COMP1000 Review

Lecture 1

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- Misc

COMP1000 Topics

- **Computation/Programming**
- Variables, I/O
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- Misc

What Makes Up a Computer?

- Hardware
  - Physical components
  - Wide variety of types and manufacturers
  - Abstracted to a simple set of ideas for Computer Science
- Software
  - Programs (i.e., instructions)
  - Wide variety of purposes

High Level Hardware View

Input Devices

Processor (CPU)

Main Memory (RAM)

...

Output Devices

Storage Devices

Main Memory (RAM)

Addresses	Bits	Grouping
byte 0	01110011	4 bytes at address 0
byte 1	01110100	
byte 2	01100001	
byte 3	01110010	
byte 4	01110111	2 bytes at address 4
byte 5	01100001	
byte 6	01110010	2 bytes at address 6
byte 7	01110011	
	...	

Running a Program

CPU doesn't understand Java directly

Processor (CPU)

Memory (RAM)

...

Data/Input

Output

Storage Device

Program

Compilers

Java Source Code

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

→ Java Compiler →

Byte Code

01111010000101010100
10001101000110100011
11100101010100101001
10001010110001010000
01110101010111000110
01100100001001101010
10101001000011110001
11000000011110100010
10101010100100001110…

Java Virtual Machine

- Java byte code also can't be executed by a CPU directly
- Instead, the Java Virtual Machine (JVM) is another program that interprets the byte code and translates it into the native CPU language
  - Allows a program to be compiled once and run on all types of computers (that have a JVM available and installed)
- Other high level languages work differently

Building a Java Program

Java Source Code

Processor (CPU)

Memory (RAM)

...

Storage Device

Java Compiler

Java Byte Code

Running a Java Program

Program Input → Processor (CPU) ↔ Memory (RAM) ... → Java Byte Code

Storage Device: JVM, Libraries

COMP1000 Topics

- Computation/Programming
- **Variables, I/O**
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- Misc

Variables

- Each variable has a name that the programmer uses to access and modify that variable's value
- Each variable holds exactly one value that is stored in a particular memory location
- Over time, as a program executes, the value of a variable can change

Variable Names

- In Java, variable names:
  - Must start with either a letter (uppercase or lowercase) or an underscore
  - Must contain only letters, digits, and underscores
  - Are case sensitive

- Examples: `count`, `x`, `user_input2`
- Invalid ~~names~~: 42, 5x, #yolo, file.cpp, a-b

Variable Declaration

- Every variable must be declared before you can use it in your program
- Syntax: **`TYPE NAME;`**
  - The type comes first, then then variable name, followed by a semicolon
- Examples:

```
int count;
int num_vals;
double average;
char first_initial;
```

Data Types

- There are 8 "primitive" data types in Java

  `byte`, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**

- These are all built-in types – all others hold "references" (think memory address) of an instance of some class (i.e. in object)
  - Whereas the values of primitives are themselves useful as data (e.g. true, 8, 'a'), these hold addresses to data that is actually elsewhere

Data Types (1)

- **`int`**
  - Integer, whole numbers; 4 bytes
  - Examples: 0, 15, -100464, 420712003, -1
  - Range: -231 (-2147483648) to 231-1 (2147483647)

- **`double`**
  - Numbers with fractional component (15-digit prec); 8 bytes
  - Examples: 11.23, -959.75, 0.5, -1.0
  - Range: ~10-308 to ~10308, positive or negative

- **`boolean`**
  - Only values: true, false; at least 1 byte

Data Types (2)

- **char**
  - Single character or symbol; 2 bytes
  - Examples: 'a', 'C', '3', '.', '$'
  - Actually numeric codes referring to the ASCII table (http://asciitable.com)

Data Types (3)

- **`String`**
  - A sequence of characters and/or symbols
  - Examples: "Hello World", "475!", "a", "$”
  - Non-primitive (how do we know?)
    - Useful: **`length()`**, **`charAt()`**, **`equals()`**, …
    - Concatenation: String + String

Literals

- When you type a variable value in source code, this is referred to as a “literal” – the representation of a fixed value

- The way you write the literal implicitly indicates its data type
  - `double`: decimal (`3.14`), sci. not. (`6.02e23`)
  - `char`: single quotes (`'a'`)
  - `String`: double quotes (`"hello"`)

Mixing Types

- In general, avoid or be careful
- Integers and characters interchange via the ASCII table codes
- Casting: operation that converts a value of one data type to another
  - Syntax: **`(type) value`**
  - Narrowing: larger -> smaller range

```
int x = (int) 1.7; // 1
```

  - Widening: smaller -> larger

```
double y = (double) 1 / 2; // 0.5
```

Variable Initialization

- Before you use a variable, it MUST have a value

- You can initialize a variable when you declare it or you can do so afterwards
  - Syntax after declaration: **`NAME = VALUE;`**
  - Syntax during declaration: **`TYPE NAME = VALUE;`**

- Examples

```
count = 0;
ultimate_answer = 42;
int num_vals = 10;
double pi = 3.14159;
```

Mutability

- A variable that can *change* its value is mutable
- A variable that cannot is called *immutable* or a *constant*
- Use the **`final`** keyword to create a constant - the compiler will ensure it receives exactly one value
  - By convention, use all caps for the name (e.g. **`Math.PI`**)

The Value `null`

- For any non-primitive, the **`null`** value says that it points to an invalid/non-existent object (think: bad memory address)
- This is the default value in some circumstances (e.g. member variable, array value)

Terminal Input

- Java doesn't (easily) allow reading directly from **`System.in`**

- Instead, you use a **`Scanner`** object that handles reading the input and ensures that the type of data you read matches what you want

Using a **Scanner**

- Declare and initialize a Scanner object

```
Scanner s = new Scanner(System.in);
```

  - *One per program!*
  - *Requires* **import java.util.Scanner;**

- Call methods on the **Scanner** object to read different types of values from the terminal

```
int var = s.nextInt();
double var = s.nextDouble();
String var = s.next();
String var = s.nextLine(); // whole line
```

  - Careful mixing with others due to whitespace handling

Terminal Output

- You should get into the habit of using formatted printing by default

```
System.out.printf("format", arg1, arg2, …);
String s = String.format("format", arg1, arg2, …);
```

- % in the format string followed by a converter, with optional flag(s) in between
  - Aside from new line, expects corresponding argument (1st % -> 1st arg, 2nd -> 2nd, …)

Common Converters/Flags

Converter	Flag	Description
d		An integer
f		A float (includes double)
s		A String
b		A Boolean
n		New line
	+	Includes the sign (positive or negative)
	,	Includes grouping characters
	.3	Three places after the decimal.

Quick Check

What is the output to the terminal when the following code is run?

```
String a = String.format("W%sT", "I");
String b = String.format("%d2%d", "!".length(), a.length());
System.out.printf("%s %s%n%.2f!%n", a, b, 3.14159);
```

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- **Expressions**
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- Misc

Operator Precedence

Evaluated First

( )	parentheses
*, /, %	multiplication, division, mod
+, -	addition, subtraction
=	assignment

Evaluated Last

For binary operators, if both operands are **`int`** *values, then the result is an* **`int`** *value; else* **`double`**

Quick Check: x=?

- `int x = 5/2;`
- `double x = 5/2;`
- `int x = 5.0/2;`
- `double x = 5.0/2;`
- `int x = 5/4*4;`
- `int x = 5/(4*4);`
- `double x = 5/4*4.0;`
- `int x = 5.0/4*8;`
- `double y = 5;`
  `double x = 1+y*(y/2);`

- 2
- 2.0
- Error!
- 2.5
- 4
- 0
- 4.0
- Error!
- 13.5

More Operators

- `x += a; // x = x + a`
  - Also: `-=` `*=` `/=`

- `x++; // x = x + 1`
  - Also: `--`

`Math` Class

Remember useful static methods available via the Math class

```
Math.sqrt(double)
Math.pow(double, double)
Math.abs(double)
Math.log(double)
Math.log10(double)
…
```

Boolean Expression

An expression that evaluates to a **`true`** or **`false`**

Comparison operators, returns **`true`** if…

```
!a         (a is false)
a == b     (a is equal to b)
a != b     (a is not equal to b)
a < b      (a is less than b)
a <= b     (a is less than or equal to b)
a > b      (a is greater than b)
a >= b     (a is greater than or equal to b)
```

Equality Gotcha

- Remember, equality only works for constant values/primitive variables – not objects!
- Object variables hold a reference (think memory address), so equality is asking if they are actually the same object

```
String a = "WIT";
String b = "wit".toUpperCase();
System.out.printf("%b%n", a==b);
System.out.printf("%b%n", b==a);
System.out.printf("%b%n", a==a);
System.out.printf("%b%n", b==b);
System.out.printf("%b%n", a.equals(b));
System.out.printf("%b%n", b.equals(a));
```

Complex Boolean Expressions

- Logical AND

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

- Logical OR

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Checkup

a	b	c	((a && !b) || (!a && b)) && !c
false	false	false	
false	false	true	
false	true	false	
false	true	true	
true	false	false	
true	false	true	
true	true	false	
true	true	true	

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- **Arrays**
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- Misc

Arrays

- An array is a fixed-size list of variables of the same type, that represents a set of related values
- Access many values via a single variable name
- There is special syntax to create an array and to use its elements

Declaring an Array

Similar syntax to other variables, but type is now **TYPE[]**

```
TYPE[] name;
```

Examples…

```
int[] counts;
double[] costs;
boolean[] tf;
String[] names;
```

Initializing an Array

- Must set the size of the array at initialization, but options to set initial element values

- First option: set all to **`0`**/**`false`**/**`null`** via new (remember, arrays are objects!)…

```
int[] counts = new int[5];
double[] costs = new int[4];
boolean[] tf = new boolean[2];
String[] names = new String[3];
```

Initialize an Array with Size/Values

- If at same time as declaration…

```
int[] counts = {1, 2, 3, 4, 5};
double[] costs = {1.1, 2.2, 3.3, 4.4};
boolean[] tf = {false, true};
String[] names = {"foo", "bar", "baz"};
```

- If later…

```
boolean[] tf;
…
tf = new boolean[] {false, true};
```

Array Size

- Once initialized, the array has a fixed size available via a member variable: **`length`**

- Example…

```
int[] counts = new int[5];
boolean[] tf = {false, true};

System.out.printf("%d %d%n",
    counts.length, tf.length);
```

Accessing Array Elements

- Every element within the array has an “index” (think address, relative to the beginning of the array): **`0`** – **`array.length-1`**

- After initialization, access an individual element using any expression that resolves to an integer within brackets

```
array[index expression] = value;
var = array[index expression] * 2;
System.out.printf("%d%n",
   array[index expression]);
```

- Bad index raises ArrayIndexOutOfBoundsException

```
int[] a = {5, 4, 3, 2, 1};
System.out.printf("%d", a[a.length]);
```

Checkup

```
int[] a = {5, 4, 3, 2, 1};
int x = 2;

System.out.printf("%d %d %d %d %d",
   a[3], a[x], a[x/2],
   a[x+x], a[x-2]);
```

Arrays in Memory

Arrays are stored in memory so that all the elements are sequential, in order:

```
int[] counts;
counts = new int[8];
counts[3] = 10;
```

address	value	variable
1000	0	counts[0]
1004	0	counts[1]
1008	0	counts[2]
1012	10	counts[3]
1016	0	counts[4]
1020	0	counts[5]
1024	0	counts[6]
1028	0	counts[7]
1032		
1036		
	…	

Arrays of … Arrays (Twist!)

- Same concept, but the data type of each array element is itself an array

- Example: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}};
System.out.printf("%d%n",m[2][0]);
```

Multidimensional Arrays

```
int[][] m;
m = new int[][] {{1, 2}, {3, 4}, {5, 6}};

int[][] m = new int[3][];
m[0] = new int[] {1, 2};
m[1] = new int[] {3, 4};
m[2] = new int[] {5, 6};

int[][] m = new int[3][2];
m[0][0] = 1;
m[0][1] = 2;
m[1][0] = 3;
m[1][1] = 4;
m[2][0] = 5;
m[2][1] = 6;
```

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- Arrays
- **Control Flow, Conditionals, Loops**
- Methods
- Exceptions, File I/O
- Misc

Sequential Execution

- Control flow is the order in which program statements are executed
- Remember: the program starts at **`main()`** and executes line-by-line till either **`System.exit()`** or end of **`main()`**
- However, some commands cause the execution to "hop" somewhere else
  - Conditionals, loops, exceptions

Conditional Statements

```
if (B_EXPR_A) {
    // a
} else if (B_EXPR_B) {
    // b
} else if (B_EXPR_C) {
    // c
} else {
    // d
}
// e
```

- Starts with **`if`**
  - Execute **`a`** only if **`B_EXPR_A==true`**
  - Then **`e`**
- Any number of **`else if`**
  - Tested only if **`B_EXPR_A==false`**
  - Sequentially tested:
    - **`b`** if **`B_EXPR_B==true`**
    - **`c`** if **`B_EXPR_B==false && B_EXPR_C==true`**
- Optional **`else`**
  - Only if prior **`B_EXPR_*`** all **`false`**
  - Cannot have a condition

Why is this strange?

```
if (a == true) {
    …
}
```

```
if (a) {
    …
}
```

Checkup

```
int x = 5;
if (x=5) {
  System.out.printf("foo");
}
```

Assignment Operator

The assignment operator (=) *returns* the assigned value

```
int x = 5;
System.out.printf("%b%n", x==5); // true
System.out.printf("%b%n", x==7); // false
System.out.printf("%d%n", x=5); // 5
System.out.printf("%d%n", x=7); // 7
```

Checkup

```
boolean x = false;
if (x=false) {
   System.out.printf("foo");
}
if (x=true) {
   System.out.printf("bar");
}
```

?: Operator (Ternary)

Shortcut to an “if-else” expression

```
(condition)?(result if true):(result if false)
```

```
int x=10, y;
if (x%2 == 0) {
    y = 1;
} else {
    y = 0;
}
System.out.printf("%d%n", y);
```

```
int x=10, y;
y = (x % 2 == 0)?1:0;
System.out.printf("%d%n", y);
```

`while` Loops

`while` loops are used to repeat a set of statements while some condition is **`true`**

```
int x = 1, y = 1;
while (x<100) {
   x *= 2;
   y++;
}
System.out.printf("%d %d", x, y);
```

`do-while` Loops

- A **`while`** loop body might be executed zero times if the condition is never true
- If you need to always execute the body at least once, use a **`do-while`** loop (remember the final **`;`**)

```
int x;
Scanner s = new Scanner(System.in);
do {
   System.out.printf("Enter 1 to loop: ");
   x = s.nextInt();
} while (x == 1);
System.out.printf("Freedom!%n");
```

`for` Loops

Used as a shortcut for a commonly occurring pattern

- Initialization (once before loop)
- Condition (before each iteration)
- Update (at the end of each loop body)

```
int i=0;
while (i<10) {
    System.out.printf("%d%n", i);
    i++;
}
```

```
for (int i=0; i<10; i++) {
    System.out.printf("%d%n", i);
}
```

`for-each` Loop

- Shortcut to iterate over all elements of some “iterable collection” (more later!)
- Common in many languages, added as of JDK5

```
int[] phone = {8, 6, 7, 5, 3, 0, 9};

for (int x : phone) {
   System.out.printf("%d", x);
}
System.out.printf(" Jenny%n");
```

Truth Table Example

```
boolean[] tf = {false, true};
for (boolean a : tf) {
   for (boolean b : tf) {
      for (boolean c : tf) {
         System.out.printf(
            "%-5s | %-5s | %-5s | %-5s%n",
            a, b, c,
            ((a && !b) || (!a && b)) && !c);
      }
   }
}
```

Breaking a Loop

- The `break` statement allows you to terminate a loop early
  - Immediately ends the inner-most loop in which it is found (like `return` for loops)

- When used well, typically improves efficiency by reducing the number of unnecessary iterations (e.g. when something is found early in an array)

Example (try with/without **break**)

```
final int[] haystack = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
final int needle = 2;

boolean found = false;
int i = 0;

for (i=0; i<haystack.length; i++) {
    if (haystack[i]==needle) {
        found = true;
        break;
    }
}

System.out.printf("found: %b, loop iterations: %d%n",
                  found, i);
```

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- **Methods**
- Exceptions, File I/O
- Misc

Methods

- Programs can be logically broken down into a set of tasks
- Individual tasks can be separated out from the main program into methods
- A method is simply a mini-program that completes a specific task
- Great for avoiding mistakes from writing the same code in multiple places in the program

Method Pieces

```
public static int smallerOf(int a, int b) {
   return (a<=b)?a:b;
}
```

Every method has…

- A name
  - Multiple can have the same ("overloading") as long as something about parameter list is changed (type, number)
- A return type (or **void**)
- A parameter list (any number of [type name],)
- Visibility (**public**, **private**, **protected**)
  - None = package protected (more later)
- Membership (**static** vs. member)
  - Owned by the class (static) or each object

Invocation ("Calling")

- Static:
  - `ClassName.methodName([arg1, arg2, …])`
  - `methodName()` if in same class (laziness!)
- Member:
  - `objName.methodName([arg1, arg2, …])`

- Execution stops, hops to method, goes back to line when hit end of method or first use of **`return`** (there might be multiple)
  - Even **`void`** methods can use **`return;`**

Example

```
public class Bar {
    public static int smallerOf(int a, int b) {
        return (a<=b)?a:b;
    }

    public static void main(String[] args) {
        int a=1, b=2;
        System.out.printf("%d %d %d%n", a, a, Bar.smallerOf(a, a));
        System.out.printf("%d %d %d%n", b, b, smallerOf(b, b));
        System.out.printf("%d %d %d%n", a, b, smallerOf(a, b));
        System.out.printf("%d %d %d%n", b, a, smallerOf(b, a));
    }
}
```

Parameter Handling

- When invoking a method, argument values are *copied* to parameters
- Straightforward for primitive variables
  - Means variables unchanged in invoking context
- Since object variables' value is a reference (think memory address), "copying" means that method can in fact *change* the variable permanently
  - Includes arrays (which are actually objects)

Checkup (1)

```java
public static void change(int a) {
   a--;
}

public static void main(String[] args) {
   int x = 10;
   System.out.printf("before: %d", x);
   change(x);
   System.out.printf(", after: %d%n", x);
}
```

Checkup (2)

```
public static void change(int[] a) {
   a[0]--;
}

public static void main(String[] args) {
   int[] x = {10};
   System.out.printf("before: %d", x[0]);
   change(x);
   System.out.printf(", after: %d%n", x[0]);
}
```

Class Constants

- Every variable has a "scope"
  - Where it can be accessed
- Defined by where it is declared
  - From that point to the } of the closest { before
- To have a value accessible in multiple methods within the class, use a variable with class-level scope (declared outside any method)
  - In most cases it is a dangerous practice to have a variable with class-level scope that is mutable

Example

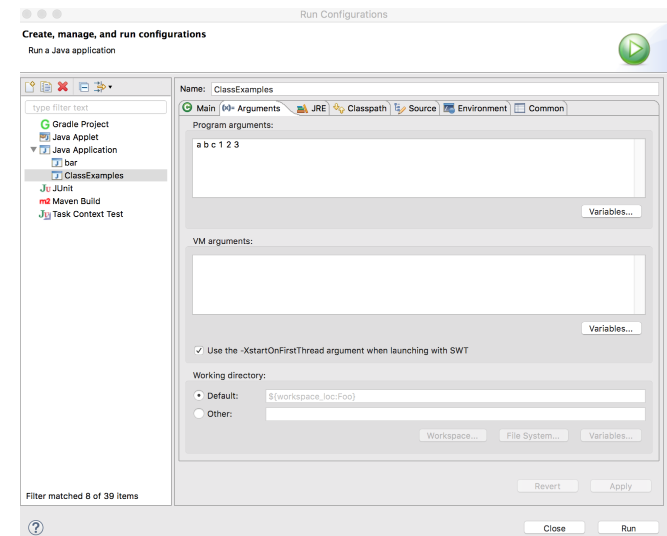
```
public static final double DOLLARS_PER_EURO = 1.05;

public static double dollarsToEuros(double dollars) {
    return dollars / DOLLARS_PER_EURO;
}

public static double eurosToDollars(double euros) {
    return euros * DOLLARS_PER_EURO;
}

public static void main(String[] args) {
    System.out.printf("1 dollar is %.2f euros%n", dollarsToEuros(1));
    System.out.printf("1 euro is %.2f dollars%n", eurosToDollars(1));
}
```

A Note on **`main`**

- The **`main`** method is just like any other
- The parameter is an array of strings, which are any command-line arguments supplied to the program when it is run by the JVM
  - Click the “Run” menu -> “Run Configurations”
  - Find your application on the left list under “Java Application”
  - Click the “Arguments” tab on the right
  - Type some values, separated by spaces, into the “Program arguments” box

Example

```
public class ClassExamples {
    public static void main(String[] args) {
        System.out.printf("%d:", args.length);
        for (String arg : args) {
            System.out.printf(" %s", arg);
        }
    }
}
```



Checkup

Write a method that searches a supplied array (of integers) for a supplied integer value. If found, return the index of the value; else, -1 (why?).

Answer

```
public static int searchArray(int[] haystack, int needle) {
    for (int i=0; i<haystack.length; i++) {
        if (haystack[i] == needle) {
            return i;
        }
    }
    return -1;
}
```

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- **Exceptions , File I/O**
- Misc

Exceptions

Programmatic mechanism to handle errors that occur during execution of the program (runtime)

Verbs…

- **Throw**: when an error occurs, code *throws* a new instance of an Exception class
- **Catch**: code that knows how to handle a particular type *catches* an exception to handle it

Exception Mechanics

- When an exception is thrown, execution sequentially pops out of each scope until either code catches it, or the program ends
- Programmers use **`try-catch`** blocks to handle potential runtime errors within a section of code

try-catch

```
try {
    STATEMENTS THAT MIGHT THROW EXCEPTIONS
} catch (EXCEPTION_TYPE1 EXCEPTION_VARIABLE1) {
    STATEMENTS THAT HANDLE EXCEPTION_TYPE1
} catch (EXCEPTION_TYPE2 EXCEPTION_VARIABLE2) {
    STATEMENTS THAT HANDLE EXCEPTION_TYPE2
}
…
```

Example

```
Scanner input = new Scanner(System.in);
int inputValue = 0;

try {
   System.out.printf("Enter an integer: ");
   inputValue = input.nextInt();
} catch (InputMismatchException ex) {
   System.out.printf("Error! Integer required!%n");
   System.exit(0);
}

System.out.printf("%d^2=%d%n",
      inputValue, inputValue*inputValue);
```

Passing the Buck

If a method programmer opts not to catch exception(s), these should be listed in the method signature via the throws keyword

```
public static int readInt(Scanner s) throws InputMismatchException {
    System.out.print("Enter an integer: ");
    return s.nextInt();
}
```

Example

```
public static void doSomethingBad() throws IOException
{
    throw new IOException("Oops");
}

public static void main(String[] args) {
    try {
        doSomethingBad();
        System.out.printf("Yay :)%n");
    } catch (IOException e) {
        System.out.printf("%s :(%n", e.getMessage());
    }
}
```

I/O

- I/O stands for Input/Output
- So far, we've used a Scanner object based on **`System.in`** for all terminal input (usually user's keyboard) and **`System.out`** for all terminal output
- **`System.in`** and **`System.out`** are predefined I/O objects that are available automatically in every Java program

Files

- Files are useful to store large data sets for a program and/or to save the need to type in all the input data values individually

- Accessed via File objects

```
File f = new File("path");
```

  - Paths default to current directory, unless absolute (prefixed with `c:/` or `/`) are given

- For basic cases, we use **`Scanner`** objects for file input and **`PrintWriter`** objects for output

Issues with Files

- When opening files, Java forces you to handle the situation of a file not being found via FileNotFoundException
- When you are done, you should close the file – make sure this happens no matter what!
- To handle both these issues, we use the **`try-with-resource`** block in this class

Example File Input

```
try (Scanner fin = new Scanner(new File("test.txt"))) {
    while (fin.hasNextLine()) {
        String nextLine = fin.nextLine();
        System.out.printf(nextLine+"%n");
    }
} catch (FileNotFoundException ex) {
        System.out.printf("File not found!%n");
        System.exit(0);
}
```

Example File Output

```
try (PrintWriter fout = new PrintWriter(new File("numbers.txt"))) {
    for (int i=1; i<=100; i++) {
        fout.printf("%d%n", i);
    }
} catch (FileNotFoundException ex) {
    System.out.printf("File not found!%n");
    System.exit(0);
}
```

Example File Input/Output

```
try (
    Scanner fin = new Scanner(new File("numbers.txt"));
    PrintWriter fout = new PrintWriter(new File("odds.txt"));
) {
    while (fin.hasNextInt()) {
        int next = fin.nextInt();
        if (next % 2 == 1) {
            fout.printf("%d%n", next);
        }
    }
} catch (FileNotFoundException ex) {
    System.out.printf("File not found!%n");
    System.exit(0);
}
```

COMP1000 Topics

- Computation/Programming
- Variables, I/O
- Expressions
- Arrays
- Control Flow, Conditionals, Loops
- Methods
- Exceptions, File I/O
- **Misc**

Random Numbers

- Computers can't *really* come up with random numbers, but there are sophisticated algorithms (pseudo-random generators, RNGs) to make numbers that appear so via the Random object
- If the same "seed" is used, the object will produce the same sequence

```
Random rng1 = new Random();
Random rng2 = new Random(123);
System.out.printf("%d %d%n", rng1.nextInt(), rng2.nextInt());
System.out.printf("%d %d%n", rng1.nextInt(), rng2.nextInt());
```

Example

```
public static void main(String[] args) {
      Scanner s = new Scanner(System.in);

      System.out.printf("File: ");
      String fname = s.nextLine();

      long seed = 0;
      boolean achieved = false;
      do {
            try {
                  System.out.printf("Seed: ");
                  seed = s.nextLong();
                  achieved = true;
            } catch (InputMismatchException e) {
                  System.out.printf(">:(%n");
                  s.nextLine();
            }
      } while (!achieved);

      write(fname, seed, 100);
}
```

```
public static void write(String fname, long seed, int n) {
      try (PrintWriter f = new PrintWriter(new File(fname))) {
            Random rng = new Random(seed);
            for (int i=0; i<n; i++) {
                  f.printf("%d%n", rng.nextInt(10));
            }
      } catch (FileNotFoundException e) {
            System.out.printf("File not found!%n");
      }
}
```

Take Home Points

- If all of this made sense, you are ready for COMP1050
  - Note: most OOP content was ignored – that's this class!
- If you had troubles, the complete set of COMP1000 slides are on Blackboard, including exercises for you to try
  - Also feel free to talk with me!