CISC 1115 (TY8) Exam 2 Review Sheet

***NOTE**: This review sheet covers most (but not all) topics for your upcoming exam. It is here to clear up any confusion you may have about a particular concept; however, please keep in mind that we cannot and will not provide you with all the answers. Programming is a hands-on skill; please take responsibility for understanding and using these concepts. Please work on the sample exercises provided; the only way to get better at writing code is by **actually writing code**.*

Previous Topics

- Chapter 2: Variables and operators
- Chapter 3: Input and output
- Chapter 5: Conditionals and Logic
- Chapter 7: Loops

Topics (NEW)

- Chapter 9: Strings and things
- Chapter 4: Void methods
- Chapter 6: Value methods
- Chapter 8: Arrays (UP TO SEARCHING)

Chapter 2: Variables and Operators

Identifiers and Keywords

- Keywords are predefined, reserved words used in Java programming with special meanings to the compiler.
    - **For example: int, char, double, new, return, static, etc.**
- You cannot use keywords like int, for, class, etc. as variable names (or identifiers) as they are part of the Java programming language syntax.
- Identifiers are the names given to variables, classes, methods, etc.
    - **For example: n, num, number, score, etc.**

Types of Programming Errors

- **Syntax/Compiler Error:** An error that prevents the program from compiling successfully.
- **Run-time Error:** An error that occurs while the program is running.
- **Logical Error:** An error where the program runs, but does not produce the intended result.

When debugging code, you should be able to identify the error type, determine its cause, and correct the code.

Arithmetic Expressions

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

- Pre-increment (++x), i.e. System.out.println(++x); ➡➡➡ result: new value of x
- Post-increment (x++), i.e. System.out.println(x++); ➡➡➡ result: original value of x, store new value for next action
- (=): assignment vs (==): equivalence

Arithmetic Precedence

Java evaluates arithmetic expressions according to the following order of precedence:

Precedence	Operator(s)	Description	Example
1 (Highest)	()	Parentheses	(x + y) * z
2	++ --	Increment / Decrement	x++, --y
3	* / %	Multiplication, Division, Modulus	x * y, x / y, x % y
4 (Lowest)	+ -	Addition, Subtraction	x + y, x - y

Important:

- Expressions inside **parentheses** are evaluated first.
- Operators at the **same precedence level** are generally evaluated from **left to right**.
- Parentheses can be used to change the order in which an expression is evaluated.

Example:

```
int result = 2 + 3 * 4;       // 14
int result2 = (2 + 3) * 4;    // 20
```

Java.lang.Math class methods

- Math Class methods are built-in methods that make numeric operations like squaring, square root, cubing, cube root, exponentiation, and trigonometric operations easier.

- Math class methods include (but are not limited to):

    - **abs()**: java.lang.Math.abs(datatype arg) returns the absolute value of the argument. This method can handle all data types.
        - **Parameters:**
            arg: the argument whose absolute value we need
        - **Returns:**
            the absolute value of the passed argument

    - **pow()**: java.lang.Math.pow(double b, double e) method returns the value as b^e
        - **Parameters:**
            b: base
            e: exponent
        - **Returns:**
            value as base^exponent

    - **sqrt()**: java.lang.Math.sqrt(double a) method returns the correctly rounded positive square root of a double value
        - **Parameters**:
            a - a value
        - **Returns:**
            the positive square root of a. If the argument is NaN or less than zero, the result is NaN

    - **random()**: java.lang.Math.random() method returns a pseudorandom double type number greater than or equal to 0.0 and less than 1.0
        - **Returns:**
            a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

        - **To generate a double value within a specific range:**
            Math.random() * (max - min) + min

      Where min is inclusive, and max is exclusive
      (i.e., to generate a number between 1 and 10, set min to 1 and max to 11)

    - **See the Java API for the full list of Math class methods: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Math.html.**

Chapter 3: Input and Output

Writing to the Screen (System.out)

- System.out represents the Standard Output Stream. That means that if we want to print any statement on the console, we should use the following statement:

System.out.print();

- There are three methods we use to print statements:
    - **print(String s)**: used to print on the console. It accepts a string as a parameter (we can concatenate parts to the string if necessary, i.e., “Hello” + “ World”, etc.; however, it is typical convention to include all string literals in the same pair of quotation marks if they are consecutive). After printing the statement, the cursor remains on the same line.
    - **println(String s)**: upgraded version of the print() method. It is also used to display text on the console. After printing the statement, it moves the cursor to the start of the next line. This is the main difference between println() and print().
    - **printf(String format, datatype args)**: It accepts two parameters:
        - format: It is a formatted String that uses placeholders with specific conversion characters to refer to the arguments that will be replaced when printing
        - args: It is an argument referenced by the format specifiers. If the number of arguments is more than the format specifiers, the other arguments are ignored. The number of arguments may be zero.
        - Conversion characters are only valid for certain data types. Here are some common ones:
            - **%s formats strings (String)**
            - **%d formats decimal integers (int)**
            - **%f formats floating-point numbers (double)**
        - Optional Modifiers:
            - **The [flags] define standard ways to modify the output and are most common for formatting integers and floating-point numbers.**
            - **The [width] specifies the field width for outputting the argument. It represents the minimum number of characters written to the output.**

- **The [.precision] specifies the number of digits of precision when outputting floating-point values. You can also use it to define the length of a substring to extract from a String.**

- %[flags][width][.precision]conversion-character (i.e., %8.2f would print a double value with two decimal places and a minimum field width of 8 characters, adding spaces to the left if necessary.)

- printf() throws a **NullPointerException** if the format is null, an **IllegalFormatException** if a format string contains illegal syntax, and an **IllegalFormatConversionException** if the placeholder and corresponding argument are not the same type.

Read Interactively from the Keyboard (System.in)

- All reading done in Java uses the Scanner class. Using this class, we can create an object to read input from the standard input channel System.in (the keyboard)

- To use the Scanner class, it is necessary to import the class Scanner from the library java.util by including the following line at the top of the program:

import java.util.Scanner;

- We can declare a Scanner object by saying:

Scanner sc = new Scanner (System.in);

  - where:

    - sc is the name of the Scanner object (this can be changed and is entirely up to the person who writes the code)
    - and System.in connects our object to standard input (the keyboard)

- The Scanner class has multiple built-in methods that we can access to help us read data; for example, **we can use sc.nextLine() to read in a line of String(s) (spaces included)**

- Other methods include:

  - **nextBoolean(): reads a boolean value from the user**
  - **nextByte(): reads a byte value from the user**
  - **nextDouble(): reads a double value from the user**
  - **nextFloat(): reads a float value from the user**
  - **nextInt(): reads an int value from the user**

- **next(): reads a complete token (String) from the user. A complete token is preceded and followed by input that matches the delimiter pattern; in most instances, this is a space.**

- **Sample exercises:**

    1. **Write a Java program that prompts the user for their first and last name and then prints out a message: "Hello, [FIRST NAME, LAST NAME]!"**

       <u>Sample Data (Input/Output)</u>:

       Enter your name: Jane Doe
       Hello, Jane Doe!

Reading and Writing Using Files

- As mentioned, we use a Scanner to read data, whether it comes from the keyboard or a file. **To read from a file, we only need to make one or two modifications from what we learned from reading from the keyboard.**

- First, since we are using files, we need to allow our program(s) to use them by including:

```
import java.io.File;  //import the File class
```

- Now we can add a **throws Exception** declaration to main and create the File object that we will be reading from:

```
File file= new File("[filename].txt");
```

- And create a Scanner as we have previously done, only changing our reference from the keyboard to the corresponding file object:

```
Scanner sc = new Scanner(file);
```

- Remember to **create the .txt file you are trying to read from, and make sure it is in the correct directory, or specify the file's path when creating the File object**; otherwise, you'll get a FileNotFoundException when you run your code.

- Writing to a file is slightly less work because we don't have to manually create the .txt file before running the program; the program creates it for us.

- Like before, we need to import the PrintWriter class to use it:

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

- We will use the PrintWriter class to create our file:

PrintWriter pw = new PrintWriter(“[filename].txt”);

- **Make sure the filename you are trying to create does not already exist in the directory you are writing to; otherwise, you will overwrite the file and lose your original data.**

- Now we can use the PrintWriter object to write, the same way that we wrote to the screen, only replacing System.out with our PrintWriter object, **i.e. pw.print(), pw.println(), pw.printf()**

- **Try modifying the programs you may have previously done so they write to both the screen AND a file.**

Closing Files

- When you are finished reading from or writing to a file, close the corresponding Scanner and/or PrintWriter:

sc.close();
pw.close();

- Closing a PrintWriter ensures that it writes any remaining output to the file.

Chapter 5: Conditionals and Logic

If and if-else statements

- An if statement specifies a statement(s) to be executed only if a particular boolean expression is true.
- An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
- An if statement can be followed by an else if statement to specify a new condition if the first condition is false. If you choose to include an else if statement, it **MUST COME BEFORE** the else statement.

- Syntax:

    - **if(condition1) {**
      //statement(s) to be executed if condition1 is true
      }
      else if(condition2) {
      //statement(s) to be executed if the condition1 is false and condition2 is true
      }
      else {
      //statement(s) to be executed if the condition1 is false and condition2 is false
      }

- **Sample exercises:**

    1. **Write a Java program that prompts the user for a number and prints whether it is even or odd.**

       <u>Sample Data (Input/Output)</u>:

       Enter number: 104
       104 is an even number.

    2. **Write a Java program that prompts the user for three numbers and prints out the largest of the three.**

       <u>Sample Data (Input/Output)</u>:

       Enter three numbers: 78 25 87
       The largest number is: 87

Switch Statements

- A switch statement lets a program select between several possible actions based on the value of a variable or expression.

Traditional Switch Statement

```
switch (choice) {
   case 1:
      System.out.println("Option 1");
      break;
   case 2:
      System.out.println("Option 2");
      break;
   case 3:
      System.out.println("Option 3");
      break;
   default:
      System.out.println("Invalid option");
}
```

- The switch expression compares its value with each case, and when it finds a match, it executes the statements for that case.
- **break** exits the switch statement and prevents execution from continuing into the next case.
- **default** executes when none of the cases match.

Rule Switch/Arrow Notation (->)

- Java also allows cases to use the **arrow (->) syntax**:

```
switch (choice) {
   case 1 -> System.out.println("Option 1");
   case 2 -> System.out.println("Option 2");
   default -> System.out.println("Invalid option");
}
```

- The code to the right of -> executes when that case matches, so a break statement is **not needed** with the arrow syntax.
- Multiple case values can share the same rule:

```
switch (choice) {
   case 1, 2 -> System.out.println("Option A");
   case 3, 4 -> System.out.println("Option B");
   default -> System.out.println("Invalid option");
}
```

- **Sample Exercise:**

    1. **Write a Java program that prompts the user to enter a number from 1–3 and uses a switch statement to print:**

        - **1 → "Red"**
        - **2 → "Green"**
        - **3 → "Blue"**
        - **Any other value → "Invalid choice"**

Nested if-else statements

- Nested if statements mean an if statement inside an if statement

- Syntax:

    - ```
      if (condition1) {
          //statement(s) to be executed if condition1 is true
          if (condition2) {
              //statement(s) to be executed if condition2 is true
          }
      }
      ```

- **Sample exercises:**

    1. **Write a Java program that prompts the user for a year and prints whether that year is a leap year. (*Note: Leap Years are any year that can be evenly divided by 4. A year that is evenly divisible by 100 is a leap year only if it is also evenly divisible by 400. For example, 1700, 1800, and 1900 are not leap years, but 1600 and 2000 are.)***

        <u>Sample Data (Input/Output)</u>:

        Enter year: 2016

        2016 is a leap year.

Logical and Relational Operators

- Boolean expressions (evaluate to true/false):
    - Less than: **a < b**
    - Less than or equal to: **a <= b**
    - Greater than: **a > b**
    - Greater than or equal to: **a >= b**
    - Equal to: **a == b**
    - Not Equal to: **a != b**
      ```

Integer and Logical Values

- We can include integers in our boolean expressions and determine whether or not they are true or false. For instance, if we were to print out:
    - System.out.print(6 > 5), we would see a result value of ‘true’
    - while System.out.print(6 < 5) or System.out.print(6 == 5) would result in a value of ‘false’ for obvious reasons

- Logical operators allow us to combine or modify Boolean expressions.

Operator	Name	Description	Example
&&	AND	True only when **both** conditions are true	x > 5 && x < 10
||	OR	True when **at least one** condition is true	x < 5 || x > 10
!	NOT	Reverses the value of a Boolean expression	!(x > 5)

- For example:
    - int x = 8;
    - System.out.println(x > 5 && x < 10); // true
    - System.out.println(x < 5 || x > 10); // false
    - System.out.println(!(x > 5)); // false

Chapter 7: Loops

While Loops

- Allows code to be executed repeatedly based on a given Boolean condition (T/F). Needs two parts: **condition/test expression and increment/decrement/update expression**

- Syntax:

    - ```
      int i = 1;
      while(i <= 10) {
          //statement(s) to be executed while the condition is true
          i++;
      }
      ```

    - where:

        - i <= 10 is a boolean expression. I**f the condition evaluates to true, we execute the loop body and go to the update expression.** Otherwise, we will exit from the while loop without reaching the statements inside the body.
        - And i++ increments the variable i

- How many times does this loop run? What would happen if you forgot to include i++;? Try writing code that uses a while loop like this.

Sentinel-Controlled Loops

- A **sentinel value** is a special value that indicates when input should stop.
- A sentinel-controlled loop continues reading and processing data until the sentinel value is entered.
- For example:
    - ```
      System.out.print("Enter a number (0 to stop): ");
      int number = stdin.nextInt();

      while (number != 0) {
        System.out.println("You entered: " + number);

        System.out.print("Enter another number (0 to stop): ");
        number = stdin.nextInt();
      }
      ```

- Notice that input is read **before the loop** and again **inside the loop** so that the condition can eventually become false.

Reading Until End-of-File (EOF)

- When reading from a file, you may not always know how many values are in the file. You can use a while loop to keep reading as long as more input is available.

- Scanner methods such as **hasNextInt()** and **hasNextDouble()** can be used to check whether another value is available before reading it.

- Example:

```
while (sc.hasNextInt()) {

   int number = sc.nextInt();

   System.out.println(number);

}
```

- The loop stops when there are no more integers to read from the file.

Do-While Loops

- A do-while loop is similar to a while loop, except it checks the condition after the loop body executes.

- Because of this, a do-while loop will always execute at least once.

- Syntax:

```
int i = 1;

do {
   System.out.println(i);
   i++;
} while (i <= 10);
```

- Key Difference:
    - **while** checks the condition **<u>before</u> executing the loop body**.
    - **do-while** checks the condition **<u>after</u> executing the loop body**.

- Therefore, a while loop may execute zero times, while a do-while loop executes at least once

- **Sample exercises:**

    1. **Write a Java program that prompts the user for age, prints "Sorry, you are ineligible to vote.", and increments the age until the user is eligible to vote, then prints "Congratulations, you are eligible to vote!" once they are 18+.**

       <u>Sample Data (Input/Output)</u>:

       Enter your age: 16

       Sorry, you are ineligible to vote.

       Sorry, you are ineligible to vote.

       Congratulations, you are eligible to vote!

    2. **Write a Java program that prompts the user to input an integer and then outputs the number with the digits reversed. For example, if the input is 12345, the output should be 54321.**

       <u>Sample Data (Input/Output)</u>:

       Enter a number: 12345

       Reversed of 12345 is: 54321

For Loops

- Unlike a while loop, a for statement includes **initialization, a condition, and an increment/decrement** in one line.

- Syntax:

    - **for (int i = 1; i <= 10; i++) {**
      //statement(s) to be executed while the condition is true
      }

    - where:
        - int i = 1; is the initialization
        - i <= 10; is a boolean condition
        - And i++ increments the variable i

- How many times does this loop run? Try writing code that uses a for loop like this.

Using a Header Value

- Sometimes a file contains a header value that tells us how many data values or records follow.

- Since the number of repetitions is known, you can use a for loop to process the data.

- Example file:

```
4
10
20
30
40
```

- Example:

```
int count = sc.nextInt();

for (int i = 0; i < count; i++) {
   int number = sc.nextInt();
   System.out.println(number);
}
```

- The header value is read before the loop and determines how many times the loop executes.

- **Sample exercises:**

1. **Write a program to print numbers from 10 to 100 in increments of 10.**

   <u>Sample Data (Output only)</u>:

   10 20 30 40 50 60 70 80 90 100

2. **Write a program that prompts the user to input a positive integer. It should then print the multiplication table of that number. (0 - 12)**

   <u>Sample Data (Input/Output)</u>:

   Enter a number: 5
   Multiplication Table of 5
   5 x 0 = 0
   5 x 1 = 5
   ....
   5 x 12 = 60

Choosing the Appropriate Loop

Situation	Typical Loop	How Repetition Ends
Number of repetitions is known	for	Counter reaches its limit
File provides a header/count	for	Counter reaches the header value
Number of repetitions is unknown	while	Condition becomes false
Special value indicates the end	while	Sentinel value is entered
Reading all values from a file	while	End-of-file is reached
Loop body must execute at least once	do-while	Condition becomes false

More Complex Loops

- **Nested loops**: if a loop exists inside the body of another loop, it's called a nested loop.
- **Loops with more than one condition**: a condition is a Boolean expression and can have more than one part.
- For example, to vote, you need to be 18+ AND an American citizen. In this case, a person is eligible to vote only when both requirements are true. The same applies to loop conditions.
    - When using AND (&&), both parts of the condition must be true to evaluate to true
    - When using OR (||), only one part needs to be true to evaluate to true
- **Sample exercises:**

    1. **Write a Java program that prints out each day of the week for a month (approximately 4 weeks).**

       <u>Sample Data (Output)</u>:

       Week: 1
       Day: 1
       Day: 2
       Day: 3
       ....

Week: 2
Day: 1
Day: 2
Day: 3
....
....

2. **Write a Java program that prompts the user for a number of rows and then prints out a half pyramid of asterisks ('*') with the same number of rows.**

   <u>Sample Data (Input/Output)</u>:

   Enter the number of rows: 5

   *
   * *
   * * *
   * * * *
   * * * * *

3. **Write a Java program that prompts the user for a number less than 100 and prints all even numbers from that number up to and including 100.**

   <u>Sample Data(Input/Output)</u>:

   Enter a number: 85

   86 88 90 92 94 96 98 100

Tracing Code

When tracing code, follow the program **one** statement at a time and **keep track of how variable values change**.

- Evaluate expressions and conditions **in the order they are executed**.
- Determine which statements execute.
- **Keep track of any changes to variable values**.
- For loops: keep track of **each iteration** and when the loop condition becomes false.
- Continue through the program using the **updated values**.
- Determine the exact output produced by the code.

Chapter 9: Strings and Things

String Representation

- Strings are used for storing text.
- A String variable contains a collection of characters surrounded by double quotes; it can be declared using a string literal, such as:

String greeting = “Hello”;

- Like arrays, **Strings are immutable;** whenever a change is made to a String, an entirely new String is created.
- Methods used to obtain information about a String are known as **accessor methods**. One accessor method you can use with Strings is **length()**, which returns the number of characters in the String.

System.out.println(“The length of String greeting is: “ + greeting.length());

What should this print out? How long is our String greeting?

- **Sample exercises:**

1. **Write a Java program that prompts the user for their first and last name, prints out the length of both, and then greets them with a message.**

   <u>Sample Data (Input/Output)</u>:

   Enter first name: John
   Enter last name: Doe

   The length of first name: John is 4
   The length of last name: Doe is 3

   Hello, John Doe!

String Methods

- As mentioned earlier, when we create a String, we have access to methods that perform certain operations on Strings.

- These methods include (but are not limited to):
  - **char charAt(int index)**
    - **Parameters:**
      index - the index of the char value

        - **Returns:**
            the char value at the specified index of this string. The first char value is at index 0
        - **Throws:**
            IndexOutOfBoundsException - if the index argument is negative or greater than or equal to the length of this string
        - Example: what does **greeting.charAt(2)** return?
          *** Note: remember that Strings are char arrays and array indexes start from 0**

    - **int compareTo(String anotherString)**
        - **Parameters:**
            anotherString - the String to be compared
        - **Returns:**
            the value 0 if the argument string is equal to this string;
            a value less than 0 if this string is lexicographically less than the string argument;
            and a value greater than 0 if this string is lexicographically greater than the string argument

    - **int compareToIgnoreCase(String str)**
        - **Parameters:**
            str - the String to be compared
        - **Returns:**
            a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations

    - **String concat(String str)**
        - **Parameters:**
            str - the String that is concatenated to the end of this String
        - **Returns:**
            a string that represents the concatenation of this object's characters followed by the string argument's characters
          ***note: we can also use the + operator between strings to concatenate them.**

    - **boolean equals(Object anObject)**
        - **Parameters:**
            anObject - The object to compare this String against
        - **Returns:**
            true if the given object represents a String equivalent to this string, false otherwise

- **boolean equalsIgnoreCase(String anotherString)**
  - **Parameters:**
    anotherString - The String to compare this String against
  - **Returns:**
    true if the argument is not null and it represents an equivalent String ignoring case; false otherwise

- **int indexOf(int ch)**
  - **Parameters:**
    ch - a character (Unicode code point)
  - **Returns:**
    the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur

- **int indexOf(int ch, int fromIndex)**
  - **Parameters:**
    ch - a character (Unicode code point)
    fromIndex - the index to start the search from
  - **Returns:**
    the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur

- **String substring(int beginIndex)**
  - **Parameters:**
    beginIndex - the beginning index, inclusive
  - **Returns:**
    the specified substring
  - **Throws:**
    IndexOutOfBoundsException - if beginIndex is negative or larger than the length of this String object

- **String substring(int beginIndex, int endIndex)**
  - **Parameters:**
    beginIndex - the beginning index, inclusive
    endIndex - the ending index, exclusive
  - **Returns:**
    the specified substring
  - **Throws:**
    IndexOutOfBoundsException - if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex

- **See the Java API for full list of String class methods:** https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/String.html

Integer and Character Variables

- A char is a single character, such as a letter, a digit, a punctuation mark, a tab, a space, or something similar. A char literal is a single character enclosed in single quote marks like this:

char c = 'a';

- A variable whose type is char is stored as a Unicode character in 16 bits (2 bytes).

- We can perform computations and compare equivalence between char and int variables because of char ASCII values.
- Consult the table:

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- The ASCII chart shown lists all of the printable characters.
- There are additional characters (whitespace, control codes) that are represented by codes 0-31.

- Note that there are different codes for “A” (decimal 65) and “a” (decimal 97).

- The computer has no idea what an “A” or an “a” is, or that they represent the same letter.

- This is important to remember – a computer only understands electronic signals, and has no knowledge of our alphabet.

- Some characters are hard to type. For these, Java provides escape sequences. This is a backslash followed by an alphanumeric code. For instance, '\n' is the newline character. '\t' is the tab character. '\\' is the backslash itself.

- The following escape sequences are defined:

    - \b - backspace
    - \t - tab
    - \n - new line
    - \r - carriage return
    - \" - double quote, "
    - \' - single quote, '
    - \\ - backslash, \

- **Sample exercises:**

    1. **Write a Java program that prompts the user to enter a String, counts the number of vowels (a, e, i, o, u) in the String, and prints out the result.**

       <u>Sample Data (Input/Output)</u>:

       Enter a String: good morning starshine, the earth says hello!

       The number of vowels in this String is: 13

    2. **Write a Java program to check whether the first two characters of a given string are present at the end of the string.**

       <u>Sample Data (Output only)</u>:

       The given string is: educated
       The first two characters appear at the end: true

Chapter 4/6: Methods (Call By Value)

A method in Java is a block of code that performs specific actions when called.

For instance, if you write instructions to draw a circle in a method, it will perform that task. You can pass values (parameters) into methods, and they are executed only when called. They are also referred to as functions.

The primary uses of methods in Java are:

- It allows code reusability (define once and use multiple times)
- It breaks a complex program into smaller chunks of code
- It increases code readability

Call by Value means calling a method with a parameter as a value. In this approach, the argument value is passed to the parameter.

In call by value, modifications to the parameter passed do not affect the caller's scope (where the method was called), unless the method returns a value.

Void Methods

- Void methods mean that the method does not return a value (e.g., main())

Value Methods

- Value methods have a return type (int, double, String, boolean, etc.) and must return a value of that type.

- **Sample exercises:**

1. **<u>Trace the following program</u>:**

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

        int a = 30;
        int b = 45;
        System.out.println("Before swap (in main), a = " + a + " and b = " +
                                    b);

        // Invoke the swap method
        swap(a, b);
        System.out.println("After swap (in main), a = " + a + " and b is " + b);
```

```
        //Invoke the modify method
        b = modify(a, b);
        System.out.println("After modify (in main), a = " + a + " and b is " +
                                b);
}

public static void swap(int a, int b) {

        System.out.println("Before swap (in method), a = " + a + " b = " + b);

        // Swap n1 with n2
        int c = a;
        a = b;
        b = c;

        System.out.println("After swap (in method), a = " + a + " b = " + b);
}

public static int modify(int a, int b) {

        System.out.println("Before modify (in method), a = " + a + " b = " +
                                b);
        // modify values
        a = b * 2;
        b = a * 3;

        System.out.println("After modify (in method), a = " + a + " b = " + b);
        return a;
}
```

Chapter 8: Arrays

One-Dimensional Arrays

- Arrays store multiple values in a single variable, instead of declaring separate variables for each value.

- To declare a one-dimensional array, we define the variable type with square brackets:

int [] numbers

- We have now declared a variable that holds an array of integers. To insert values into it, we can use an array literal - place the values in a comma-separated list inside curly braces:

int [] numbers = {10, 20, 30, 40};

- If we know the intended size of our array beforehand (or at least the maximum threshold we are confident the array won't exceed), we can initialize its size first and add values later. We can do this by using the new keyword and specifying the size in square brackets:

int [] numbers2 = new int [SIZE GOES HERE];

    - In this case, we want to create an array of the same size as the previous example, so we would declare it as such:

int [] numbers2 = new int [4];

- We can access a specific array element by referring to its index. **IMPORTANT TO REMEMBER: ARRAY INDEXES START AT 0**

- The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types).

- Let's populate the numbers2 array with the same values as the numbers array.

- We want to put the same values in the same position:

    - where the value 10 is in the first position
    - the value 20 is in the second position
    - the value 30 is in the third position
    - and the value 40 is in the fourth position

```
numbers2[0] = 10;
numbers2[1] = 20;
numbers2[2] = 30;
numbers2[3] = 40;
```

What would happen if we changed 3 to 4, or tried to set numbers2[4] = 50? Why would that not work?

- To find out how many elements an array has, we can use the length property.
    - **numbers2.length**
    - What should System.out.println(numbers.length); print out?

- Each element in the array is accessed via its index. The index begins at 0 and ends at (total array size) - 1. All the elements of the array can be accessed using a for Loop.

```
// accessing the elements of the specified array
for (int i = 0; i < numbers2.length; i++) {
        System.out.println("Element at index " + i + " : "+ numbers2[i]);
}
```

Using Arrays in Methods

- You can pass arrays to a method just like normal variables.
.
- When we pass an array to a method as an argument, actually the address of the array in memory is passed (reference). Therefore, **any changes to this array in the method will affect the array.**

- **Sample exercises:**

1. **Write a Java program that reads data (integers) from an external data file (input file) with no more than 100 integers, one per line, into an integer array: numbers, and saves the number of integers read into an integer: size. Print out the array (up to n) in main.**

   Write a method: average() that accepts an integer array: nums and an integer: n, which will calculate and return the average (hint: the average SHOULD be a double; how can we ensure that we get the accurate number and avoid integer division?) of the n numbers in the array and print out this value in main.

   Write another method, stats(), that accepts an integer array nums and an integer n, and prints how many numbers are even, how many are odd, how

many are greater than or equal to the average, and how many are below the average.

<u>Sample Data (Input/Output)</u>:

(Input file)

5 11 2 4 32

(Output)

Array: 5 11 2 4 32
Average of array: 10.8

Number of even numbers: 3
Number of odd numbers: 2
Number of numbers equal to or greater than average: 2
Number of numbers below average: 3

2. **Write a Java program that has an integer array declared and initialized in the main method. Write a method printArray() that prints the array, and two separate methods, arrayMax() and arrayMin(), that accept an integer array as a parameter and calculate and return the maximum and minimum values of the given array, respectively. Call each of these methods in main and print all output to the screen.**

   <u>Sample Data(Output only)</u>:

   Array values: 45 12 48 53 55
   Maximum value in the array is: 55
   Minimum value in the array is: 12

3. **Write a Java program that has an integer array declared and initialized in the main method. Write a method that prints the array, and another method called powerOfTwo() that accepts an integer array as a parameter and updates each element to its original value squared. Call both methods in main so your program prints the original array, modifies the array, and then prints the modified array.**

   <u>Sample Data (Output only)</u>:

   Array values (original): 2 4 6 8 10
   Array values (modified): 4 16 36 64 100

Parallel Arrays

- Parallel Arrays: multiple arrays of the same size such that the i-th element of each array is closely related and all i-th elements together represent an object or entity. An example of a parallel array is two arrays that represent x and y coordinates of n points.

- Below is an example (***note: the arrays do not have to be of the same data type to be parallel; we could have a set of parallel arrays that are of Strings and corresponding integers**):

String [] name = {“Usagi Tsukino”, ”Ami Mizuno”, “Rei Hino”, “Makoto Kino”, “Minako Aino”};

String [] identity = {“Moon”, “Mercury”, “Mars”, “Jupiter”, “Venus”};

String [] guardianOf = {“Love and Justice”, “Water and Wisdom”, “Fire and Passion”, “Thunder and Courage”, “Love and Beauty”};

name:

0	1	2	3	4
Usagi Tsukino	Ami Mizuno	Rei Hino	Makoto Kino	Minako Aino

identity:

0	1	2	3	4
Moon	Mercury	Mars	Jupiter	Venus

guardianOf:

0	1	2	3	4
Love and Justice	Water and Wisdom	Fire and Passion	Thunder and Courage	Love and Beauty

This way, we can easily store the information, and the first index of each array corresponds to the data for that person.

- **Sample exercises:**

    1. **Write a Java program that prompts the user for a size (number of students/number of grades) and then declares an integer array of that size. Then prompt the user to enter integer grades (0 - 100) to populate the array (hint: use a for loop). Create a separate method called letterGrade() that accepts an integer array and returns a new parallel String array populated with the equivalent letter grade. (hint(s): parallel arrays have the same size; once again, use a for loop, this time you also need if/else if/else blocks) Print out the entire array of both integer grades and letter grades (you can create a separate method for this).**

Letter Grade	Percent Grade	4.0 Scale
A+	97-100	4.0
A	93-96	4.0
A-	90-92	3.7
B+	87-89	3.3
B	83-86	3.0
B-	80-82	2.7
C+	77-79	2.3
C	73-76	2.0
C-	70-72	1.7
D+	67-69	1.3
D	65-66	1.0

(Use the above table to help you determine the necessary condition(s); use F as the letter grade for any integer lower than 65)

<u>Sample Data (Input/Output)</u>:

Enter number of students: 4
Enter student 1 grade: 79
Enter student 2 grade: 95
Enter student 3 grade: 96
Enter student 4 grade: 83

Student grades:
Student 1: 79 Final Grade: C+
Student 2: 95 Final Grade: A
Student 3: 96 Final Grade: A
Student 4: 82 Final Grade: B-

Searching

- A key is a value that you are looking for in an array.

- The simplest type of search is the **sequential (or linear) search**.

- In a sequential search, each element of the array is compared to the key, in the order it appears, until the desired element is found.

- If you are looking for an element near the front of the array, a sequential search will find it quickly.
- The more data that must be searched, the longer it will take to find the data that matches the key.

- Consider this method, which will search for a key integer value:

    - If found, the index (subscript) of the first location of the key will be returned.
    - If not found, -1 will be returned.

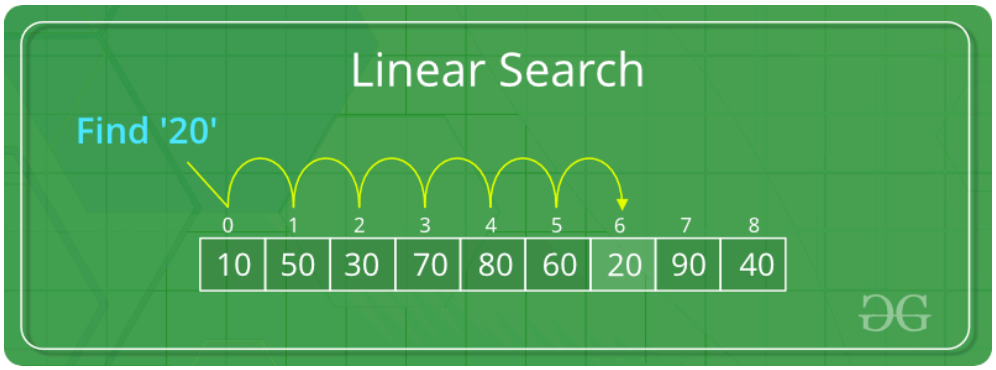
```
public static int search(int [ ] numbers, int key) {
        for (int i = 0; i < numbers.length; i++) {
                if ( numbers[i] = = key )
                        return index;  // We found the key and this index!
        }
        // If we get to the end of the loop, a value has not yet
        // been returned.  We did not find the key in this array.
        return -1;
}
```

- If the key value is found, the index (subscript) of the location is returned. This tells us that the return value x will be the first integer found such that:

numbers [x] = key

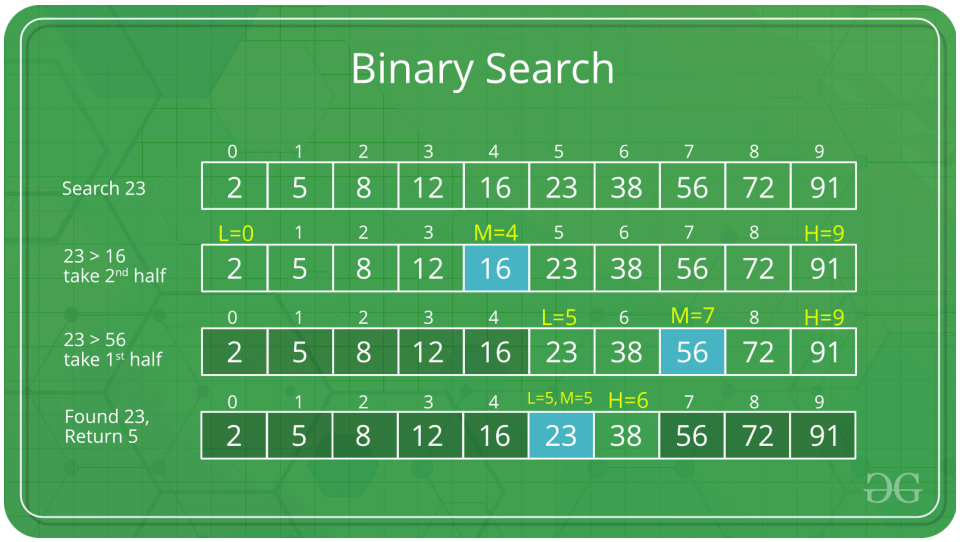
- There may be additional locations of the key in this array beyond this location.

- Types of searches:
    - **Linear search: the list or array is traversed sequentially, and every element is checked (sequential search!!)**



        - Start from the leftmost element of arr[] and, one by one, compare x with each element of arr[]
        - If x matches an element, return the index.
        - If x doesn't match any of the elements, return -1.

    - **Binary search:**



        - Search a sorted array by repeatedly dividing the search interval in half.
        - Begin with an interval covering the whole array.
        - If the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
        - Otherwise, narrow it to the upper half.
        - Repeatedly check until the value is found or the interval is empty.

```
public static int binarySearch(int arr[], int l, int r, int x) {
        if (r >= l) {
                int mid = l + (r - l) / 2;

                // if the element is present at the middle itself
                if (arr[mid] == x)
                        return mid;

                // if element is smaller than mid, then
                // it can only be present in the left subarray
                if (arr[mid] > x)
                        return binarySearch(arr, l, mid - 1, x);

                // else the element can only be present
                // in right subarray
                return binarySearch(arr, mid + 1, r, x);
        }

        // We reach here when element is not present in array
        return -1;
}
```

- Sample exercises:

    1. **Modify the previous exercise to use three parallel arrays: an array of Strings called firstName, an array of Strings called lastName, and an array of integers called grades, and populate them. Add a search method that allows the user to input a last name and a first name, prints the student's corresponding integer and letter grade if they are found in the database, and prints "Invalid -- Student not found" if they are not.**

       <u>Sample Data (Input/Output)</u>:

       Enter student's last name: Wilkins
       Enter student's first name: Samantha-Eve

       Student's grade: 96 - A

       Enter student's last name: Sloane
       Enter student's first name: Rex

       Invalid -- Student not found

       Enter student's last name: Grayson
       Enter student's first name: Mark

       Student's grade: 79 - C+