

Notes: Introduction to Java Programming

Writing Java Code

- You write Java code as text. You then **compile** it into compiler instructions using a program called a compiler. When your java code is compiled it turns into bytecode. Then you can **run** the bytecode using the Java Runtime Environment.
- In this class you will be writing your Java code using an Integrated Development Environment (IDE) called **JGrasp**. IDEs allow you to type your code, compile your code, and run your code.

Class and Main Declarations

- Java code must be written inside of a **class**

```
public class ClassName {  
}
```

- A class is executable only if it has a **main** method; the main method is the launch point of a Java program

```
public static void main(String[] args) {  
}
```

Printing to the Screen

- `System.out.println("text");` prints the information inside the parentheses to the console, followed by a new line
- `System.out.print("text");` prints the information inside the parentheses to the console, without a following new line
- Note that statements/commands in Java end with semicolons

Escape sequences

<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\"</code>	Double quote

	Backslash
---	-----------

Comments

- Comments are used to leave descriptive notes in code or to prevent code from executing:
 - Single Line comments

```
// The rest of the line after two forward slashes is ignored by the compiler
// Single line comments are... for a single line
```

- Block comments

```
/*
Everything between the opening slash-star and the closing star-slash is
ignored by the compiler. These block comments can span several lines of text.

It is important not to put a space between the slash and the star.
*/
```

- When code is put in a comment it is considered “commented out” and does not execute

```
// System.out.println("This line doesn't execute.");
```

Methods

- The `main` should be short and give an good overview of what your program does (like an outline to an essay) by calling other methods
- **Declaring a method:** defines the method, but does not execute it

```
public static void sayHi() {
    System.out.println("Hi!");
}
```

- **Calling a method:** executes the method

```
sayHi();
```

- Methods can call other methods
- Methods can call themselves, but should not do so in this course
- Not all programming solutions are created equal; when writing code, you should write DRY (**D**o not **R**epeat **Y**ourself) code where possible. You do this by defining methods and then calling them multiple times.

Style

- Classes start with a capital letter, no spaces, all other words in name are also capitalized, e.g.:

```
public class TestMain
```

- Methods start with lowercase letters, no spaces, all other words in name are capitalized, e.g.:

```
public static displayRules()
```

- Indentation is important even though Java ignores it; when you open a curly brace everything inside is indented, e.g.:

```
public static void main(String[] args) {  
    System.out.println("I'm indented one tab");  
}
```

Notes: Primitive Data and Definite Loops

Literals / Values

- int literal: number without a decimal, e.g., -7, 0, 103
- double literal: number with a decimal, e.g., -7.0, 0.2, 103.5
- String literal: characters surrounded by quotes, e.g., "hello world"

Data types

- Three main data types (you will learn more later)

Data type	Description
int	integers, pos, zero, neg, up to $2^{31}-1$
double	floating point numbers (real), pos, zero, neg, up to 10^{308}
String	series of text characters

Operators

- **Java precedence:** when operators of the same precedence appear next to each other, they are evaluated left to right.

Priority	Operation
1	parens
2	unary operations, casting
3	multiplication, division, mod
4	addition, subtraction, string concatenation
5	less than, less than or equal to, greater than, greater than or equal to
6	equal to, not equal to

- **Integer division:** When dividing integers, all information after the decimal point is lost. This is called truncating.

```
System.out.println( 6 / 2 );      //results in 3
System.out.println( 6 / 2.0 );   //results in 3.0
System.out.println( 6.0 / 2 );   //results in 3.0
System.out.println( 13 / 2 );    //results in 6
System.out.println( 13 / 2.0 );  //results in 6.5
```


- **Casting:** You can force Java to change a data type. If you cast a double to an int, you always round down (truncate).

```
System.out.println( (double) 47 );      //results in 47.0
System.out.println( (double) 47 / 2 );   //results in 23.5
System.out.println( (double)(47 / 2) );  //results in 23.0 bc the paren's ha
ppen first
System.out.println( (int) 3.2 );         //results in 3
System.out.println( (int) 3.9 );         //results in 3
```

Variables

- We need to declare variables with a type and a name before they can be used
- Once a variable is declared you cannot redeclare it (with a type)

Assignment statements

-  is the assignment operator, it has nothing to do with equality
- Assignment statements should be read right-to-left

```
// 1 is stored in the int variable named x
int x = 1;

// 1 is added to the current value of x (in this case 1), resulting in 2
// and that is stored back into x
x = x + 1;

// "hello" is stored in the String variable named greeting
String greeting = "hello";
```

For Loops

- Canonical Example

```
for(int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

- Structure

```
for(initialization; test; update) {  
    body  
}
```

The order of execution of a for-loop is:

1. initialization
2. check the test condition
3. if the test condition is `true`, execute the statements inside the body of the loop
4. execute the update
5. repeat steps 2 and 3 until the condition is `false`

- Nested Loop Example

```
for(int i = 1; i <= 3; i++) {  
    for(int j = 1; j <= i; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

Generally the outer loop of two nested for-loops controls the number of rows of output while the inner loop controls the output on a single line. The code above produces:

```
1  
12  
123
```

Notes: Definite Loops, Constants

Constants

- Should be at the top of your program
- Start with `public static final`
- Can be used throughout the program and are considered "global variables"

Nest Loops

- With ASCII Art drawing using nested loops:
 - Though for loops generally start with 0 (for example when we learn Arrays and ArrayLists in the future), in ASCII art, we often start with 1
 - The outer loop generally controls the number of line of output
 - The inner loops should have `System.out.print` statements inside, to ensure that Characters are printed next to each other
 - At the very end of the outer loop's body, there is usually a `System.out.println()` so that the next iteration of the outer loop prints on the next line of output

```
/* output:
.....
.....
.....
*/
public static void dotBox() {
    // controls number of lines of output
    for(int line = 1; line <= 3; line++) {
        // controls number of columns of output per line
        for(int dot = 1; dot <= 6; dot++) {
            System.out.print(".");
        }
        System.out.println();
    }
}
```

Case Study: Hourglass Figure

From the textbook:

```
// This produces the top half of the hourglass figure
/* output:
+-----+
|\..../|
| \../ |
|  \ /  |
|   \   |
|  /\   |
| /\.. \ |
|/....\|
+-----+
*/

public class DrawFigure2 {
    public static final int SUB_HEIGHT = 4;

    public static void main(String[] args) {
        drawLine();
        drawTop();
        drawBottom();
        drawLine();
    }

    // Produces a solid line
    public static void drawLine() {
        System.out.print("+");
        for (int i = 1; i <= (2 * SUB_HEIGHT); i++) {
            System.out.print("-");
        }
        System.out.println("+");
    }

    public static void drawTop() {
        for (int line = 1; line <= SUB_HEIGHT; line++) {
            System.out.print("|");
            for (int i = 1; i <= (line - 1); i++) {
                System.out.print(" ");
            }
            System.out.print("\\");
        }
    }
}
```



```

        for (int i = 1; i <= (2 * SUB_HEIGHT - 2 * line); i++) {
            System.out.print(".");
        }
        System.out.print("/");
        for (int i = 1; i <= (line - 1); i++) {
            System.out.print(" ");
        }
        System.out.println("|");
    }
}

// This produces the bottom half of the hourglass figure
public static void drawBottom() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
            System.out.print(" ");
        }
        System.out.print("/");
        for (int i = 1; i <= 2 * (line - 1); i++) {
            System.out.print(".");
        }
        System.out.print("\\");
        for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
            System.out.print(" ");
        }
        System.out.println("|");
    }
}
}

```

Notes: Parameters, Return Types, and Objects

Key Words

- **scope**: the part of the program where a variable exists
- A variable declared in a for loop exists only in that loop
- A variable declared in a method exists only in that method
- **parameter**: sends information **in** from the caller of a method
- **return**: sends information **out** from a method to its caller

parameters

- Parameters are used to bring information into a method
- Parameters (formal parameters) must have the data type in front of a variable name
- Arguments (actual parameters) are what we call the information actually passed to the method
- You can have methods with the same exact name, as long as the number or order of types of the parameters is different. This is called **method overloading**.

```
public static void main(String[] args) {
    // calls the method "food" with the arguments "eggs", "salad", and "steak"
    food("eggs", "salad", "steak");
}

// Defines the method "food" to take three parameters
public static void food(String breakfast, String lunch, String dinner) {
    System.out.println("I had " + breakfast + " for breakfast");
    System.out.println("I had " + lunch + " for lunch");
    System.out.println("I had " + dinner + " for dinner");
}
```

return statements

- A return statement sends information **out** from a method to its caller
- There cannot be statements executed after a return statement in a method
- The return type should be provided as the third keyword in your method header
- The type of the information returned from a method must match the return type, unless the return type is `void` which means that nothing is returned from the method.

```
public static void main(String[] args) {
    hello();    // calls a void method
    three();    // calls method three() but does nothing with returned value
    int val = three(); // calls three() and saves the returned value in variable 'val'
    System.out.println("method three() returned: " + val); // displays the value returned by three()
}

// This method has a return type of void
public static void hello() {
    System.out.println("Hello");
}

// This method has a return type of int
public static int three() {
    return 3;
}
```

Math class

- The **Math** class has a number of useful methods; check out more in the [Java API](#)
- Many of these methods return double. If you want an int, you will use casting

Math methods

Methods	Description
<code>Math.abs</code>	returns the absolute value of a number
<code>Math.sqrt</code>	returns the square root of a number
<code>Math.pow(base, exp)</code>	returns base raised to the exp
<code>Math.max(x, y)</code>	returns the larger of x and y
<code>Math.min(x, y)</code>	returns the smaller of x and y
<code>Math.ceil(x)</code>	returns x rounded up to the nearest whole number
<code>Math.floor(x)</code>	returns x rounded down to the nearest whole number
<code>Math.round(x)</code>	returns x rounded in the appropriate direction (up for 0.5 and above), down otherwise

String class

String methods

Methods	Description
<code>str.length()</code>	returns the number of characters in str
<code>str.charAt(index)</code>	returns the character at index
<code>str.indexOf(str2)</code>	returns the index of the first occurrence of str2, -1 if str2 is not present
<code>str.substring(start, stop)</code>	returns a string of the characters from start (inclusive) to stop (exclusive)
<code>str.toUpperCase()</code>	returns str in all uppercase
<code>str1.equals(str2)</code>	tests whether str1 contains the same characters as str2

Methods	Description
<code>str1.equalsIgnoreCase(str2)</code>	tests whether str1 contains the same characters as str2, ignoring case
<code>str1.startsWith(str2)</code>	tests whether str1 starts with the characters in str2
<code>str1.endsWith(str2)</code>	tests whether str1 ends with the characters in str2
<code>str1.contains(str2)</code>	tests whether str2 is found inside of str1

String are Objects

- Strings in Java are objects
- Strings can contain the same characters but not be equal because in Java they are stored as different objects (even though they have the same characters)
- Because of this, you should use `.equals()` when comparing Strings and not `==`

```
// word1 and word2 are different objects
String word1 = "hello";
String word2 = "hello";

// do not use!
if (word1 == word2) {
    ...
}

// use this instead!
if (word1.equals(word2)) {
    ...
}

// you could also have use this; it does the same as the one directly above
if (word2.equals(word1)) {
    ...
}
```

Notes: User Input and Conditionals

Scanner class / User input

- We will be using `Scanner` for user input
- In order to use `Scanner` you need to add an import statement to the top of your code: `import java.util.*;`
- In main, you will use `Scanner console = new Scanner(System.in);` to create a Scanner object named console that you can pass to any of your methods that need user input
- Note: The `console` name is arbitrary, if it makes sense use a different name
- Note: You should only ever construct 1 Scanner object and pass it in as a parameter to only the methods that need it
- **token:** A sequence of characters that are not white space (e.g., tabs, spaces, etc)

Scanner methods

Method	Description
<code>nextInt()</code>	reads a token of user input as an <code>int</code> ; can only read ints, otherwise error
<code>nextDouble()</code>	reads a token of user input as a <code>double</code> ; can read doubles and ints (converts to double)
<code>next()</code>	reads a token of user input as a <code>String</code>
<code>nextLine()</code>	reads a line of user input as a <code>String</code> ; will include white space characters

Example

```
import java.util.Scanner;
public class UserInputExample {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        age(console);
        int diff = age(15);
        System.out.println(diff + " years until you are 40.");
    }

    // this method PROMPTS for an age and PRINTS the result
    public static void age(Scanner console) {
        System.out.print("How old are you? ");
        int age = console.nextInt();
        System.out.println("You'll be 40 in " + (40 - age) + " years.");
    }

    // this method TAKES an age and RETURNS the result
    public static int age(int age) {
        return 40 - age;
    }
}
```

Expressions that result in a boolean (true/false)

Relational Operators

Operator	Description	Example	Result
<code>==</code>	equals (for primitive types)	<code>1 + 1 == 2</code>	<code>true</code>
<code>s.equals()</code>	equals (for Strings and other reference types)	<code>s.equals("hi")</code>	
<code>!=</code>	does not equal (for primitive types)	<code>3.2 != 2.5</code>	<code>true</code>
<code>!s.equals()</code>	not equals (for Strings and other reference types)	<code>!s.equals("hi")</code>	
<code><</code>	less than	<code>10 < 5</code>	<code>false</code>
<code>></code>	greater than	<code>10 > 5</code>	<code>true</code>
<code><=</code>	less than or equal to	<code>126 <= 100</code>	<code>false</code>
<code>>=</code>	greater than or equal to	<code>5.0 >= 5.0</code>	<code>true</code>

Logical operators

Operator	Description	Example	Result
<code>&&</code>	and	<code>(2 == 3) && (-1 < 5)</code>	false
<code> </code>	or	<code>(2 == 3) (-1 < 5)</code>	true
<code>!</code>	not	<code>!(2 == 3)</code>	true

- `&&` (and) is used in Java to check if two conditions are BOTH true
- `||` (or) is used in Java to check if AT LEAST ONE of two conditions is true
- `!` (not, sometimes read as "bang") is used in Java to negate a condition (make true become false, or make false become true).

Logical Truth Table

p	q	p && q	p q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Negating a boolean

p	!p
true	false
false	true

Conditionals

- `else` can only be used when paired with an `if`
- Note: there should NOT be a semicolon at the end of an `if`-statement condition
- In Java, indentation does not cause statements to belong together. You must use `{}`s

if statements in sequence

```
// independent tests; not exclusive
// 0, 1, or many of the statement(s) may execute
// every test in every if block is checked
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
```

if / else if (no else)

```
// 0, or 1 of the if blocks may execute
// at most only 1 of the if blocks execute
// it could be the case that 0 if blocks execute because there is no else
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```

if / else if / else

```
// exactly 1 of the if blocks will execute
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```

- If statement conditions are evaluated in sequence (top to bottom). If the condition is `true`, then the associated block is executed and the rest of the conditions are skipped. If the condition is `false`, the next condition is tested.
- If there is no `else`, then it is possible that none of the blocks are executed (if none of the conditions were true). However, if there is an `else`, then if the `else` is reached (meaning all conditions before it were false) its associated block will be executed (as `else` basically means "otherwise do this")

Notes: Common Algorithms and printf

Common Algorithms

These are common patterns in programming that are important to know!

Cumulative Sum

```
// returns the sum of integers from 1 up to n
public static int calculateSum(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    return sum;
}
```

Max

```
public static int findMax(Scanner console, int n) {
    int max = Integer.MIN_VALUE;

    for (int i = 0; i < n; i++) {
        System.out.print("Enter a value: ");
        int num = console.nextInt();

        if (num > max) {
            max = num;
        }
    }
    return max;
}
```

Even or Odd

```
public static void evenOrOdd(int n) {  
    if (n % 2 == 0) {  
        System.out.println(n + " is even.");  
    } else {  
        System.out.println(n + " is odd.");  
    }  
}
```

Replicate Entire String

```
// returns a String containing n replications of s  
public static String replicate(String s, int n) {  
    String output = "";  
    for (int i = 0; i < n; i++) {  
        output = output + s;  
    }  
    return output;  
}
```

Reverse String

```
public static String reverse(String phrase) {  
    String output = "";  
    for (int i = 0; i < phrase.length(); i++) {  
        output = phrase.charAt(i) + output;  
    }  
    return output;  
}
```

Using printf

- The f in `printf` stands for formatted
- Allows you to format what you are printing

Example

```
double x = 38.421;
double y = 152.734009;
// the below line will output: formatted numbers: 38.42, 152.7
System.out.printf("formatted numbers: %.2f, %.1f\n", x, y);
```

Common Format Specifiers

Specifier	Result
<code>%.2f</code>	Floating-point number, rounded to nearest hundredth
<code>%d</code>	Integer
<code>%6d</code>	Integer, left-aligned, 6-space-wide field
<code>%f</code>	Floating-point number
<code>%16.3f</code>	Floating-point number, rounded to nearest thousandth, 16-space-wide field
<code>%s</code>	String

Notes: while loops and fence post problems

definite vs indefinite loop

- **definite loop**: executes a known number of times
 - e.g. Print the numbers 1 to 100 to the screen
- **indefinite loop**: where the number of times the loop will repeat is unknown prior to the code executing
 - e.g. Ask the user for a number until they enter a value between 1 and 100
- for-loops are typically definite loops and while-loops are typically indefinite loops, but this is not always the case. You must pay attention to whether or not you know how many times the loop will number

`while` loop

- Repeatedly executes its body as long as a logical test is true
- Note: The `for` loop is a specialized form of the `while` loop
- Use `while` when it is **unknown** how many times the loop will repeat (meaning you don't know right now, even if you could put in some time and effort to figure out the exact amount)
- Use `for` when it is **known** how many times the loop will repeat

Structure

```
while (test) {  
    statement(s);  
}
```

- while loops are like repeating if statements; the body of the loop repeats if the condition is true when checked
- the condition of a while loop is checked at the start of the loop; if this condition is not true to start, the loop will not execute
- after the body of the loop is executed, the condition is checked again; if the condition is true again, the loop executes again
- while loops can be used to count or do other tasks that run a set number of times, but in these cases, generally a definite (for loop) is more appropriate

Sentinels

- **sentinel value:** A value that signals the end of user input
- **sentinel loop:** A loop that repeats until a sentinel value is seen

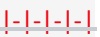


Example

```
// in the following loop the sentinel value is anything but "yes"
// note: that the sentinel value is the stopping case
// and is opposite of the repeating case (the condition)

String again = "yes";
while (again.equals("yes")) {
    // do something

    System.out.print("Go again? (yes/no) > ");
    again = console.next();
}
```

Fencepost problems

- Fencepost problems are when you have a repeating pattern that needs to happen, but part of the pattern doesn't repeat exactly.
 - e.g. If you want to print 1, 2, 3, 4, 5 to the screen using a loop, you need to print each of the numbers followed by a comma, except the last number (5 in this case)
- Sometimes this is illustrated as  with the  representing a fence post and the  representing the wire of the fence; Also sometimes called a "loop-and-a-half"
- The idea is that you need 1 more post than you do wire sections; you begin with a post and end with a post
- Common solutions usually have the loop run one less times than needed and then handle the last post outside of the loop

Examples

```
// handles the first post outside the loop
// prints a comma separated list of numbers from 1 up to max
public static void printNumbers(int max) {
    System.out.print(1);
    for(int i = 2; i <= max; i++) {
        System.out.print(", " + i );
    }
    System.out.println();
}
```

```
// handles the last post outside the loop
// prints a comma separated list of numbers from 1 up to max
public static void printNumbers(int max) {
    for(int i = 1; i <= max - 1; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(max);
}
```

Notes: Random numbers and Assertions

Random class

- A `Random` object generates pseudo-random numbers
- pseudo-random means simulated randomness, but not truly random
- In order to use the `Random` class you will need to import the util package: `import java.util.Random;`

Random methods

Method Name	Description
<code>nextInt()</code>	returns a random integer
<code>nextInt(max)</code>	returns a random integer in the range [0, max) (i.e., 0 to max - 1 inclusive)
<code>nextDouble()</code>	returns a random real number in the range [0.0, 1.0)

To get a number in an inclusive range of min to max

```
nextInt(max - min + 1) + min
```

Example code

```
Random rand = new Random();

// randomNumber1 will store a random number in the range 0 – 9
int randomNumber1 = rand.nextInt(10);

// randomNumber2 will store a random number in the range 1 - 20
int randomNumber2 = rand.nextInt(20) + 1;

// randomNumber3 will store a random number that is one of the first 5 even numbers (0, 2, 4, 6, 8)
```

```
int randomNumber3 = rand.nextInt(5) * 2;
```

Boolean

- `boolean` is a type
- `boolean` variables can hold either `true` or `false`
- Using a `boolean`
 - create a boolean variable
 - pass a boolean value as a parameter
 - return a boolean value from methods
 - call a method that returns a boolean and use it as a test

Boolean Zen

- do not test a result against `true`

```
// don't do this
if(result == true) {...}

// do this instead
if(result) {...}
```

- do not test if a condition is `true` and then `return true` as a result, just return the boolean expression itself

```
// don't do this
if(count > 10 == true) {
    return true;
}
// do this instead
return count > 10;
```

- do not create variables to return information that can be returned without a variable


```
// don't do this
boolean result;
if(!word.equals("y")) {
    result = false;
}
return result;

// do this instead
if(!word.equals("y")) {
    return false;
}
```

Logical assertions

- Assertion: A statement that is either **true** or **false**
- Tips for solving assertion problems:
 - Right after a variable is initialized, its value is known
 - At the start of a loop's body, the loop's test must be true
 - After a loop, the loop's test must be false
 - Inside a loop's body, the loop's test may become false
 - Reading from a Scanner, reading from a Random object, or parameter values are unknown and usually result in a "Sometimes" assertion
 - If you are unsure, guess "Sometimes"

Notes: Token-based Processing and File Output

File Paths

- Absolute vs Relative File Paths
 - **absolute path**: complete path to a file, can use that path anywhere on your system and it will locate the file
 - usually begins with `C:/` (on Windows) or `/` (on Mac/Linux)
 - e.g. for Windows: `C:/Documents/cs141/lectures/day11/numbers.txt`
 - e.g. for Mac/Linux: `/Users/chess/Documents/cs141/lectures/day11/numbers.txt`
 - **relative path**: path to the file from the current directory, if you used this path in a different directory it would not go to your file
 - usually are a single file name or a series of folders followed by a single file name
 - e.g. for Windows: `lectures/day11/numbers.txt`
 - e.g. for Mac/Linux: `lectures/day11/numbers.txt`
- Note that when you put File Paths into code, you need to make sure that all the slashes are forward slashes (/) because otherwise you could create escape sequences in your path Strings such as `"\n"` in `"day11\numbers.txt"`
- Some directory (folder) commands that you should know:
 - `.` means the directory that you are in
 - `/` means the root directory (when at the front of a path)
 - `..` means the parent directory ("go back a folder")
 - Files end with an extension (.doc, .txt, .java, .pdf, etc) and Folders end with a slash (cs141/, homeworks/, Documents/, etc)
 - `*` means "wildcard"; meaning, anything can go here

File Processing

- To make a .txt file in jGRASP: File > New > Plain Text
 - We will use text files to store information that we later read out instead of reading from the console
- To read from a text file:
 - Use the `File` class. You need to have an import statement, `import java.io.*;`
 - Create a new File object by saying: `File f = new File("numbers.txt");`
 - Then connect your file to a Scanner object with: `Scanner input = new Scanner(f);`
 - Or you can combine these lines into one: `Scanner input = new Scanner(new File("numbers.txt"));`
- Once you have a Scanner object, you can then use the methods we have learned previously for reading in information: `input.next()`, `input.nextInt()`, `input.nextDouble()`
 - Note that Scanner objects can only go forwards, they cannot read information backwards; if you need to read a file twice then you would need to create two Scanner objects
- Because of some exception rules in Java you need to add `throws FileNotFoundException` to any method that constructs a Scanner from a File object or calls a method that does so

File Example

```
import java.io.*; // to use the File class
import java.util.*; // to use the Scanner class
public class FileExample {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double n = input.nextDouble();
            System.out.println("n = " + n);
            sum += n;
        }
        System.out.println("sum = " + sum);
    }
}
```

Testing for valid input with Scanner

Assuming a Scanner object has already been created named *input*

Method name	Description
<code>input.hasNext()</code>	returns <code>true</code> if there are more tokens of input to read
<code>input.hasNextInt()</code>	returns <code>true</code> if there is a next token and it can be read as an <code>int</code>
<code>input.hasNextDouble()</code>	returns <code>true</code> if there is a next token and it can be read as an <code>double</code>

Common Errors/Exception Messages

- `InputMismatchException`: When you try and read a token of the wrong type
- `NoSuchElementException`: When you try and read a token that does not exist

Common File Methods

Method name	Description
<code>f.canRead()</code>	returns whether the file <i>f</i> is able to be read
<code>f.delete()</code>	removes the file <i>f</i> from the disk
<code>f.exists()</code>	returns true if the file <i>f</i> exists, otherwise returns false
<code>f.getName()</code>	returns the name of file <i>f</i>
<code>f.length()</code>	returns the number of bytes in the file <i>f</i>
<code>f.renameTo(name)</code>	changes the name of file <i>f</i> to <i>name</i>

File Output / PrintStream

- Requires creation of a `PrintStream` object
- `PrintStream` is an object in the `java.io` package that lets you print output to a destination (e.g., a file)
- All the methods you have been using for `System.out` can also be used on `PrintStream` objects
- Important `PrintStream` details
 - If a given file does not exist, then it will be created for you
 - If a given file already exists, then it will be overwritten
 - The output you print will no longer appear on the console (it will be written to the file instead)
 - Do not open the same file for both reading and writing at the same time

Code Example

```
PrintStream output = new PrintStream(new File("output.txt"));
output.println("hello world");
```

Scanning a String

- In addition to using a Scanner to read input from the console (`Scanner console = new Scanner(System.in);`) or using the Scanner to read input from an input file (`Scanner input = new Scanner(new File("data.txt"));`), you can also use a Scanner to read tokens from a simple String:
 - `Scanner lineScan = new Scanner("scan this string literal");`
- Consider the following ex which prints all words in the string that begin with "a"

```
// creates a new Scanner that scans through the String literal provided
Scanner lineScan = new Scanner("spider ant elephant aardvark antelope");
while(lineScan.hasNext()) {
    //reads in current token and advances to the next token
    String word = lineScan.next();
    if(word.startsWith("a")) {
        System.out.println(word);
    }
}
```

- Example which counts the number of words in a String:

```
public class ScanStringExample {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter a phrase > ");
        String line = console.nextLine(); // nextLine reads until a \n
        int count = 0;
        // creates a new Scanner that scans through the phrase entered
        Scanner lineScan = new Scanner(line);
        while(lineScan.hasNext()) {
            //reads in current token and advances to the next token
            String word = lineScan.next();
            count++;
        }
        System.out.println("Number of words entered = " + count);
    }
}
```

File Processing

- `Scanner` is the main tool to use for file processing

- When doing both token-based processing and line-based processing use two different `Scanner` objects

Token-based Processing

- Processes the data in tokens using `nextInt()`, `nextDouble()`, and `next()`
- Skips past any newline characters
- Should not be used if your input is line-based, because token-based processing ignores the line breaks and looks only at the tokens

Line-based Processing

- Process data by line using `nextLine()`
- When doing line-based processing you often use a while loop, because you are unsure of the number of lines you will read in; an example method is shown below. Note that the method below returns the empty string if nothing is found.

```
// searches for and returns the next line of the given input that contains
// the given phrase; returns an empty string if not found
public static String find(Scanner input, String phrase) {
    while (input.hasNextLine()) {
        String line = input.nextLine();
        if (line.toLowerCase().contains(phrase)) {
            return line;
        }
    }
    return "";
}
```

Hybrid Approach

- Use line-based processing on the file, but then use token-based processing for the individual lines of the file
- Do this by passing the line itself (a String) into a new Scanner object to use token-based processing
- An example of this approach is shown below

```
public static void print(String line) {  
    Scanner data = new Scanner(line);  
    String name = in.next();  
    for  
    System.out.print(rank + "\t" + rating + "\t" + votes + "\t");  
    while (data.hasNext()) {  
        System.out.print(data.next() + " ");  
    }  
    System.out.println();  
}
```

Notes: Arrays

Native Arrays

- **array**: an object that stores many values of the same type
- **element**: one value in an array
- **index**: a zero-based integer to access an element from an array

index	0	1	2	3	4	5	6	7	8	9
value	12	49	-2	26	5	17	-6	84	72	3

- In the array above, 12 is stored as element 0, 5 is stored as element 4, and 3 is stored as element 9
- To declare and initialize an array use `type[] name = new type[length];`

```
// creates a new int array to hold 10 values; all starting at zero
int[] numbers = new int[10];
```

- To access elements use `name[index]`
- To modify elements use `name[index] = value`

Details on Arrays

- Different array types have different default values; the default values for common types are shown in the table below

Type	Default Value
<code>int</code>	0
<code>double</code>	0.0
<code>boolean</code>	false
<code>String</code>	null

- If you know the values of an array you want to define, you can use the array initializer syntax
 - `int[] numbers = {2, 3, 5, 7, 11, 13, 17};`
- Arrays have **random access** which means you can access any element in the array without having to loop through the entire array

- The legal indexes of an array are between 0 and the array's length - 1. If you read or write outside of this range you will generate an `ArrayIndexOutOfBoundsException`.
- Arrays and for-loops are best friends
- An array's length field stores the number of elements, `name.length`, notice that there are no parens after length because it is a field and not a method

Printing Arrays

- Arrays are objects, they behave different from primitive types (e.g., int, char, boolean)
- Every object in Java has a special method named `toString()` that defines how that object should be converted into a `String`
- The default `toString()` method for arrays is to print the memory address of the array, so you should need to use a for-loop to print the contents of an array

```
// print out the elements of an array
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println();
```

- **Instead**, you can use the `toString` method of the `Arrays` class to print your array

```
System.out.println(Arrays.toString(name));
```

where *name* is the name of the array you are printing

Code Examples

```
// print out the elements of an array
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println();
```

```
// multiply every value in an array by 2
for (int i = 0; i < numbers.length; i++) {
    numbers[i] = 2 * i;
}
```

```
// count how many values are above 5
int count = 0;
for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] > 5) {
        count++;
    }
}
```

```
// using an array to count the number of each digit in a number
int num = 229231007;
int[] counts = new int[10];
while (num > 0) {
    int digit = num % 10;
    counts[digit]++;
    num = num / 10;
}
```

```
// print a histogram for the values in an array
for (int i = 0; i < numbers.length; i++) {
    System.out.print(i + ": ");
    for (int j = 0; j < counts[i]; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

Arrays Class

The Arrays class (java.util.Arrays) has useful methods for manipulating arrays:

Methods of the [Arrays](#) class

Method Name	Description
equals(array1, array2)	returns true if the two arrays contain the same elements in the same order
fill(array, value)	sets every element in the array to have the given value
sort(array)	arranges the elements in the array into ascending order
toString(array)	returns a String representing the array

Code Examples

```
// create an array called numbers that stores 100 int's
int[] numbers = new int[100];

// use the Arrays class to fill all 100 spots in numbers to be -5
Arrays.fill(numbers, -5);

// use the Arrays class to print all the values in numbers
System.out.println(Arrays.toString(numbers));
```

Notes: More on Arrays

Value Semantics

- Value semantics are used with primitive types in Java (e.g., int, char, boolean)
- If you set a variable equal to another it takes on that new value, but does not constantly update so those variables always have the same values

```
int x = 3;
int y = 4;
int z = x;
z = 42; // z is set to 42, but x does not update to also be 42 (x is still 3)
```

- Because of this, if a primitive variable is change inside a method, the variable in the main is not updated (unless used in conjunction with a return statement)

```
int x = 10;
int y = 20;
add(x,y); // after this call x will still be 10, y will still be 20
y = add(x,y); // after this call x will still be 10, y will be 30

public static int add(int x, int y) {
    // this changes the value of the local var x (not the one in the main)
    x += y;

    // this will print the updated values of x and y,
    //even though these changed values may not persist to the main
    System.out.println(x + " " + y);

    // this returns the local var x's value to where the method was called
    return x;
}
```

Reference Semantics

- Arrays use reference semantics (to contrast, the primitive types use value semantics)
- Arrays use references to where it is stored in memory, not the actual values

```
int[] values = {1, 2, 3};  
int[] values2 = values; // values and values2 point to the same array in memory  
values[0] = 7; // changes both the first value of values, and the first value of values2 to 7
```

- Because of this, you do not need to return an array from a method to get its updated values, since both arrays refer to the same array in memory there is no reason to return the array

```
int[] values = {4, 5, 6};  
triple(values); // after this call, values will contain 12, 15, 18  
  
public static void triple(int [] values) {  
    for (int i = 0; i < values.length; i++) {  
        values[i] = values[i] * 3;  
    }  
}
```

Array Mystery Problems

- Pay attention to the loop bounds, often they will not go until the length of an array
- You should reference the updated values of the array when evaluating the answer instead of the original array
- Note that `a[i]` and `i` are different! `a[i]` refers to the element at index `i`, whereas `i` is just an index

More Code Examples

```
// return the sum of all elements in an array of integers
public static int sum(int[] values) {
    int sum = 0;
    for (int i = 0; i < values.length; i++) {
        sum += values[i];
    }
    return sum;
}

// print the values of an array
int[] primes = {2, 3, 5, 7, 11, 13, 17};
System.out.println(Arrays.toString(primes));
```

Solving an array problem "in place" refers to solving an array problem without creating a new array; you are able to rearrange the array's elements in the original array.

```
// reverse the elements of an array
public static void reverse(int[] values) {
    for (int i = 0; i < values.length / 2; i++) {
        int temp = values[i];
        values[i] = values[values.length - 1 - i];
        values[values.length - 1 - i] = temp;
    }
}

// apply Math.abs to all elements of an array
public static void makePositive(int[] values) {
    for (int i = 0; i < values.length; i++) {
        values[i] = Math.abs(values[i]);
    }
}
```

Notes: OOP

Object Oriented Programming (OOP)

- **OOP**: programs that perform their behaviour as interactions between objects
- Objects group together related variables
- An **object** is an entity that combines **state** and **behavior**
 - An object is a blueprint for a new data type
 - An object is not executable (it does not contain a main method)
- A created object (using the **new** keyword) is an **instance** of a class
- A **client program**, is a program that uses objects (you have already been writing client programs)

State

- An object's state is defined by the collection of things that it can remember (fields/instance variables)
- **Field**: A variable inside an object that remembers something
 - Each object has its own copy of each field
 - To access use <objectinstance>.<field>
 - To modify use <objectinstance>.<field> = <value>
- Not every variable should be a field, only the things the object needs to remember

Encapsulation

- Hiding implementation details of an object from its clients
- Implemented with the **private** keyword before fields; which stops code outside of the class from being able to access it
- If you then want to give access, you add an accessor and mutator method
- Encapsulation provides **abstraction** between an object and its clients
- Encapsulation protects objects from unwanted access by clients

Behavior

- **Instance method**: A method inside an object that operates on that object
- **Implicit parameter**: The object on which an instance method is called; can be referred to with the **this** keyword
- **Constructor**: a special method that has the same name as the class and is used to create an Object instance from the class's blueprint in another program

- Must be public, same name as the class, and should initialize all instance variables
- **Accessor:** a method that provides information about the state of an object; giving clients "read only" access to the object's fields
 - Generally uses the keyword "get" on the method name, takes no parameters, and returns the value of an instance variable
- **Mutator:** a method that modifies the object's internal state; giving clients both read and write access
 - Generally used the keyword "set" on the method name, takes a parameter for the new value of an instance variable in the object, and reassigns an instance variable to the passed parameter (returns void)
- **toString():** a method that allows you to print out the contents of the Object's state

More on Constructors

- The constructor's name needs to be the same as the name of the class itself
- Initializes the state of new objects
- Runs when the client uses the `new` keyword
- Does not specify a return type (no void, no other data type, just nothing)
- Implicitly returns the new object that was created
- Every class needs a constructor, but if you do not define a constructor then Java gives it a default constructor with no parameters and sets all fields to zero
- A class can have multiple constructors, but they each have to have a unique set of parameters
 - If you do have multiple constructors it is possible for one constructor to call another constructor

More on toString()

- By convention, when you print an object Java will always call the `toString()` method of that class
 - This is because every single object inherits from the `Object` class in Java, and within that `Object` class there is a `toString()` method
- By default, Java includes a `toString()` method that returns a `String` containing the memory address of that object
- You can override the `toString()` method to customize the `String` that it returns

Example Code

```
/*
This is an example of what happens inside java.awt.Point
```

```

https://docs.oracle.com/javase/8/docs/api/index.html?java/awt/Point.html
*/

// procedural programming: a program as a sequence of commands
// object oriented programming: a group of interacting Objects

// class: a blueprint for creating an object;
//     defines an object's state and behavior
// state: things the Object remembers (fields)
// behavior: things the Object can do (methods)
public class Point {

    // fields/instance variables: (combined make up the object's state)
    // should be private and declared (but not initialized)
    int x;
    int y;

    // constructors: allow you to create an Object from this class
    // no return type, same name as the class, initialize all fields
    public Point() {
        x = 0;
        y = 0;
    }

    public Point(int startX, int startY) {
        x = startX;
        y = startY;
    }

    // accessors: allow you to access a field of an Object's state
    // use "get", return a field's value, no params
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    // mutators: allow you to change a field of an Object's state
    // use "set", take a parameter for new value, changes field variable
    public void setX(int newX) {
        x = newX;
    }

    // OTHER BEHAVIOR: other methods that don't follow the structure
    // of accessors and mutators

    // Moves this point to the specified location in the (x,y) coordinate plane.
    public void move(int x, int y) {
        this.x = x; //this.setX(x) or setX(x)
        this.y = y;
    }
}

```



```
}

// Translates this point, at location (x,y), by dx along the x axis
// and dy along the y axis so that it now represents the point (x+dx,y+dy).
public void translate(int dx, int dy) {
    x += dx; // this.x = this.x + dx;
    y += dy;
}

// toString: allows you to print out the state of an Object
// must return a string, should NOT have System.out.println here
public String toString() {
    return "(" + x + ", " + y + ")";
}
}
```

Notes: Inheritance

Inheritance

- **inheritance:** a way to form new classes based on existing classes, taking on their attributes/behavior
 - a way to group related classes
 - a way to share code between two or more classes
 - one class can extend another, absorbing its data/behavior
- **superclass:** the parent class that is being extended
 - you can refer to the constructor of the super class using `super()` as the first line in a subclass's constructor
 - you can refer to methods in a super class by using `super.methodname()` in a subclass
- **subclass:** the child class that extends that superclass (and inherits the superclass' behavior)
 - Subclasses inherit all of the public and protected instance methods of the parent class
 - Subclasses inherit all of the public and protected instance and class variables
 - Subclasses can have their own instance variables
 - Subclasses can have their own static and instance methods
 - Subclasses can override the parent class's methods
 - Subclasses can contain constructors that directly invoke the parent class's constructor using the `super` keyword
- **is-a relationship:** a hierarchical connection where one category can be treated as a specialized version of another (uses the keyword `extends`)
 - All classes except for `Object` extend `Object`; you don't have to code it (`extends Object`), Java automatically adds it in (this is why we can call `toString()` on classes that we create before we define our own)
- **inheritance hierarchy:** a set of classes connected by is-a relationships that can share common code
 - multiple levels of inheritance in a hierarchy are allowed
- **polymorphism:** ability for the same code to be used with different types of objects and behave differently with each
- **override:** to write a new version of a method in a subclass that replaces the superclass's version

Code Example

Here we are creating a new class called `AdministrativeAssistant`, but utilizing existing code from the `Employee` class. By doing this, we can utilize methods from the `Employee` class without having to rewrite code.

```
public class EmployeeClientProgram {
    public static void main(String[] args) {
        Employee sally = new Employee();
        System.out.println("Sally works " + sally.getHours() + " hours a week."); // 40

        AdministrativeAssistant bob = new AdministrativeAssistant();
        System.out.println("Bob works " + bob.getHours() + " hours a week."); // 45
    }
}

public class AdministrativeAssistant extends Employee {
    // "extends Employee" --> inherit all state and behavior of an Employee
    // i.e. getSalary, getVacationDays, and getVacationForm exist here
    // even though you don't see them

    // overrides getHours() that was inherited from Employee
    public int getHours() { return super.getHours() + 5; }

    // AdministrativeAssistant adds the takeDictation method.
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}

public class Employee {
    // ... fields, constructors, mutators, toString

    public int getHours() { return hours; } // works 40 hours / week
    public double getSalary() { return salary; } // $40,000.00 / year
    public int getVacationDays() { return vacayDays; } // 10 days
    public String getVacationForm() { return formColor; } // "yellow" form
}
```

Notes: ArrayList

ArrayList

- ArrayLists are like native arrays [] because
 - they both store many elements at one time.
- ArrayLists are different than native arrays [] because
 - the size can be variable for ArrayLists
 - length is fixed for arrays
 - ArrayLists should contain reference data types (Objects) not primitive data types
 - arrays can store either primitive or reference data types
 - ArrayLists have [built in methods \(Links to an external site.\)](#)Links to an external site. that can be called to perform common functions
 - ArrayLists in java are similar to Lists in python
 - arrays do not have built in functions (you cannot use dot-notation on an array)

Constructing

- Must import java.util.ArrayList
- When constructing an ArrayList you must specify the type of elements it will contain
 - This type is called a type parameter or a generic
- You will need to use Wrapper classes when storing int, double, char, and boolean
 - A wrapper is an object whose sole purpose is to hold primitive value
 - Once you construct the list, use it with primitives as normal
 - Java does something called auto boxing/unboxing which means that it converts between primitives and their wrapper class

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

Structure

```
ArrayList<Type> name = new ArrayList<Type>();
```

Example Code

```
ArrayList<String> names = new ArrayList<String>();
names.add("Frankie Manning");
names.add("Chick Webb");
```

ArrayList Methods

Method	Description
<code>add(value)</code>	appends value at end of list
<code>add(index, value)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear()</code>	removes all elements of the list
<code>indexOf(value)</code>	returns first index where given value is found in list (-1 if not found)
<code>get(index)</code>	returns the value at given index
<code>remove(index)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set(index, value)</code>	replaces value at given index with given value
<code>size()</code>	returns the number of elements in list
<code>toString()</code>	returns a string representation of the list such as [3, 42, -7, 15]

ArrayList vs Array

Description	Array	ArrayList
construction	<code>String[] names = new String[5];</code>	<code>ArrayList<String> list = new ArrayList<String>();</code>
storing a value	<code>names[0] = "Martin";</code>	<code>list.add("Martin");</code>
replace a value at an index	<code>names[i] = "Martin";</code>	<code>list.set("Martin", i)</code>
accessing a value	<code>String s = names[0];</code>	<code>String s = list.get(0);</code>
how many elements?	<code>names.length</code>	<code>list.size();</code>

ArrayList Code Examples

```
// moves the max value to the front of the given list, otherwise preserving the order of the elements
public static void maxToFront(ArrayList<Integer> list) {
    int max = 0;
    for (int i = 1; i < list.size(); i++) {
        if (list.get(i) > list.get(max)) {
            max = i;
        }
    }
    list.add(0, list.remove(max));
}
```

```
// returns the length of the longest String in the given list
public static int maxLength(ArrayList<String> list) {
    int max = 0;
    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);
        if (s.length() > max) {
            max = s.length();
        }
    }
    return max;
}
```

Account for re-indexing when removing from an ArrayList inside a loop

When you remove an element from an ArrayList, the ArrayList auto renumbers. You must account for, otherwise you will accidentally skip over processing elements (specifically the ones immediately after a remove).

Here are three examples of the same method re-written to account for the reindexing

```
// Version 1: removes from the list all strings of even length
public static void removeEvenLength(ArrayList list) {
    int i = 0;
    while (i < list.size()) {
        String s = list.get(i);
        if (s.length() % 2 == 0) {
            list.remove(i);
        } else {
            i++;
        }
    }
}
```

```
// Version 2: removes from the list all strings of even length
public static void removeEvenLength(ArrayList list) {
    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);
        if (s.length() % 2 == 0) {
            list.remove(i);
            i--;
        }
    }
}
```

```
// Version 3: removes from the list all strings of even length
public static void removeEvenLength(ArrayList list) {
    for (int i = 0; i < list.size(); ) {
        String s = list.get(i);
        if (s.length() % 2 == 0)
            list.remove(i);
        else
            i++;
    }
}
```