].toUpperCase;

Arrays are objects

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

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

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

Call by reference
- Multidimentional arrays
- 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
- 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
  String monthNameInCaps = monthName[0].toUpperCase;

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

- Both variables reference the same object (NOT a copy)
- If an object has no references then object 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:

  ```java
  int[] xList = {1, 2, 3};
  int[] yList = {1, 2, 3};
  ```

- What is `xList == yList`?

  - Consider `xList = yList; xList == yList?` and `xList[0] = 5; 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};
  ```

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

  - OR

    ```java
    xList.equals(yList);
    ```

Passing objects as arguments

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

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

- What does this call by reference imply to changes made to list[0]?

Compare/Contrast Style

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

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];
```

Accessing elements in multi-dimensional arrays

- Operations on multi-dimensional use multiple array indices
  - Each response with the appropriate type (for instance int or int[]).

```java
int[][] numList = {{1,2,3}, {3,4,5}, {4,5,6}};
```

Iterating through array values

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

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

References/pointers in arrays

```java
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;
    // end method main
}
```
Variable-length argument lists

- What if you wish to make a method that calculates the average for any number of parameters.
  \[ x = \text{mean}(10); \]  
  \[ -or- \quad x = \text{mean}(10, 15); \]  
  \[ -or- \quad x = \text{mean}(10, 15, 45, 2); \]

- 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;
  }
  ```

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

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;
      // …
  }
  ```

- Exercise: Using objects / simple data structures
  - Create a copy of a 2-dimensional array
  - Sort a 1-D and 2-D arrays using selection sort
  - 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 10 7 final position reached.
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:
  ```java
  System.arraycopy(fromList, fromStart, toList, toStart, count);
  ```

Growing an array

- Suppose your array is full, but you need room for one more element
  1. Create a new, larger array
     ```java
     int[] tempList = new int[origList.length + 1];
     ```
  2. Copy existing elements to this new array:
     ```java
     System.arraycopy(origList, 0, tempList, 0, origList.length);
     ```
  3. Make the old variable refer to the new data
     ```java
     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 datastructures 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
- ArrayList methods provide the following abstractions/features:
  - 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
  - Removing elements
    - Method .remove( index ) removes an object reference from the ArrayList
  - Checking size
    - Method .size() returns the size of the ArrayList
  - Retrieving elements
    - Method .get ( index ) returns the object reference.
      - Note this is a generic object reference and must be type cast to be used
  - Other methods
    - Method .set( index, object ) overwrites an existing object
- If an ArrayList only hold objects of a single object type then it should be strongly typed.
- Style: ArrayLists
  - 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>();
  ```

Traversing an ArrayList

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

### Wrapper Classes

<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>

### Wrapping (boxing/unboxing)

```java
// 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
```

### Auto-boxing/Auto-unboxing

- 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, ".-" );
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

```java
System.exit(0);
```