Review: OOP in Java

Tuesday, May 31, 2022 7:48 AM

FOR THIS COURSE: any lecture content marked with four asterisks (****) WILL be on this course's Tests and the AP Exam and should be mastered!

FOR THIS COURSE: The `CODE` provided in lectures *may* have some errors that you, the student, will need to correct. This is by design to address a common misunderstanding of some students that a simple copy-n-paste from lecture notes may not be sufficient to learn the concepts. YOU MUST PRACTICE CODING!

NOTE: The Java Languages Specifications, as of version 8 (i.e. Java 1.8), is 768 pages in length and has 13 pages for the Table of Contents alone. This course's lecture notes reviewing what was covered in the prerequisite course (Programming in Java) and Syntax-Java Overview are a 50-page summary of the Java Language Specifications found at https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Define: **OOP** in Java is *Object Oriented Programming* using the **Java** language.
Define: **Java** is a one computer science (CS) language that follows a specific syntax (or grammar) that runs on a digital computer.

This is in contrast to other computer science languages such as a *Structured Programming* using the **Pascal** language (a 1970's high-level--or human-readable--language designed to teach programming) or *Assembly Programming* using the **X86 opcode language** (a very low-level language--difficult to read by humans--designed to manipulate a computer's CPU/hardware). There are several thousand computer languages...Java being one of the longest-running (1995) and most-used languages in use as of 2022 (top 5).

A. Brief Review & Example

Concept/Program	**Realization**
Class - description made by a programmer written in the Java language	*Object* - actual "thing" created from the class description (i.e. stored in the computer's memory after declaration and instantiation)

`public class Cake`
`{`
`    private int eggs;`
 	
`    public Cake()`
`    {`
`        eggs = 5;`
`    }`
 	
`    public void setEggs(int e)`
`    {`
`        eggs = e;`
`    }`
`}`
	`Cake wedding;          // declaration`
	`wedding = new Cake();  // instantiation`
	`wedding.setEggs(10);   // invoking setEggs() method`

To help "visualize" the relation between a *Class* and an *object*, we often draw diagrams using the Universal Modeling Language (UML). These diagrams shows the general features of a *Class* (for a UML *Class* Diagram) or the detailed values stored in an *object* (for a UML *object* Diagram) in an attempt to hide the details of the Java language syntax.

UML *Class* Diagram (i.e. before programming)	UML *object* Diagram showing values stored

`+ Cake`
`- int eggs;`
`+ Cake()`
`+ void setEggs(int e)`
	(after instantiation)
	`wedding------->`
	`+ Cake`
	`- int eggs = 5`
	
	(after invoking setEggs() method)
	`wedding------->`
	`+ Cake`

	- int eggs = 10

B. Sequence of a Program & Introduction to IDE's

```
                      |------------ runtime / logic errors -------|
                      |   |-syntax errors--|                      |
                      |   |                |                      |
Programmer -----> Source Code -------> Byte Code ---------> Java VM --------> Output
          (typing)      |     (compile)    |    (interpret)       | (hardware      |
                        |                  |                      | runs)          |
                  fileName.java      fileName.class        in JRE program,      monitor,
                 (human readable)  (computer readable)     browser, plug-in, speaker,network, etc.  B.
```

C. There are two distinct processes the computer takes when *converting* Source Code to *runnable* output:

a. Compile

   i. This is the process of converting Java S*ource Code* (what the programmer types in as a *fileName.java* file) to Java *Byte Code* (what the computer generates as a *fileName.class* file).

   ii. This compile process is actually run by another program called ***javac.exe***

      1) On Mr. Meinzen's computer, javac.exe can be found at : C:\Program Files\Java\jdk1.8.0_291\bin\javac.exe.
      2) Most IDE's (including eclipse, Textpad, and others) will automatically use/call javac.exe when the user chooses to compile their Source Code.
      3) Programmers can use Command Line Interface (CLI) or console to type in the javac.exe command (or run the javac.exe program). However, this is a topic that involves security permissions.

   iii. The compiler (javac.exe) has the following functions:

      1) Scans the programmers Source Code line-by-line starting at the top and "looks for" Syntax Errors (errors that don't follow the grammatical rules of Java as specified in the Java Language Specifications to be discussed later). These include (but not limited to):
         a) spelling errors including case-sensitive errors (`name` is different than `Name`)
         b) unmatched or missing parenthesis , curly-brackets
         c) missing semi-colons and commas
         d) misplaced reserved words
         e) undeclared identifiers (i.e. using an variable before declaring the variable)
            i) identifiers are the names chosen by the programmer for their Classes, variables, and method names
      2) If it finds any Syntax Errors, the compiler attempts to continues scanning for other possible Syntax Errors to generate a list of potential errors. ***However, the <u>first</u> Syntax Error is the ONLY error that the programmer can be confident is an actual error as the compile MAY falsely detect more errors after the first.***

b. Interpret/Run

   i. This is the process whereby the Java Virtual Machine (or JVM), executes the Byte Code (i.e. *fileName.class*) that was most recently compiled

   ii. The JVM is actually another program called ***java.exe [NOTE: This differs from the compiler, javac.exe, by excluding the 'c']***

      1) On Mr. Meinzen's computer, java.exe can be found at : C:\Program Files\Java\jdk1.8.0_291\bin\java.exe.
      2) Most IDE's (including eclipse, Textpad, and others) will automatically use/call java.exe when the user chooses to run their Byte Code.
      3) Programmers can use Command Line Interface (CLI) or console to type in the java.exe command (or run the java.exe program). However, this is a topic that involves security permissions.

Review: Full Example & use of IDE (eclipse)

Tuesday, May 31, 2022   10:19 AM

C. Full Example

Suppose we wanted the computer to simulate any coin but initially limited to the following list : dime, nickel, penny, and quarter.

1. We would generally first identify the common necessary features of every "real" coin (i.e. object) we could select. In other words, every coin listed **has a** *value* and a *name*. Hint: when considering every coin, try to identify the common "nouns" (in English) or "variables" (in math) which become " fields" (in programming).

2. After we identify what every coin **"has"**, we imagine what every coin should **"do."** Every coin should be manufactured (i.e. in programming this is referred to as **instantiated** or **constructed**) and we should be able to **get** any coin's *value*. Hint: when deciding what every coin "**does**", consider all the possible coin's "verbs" (in English) or "functions" (in math) which becomes "methods" (in programming). Usually the methods act on (or use or calculate) the field variables.

3. The third step is to start drawing a UML *Class* diagram where we identify the Class name, the field variables & their types, and the methods that are in common among every coin *object*.

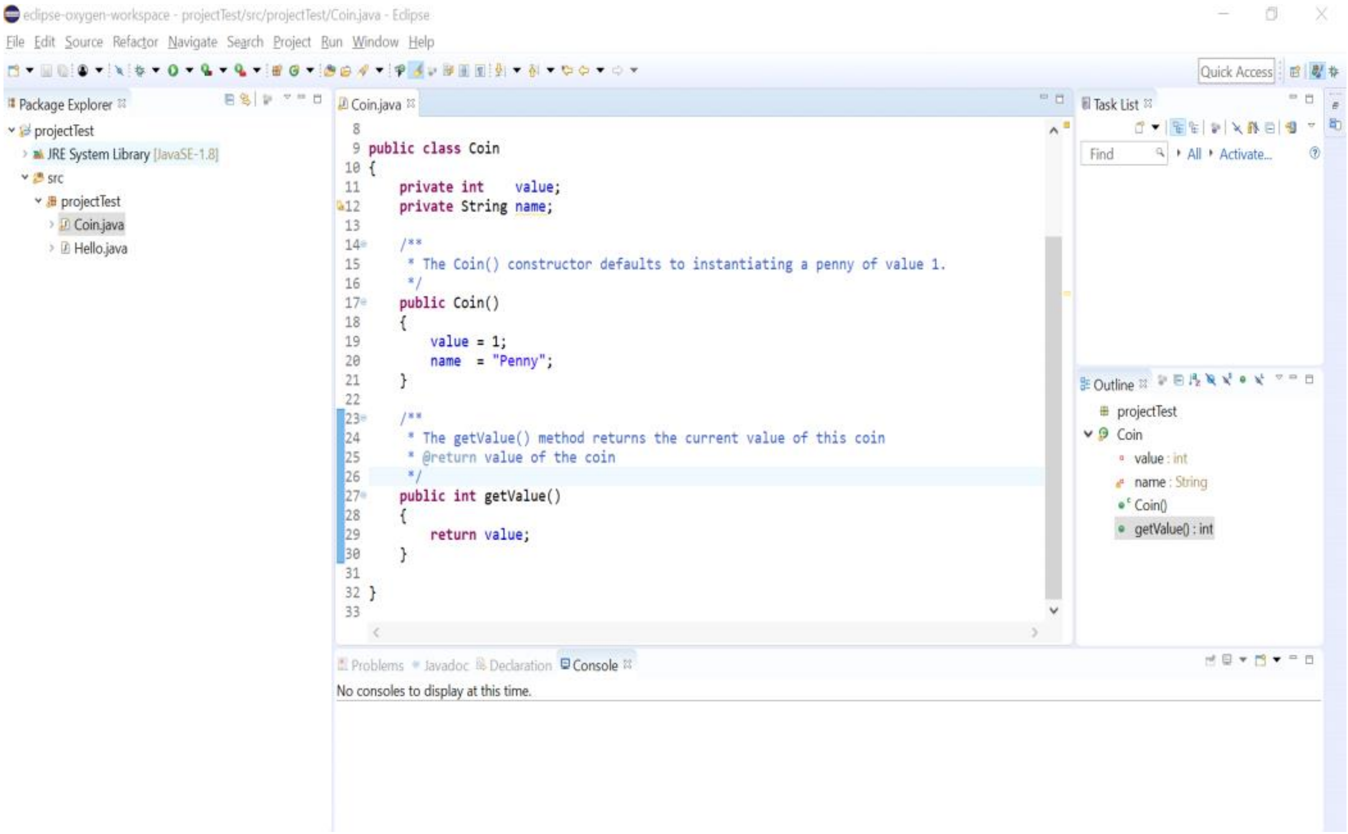
   a.

`+ Coin`	<--- class name will be "Coin" Note: the + and − are shortcuts for *public* and *private* attributes that are analagous to adjectives (in English) which will be discussed later when we add more classes.

`- int    value;`
`- String name;`
	<--- one *field variable* will be *value* that is an integer. <--- another *field variable* will be the name as text
	Notes:
	1. *int* is a Java primitive (i.e. built-in type) that means integer
	2. String is a Java class used for handling text, letters, or characters)

`+ Coin()`
`+ int getValue()`
	<---first function (or action) is the *constructor* method
	<---second function is the getValue() method…it will return an integer
	Notes:
	1. **constuctors MUST have the same name as the class**
	2. The purpose of the getValue() method is to represent the current value of the coin without changing the value.

4. The fourth step is to write, comment, and compile the class in Java using an IDE (integrated development environment) such as Eclipse or Netbeans. This may look like the following image: Note: the lower right corner has diagram that is a variation on UML *Class* Diagram.



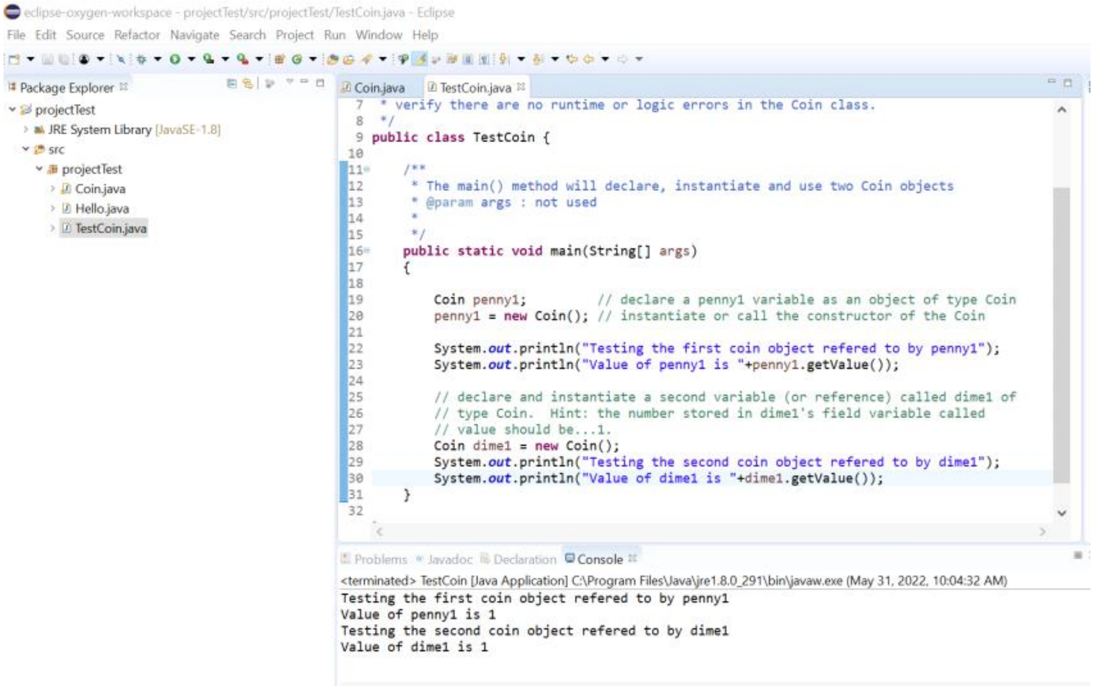
```

public class Coin
{
    private int    value;
    private String name;

    /**
     * The Coin() constructor defaults to instantiating a penny of value 1.
     */
    public Coin()
    {
        value = 1;
        name  = "Penny";
    }

    /**
     * The getValue() method returns the current value of this coin
     * @return value of the coin
     */
    public int getValue()
    {
        return value;
    }

}

```

No consoles to display at this time.

5. Up to this point, we have *defined* and *compiled* a class called **Coin** so that it doesn't have any Java syntax errors but we haven't actually "executed" or "tested" the class for any runtime or logic errors. To see if our Coin class actually "works" we need to *declare* and *instantiate* at least one Coin *object* or, better yet, many Coin *objects*. This can be done several ways but the most common way is to create a separate class called **TestCoin** with a specific method called *main()* that starts an application/program running. Traditionally, we do NOT create a UML diagram for this testing class.

    a. The goal is to declare, instantiate, and call (or use) at least two Coin objects. We will refer to these two objects as *penny1* and *dime1*.

    b.


eclipse-oxygen-workspace - projectTest/src/projectTest/TestCoin.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
projectTest
JRE System Library [JavaSE-1.8]
src
projectTest
Coin.java
Hello.java
TestCoin.java
Coin.java
TestCoin.java
Problems
Javadoc
Declaration
Console

```
 * verify there are no runtime or logic errors in the Coin class.
 */
public class TestCoin {

    /**
     * The main() method will declare, instantiate and use two Coin objects
     * @param args : not used
     *
     */
    public static void main(String[] args)
    {

        Coin penny1;         // declare a penny1 variable as an object of type Coin
        penny1 = new Coin(); // instantiate or call the constructor of the Coin

        System.out.println("Testing the first coin object refered to by penny1");
        System.out.println("Value of penny1 is "+penny1.getValue());

        // declare and instantiate a second variable (or reference) called dime1 of
        // type Coin.  Hint: the number stored in dime1's field variable called
        // value should be...1.
        Coin dime1 = new Coin();
        System.out.println("Testing the second coin object refered to by dime1");
        System.out.println("Value of dime1 is "+dime1.getValue());
    }

```

```
<terminated> TestCoin [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (May 31, 2022, 10:04:32 AM)
Testing the first coin object refered to by penny1
Value of penny1 is 1
Testing the second coin object refered to by dime1
Value of dime1 is 1
```

6. While the *penny1 object* seems to be printing out the correct information on the *console* (at bottom of screenshot), the programmer should be noticing a difference between a"real" dime (whose value should be 10) and the *dime1 object* whose value appears to be 1 rather than 10. This is a logic error which

the computer cannot correct as it does not understand a "reality" context. There are other errors but

a. The two UML *objet* Diagrams currently looks like the following:

penny1------>	+ Coin -int value = 1 -String name = "penny"			*dime1*------>	+ Coin -int value = 1 -String name = "penny"

7. There are several improvements that can be made but we will focus on two additional methods:

a. The first method is another *constructor* method with two *parameters* to help us instantiate a Coin that is different than a penny (i.e. has a value other than a 1).

```
public Coin(int v, String n)
{
    value = v;
    name  = n;
}
```

b. The second method is a `toString()` method that will make testing our code much easier by automatically converting all field variables to a `String` to be printed out by the `TestCoin` class.

```
public String toString()
{
    String temp = "This coin is a "+name+" and has value="+value;
    return temp;
}
```

c. The TestCoin main() method can now be written as

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

    Coin penny1;          // declare a penny1 variable as an object of type Coin
    penny1 = new Coin(); // instantiate or call the constructor of the Coin
    System.out.println(penny1.toString()); // call the toString() method on penny1

    // Declare and instantiate a dime object :
    //    the arguments 10 and "dime" are passed (automatically copied) to
    //    the parameters "n" and "v" in the Coin class to be assigned/stored
    //    in the dime1's field variables "name" and "value"
    Coin dime1 = new Coin(10,"dime");
    System.out.println(dime1.toString());
}
```

With console output:

```
This coin is a Penny and has value=1
This coin is a dime and has value=10
```

Review: Commenting & Expectations

Tuesday, May 31, 2022 10:42 AM

Three Java Commenting Options ****

(1) single line (or end-of-line) commenting -
   a. Syntax: `// ...anything you need to say`
   b. Result: Anything after // is ignored by compiler to end of line
   c. Usage:
      i. if used after statement, then explains usage/purpose of that statement
      ii. if used on its own line, then explains next few statements or code segment
      iii. if used on its own line, then put blank line above comment line

(2) multi-line commenting -
   a. Syntax: `/* ... anything you need to say on multiple lines... */`
   b. Result: Everything between /* and */ is ignored by compiler
   c. Usage:
      i. To explain complex segment of code that takes up more than one line
      ii. One its own lines, directly above code it explains
      iii. One or two blank lines above opening /*

(3) documentation commenting (JavaDoc utility)
   a. Syntax: `/** ... anything you need to say for use as "standard" code documentation... */`
   b. Result: Everything between /** and */ is ignored by compiler
   c. Usage:
      i. To explain feature of code for use in standard documentation (JavaDoc)
         1) Classes
         2) *Methods* (except main method)
         3) Fields
      ii. One its own lines, directly above code it explains
      iii. One or two blank lines above opening /**
      iv. May contain special tags that start with the @ character...explained later (example: `@author J. Meinzen`)
      v. javadoc utility creates standard HTML documentation from these comments

(4) Expectations for Programs to be scored:
   a. Hardcopy (printout) of each class unless specified otherwise.
   b. Each class must start on its one page unless multiple classes (complete files) can fit on a single page.
   c. New Currier font, 10 pt, no italics, no bold
   d. Appropriate and consistent **commenting and indenting**
   e. Be prepared to demonstrate to your teacher that your program (submitted version) runs **on the teacher's station**
   f. All identifiers (class names, variable names, method names) should be as meaningful in the context of the problem given.
      i. Exceptions for local variables names such as loop control variables and references for temporary use.

Review: Errors & Avoidance

Tuesday, May 31, 2022 10:55 AM

(1) Compiler / Syntax Errors
   a. breaking the "grammar" rules,
   b. easy to fix as compiler tells you
   c. common examples:
      i. forgetting a semicolon (this is not the same as adding extra semicolons!!)
      ii. misspelled words
      iii. incorrect case (capitalization)

(2) Runtime Errors
   a. unhandled exception
   b. usually due to not checking all possible values -- defensive programming
   c. results in a crash of program
   d. common examples
      i. nullPointer exception -- object declared but never initialized or instantiated
      ii. divide by zero
      iii. forgetting to initialize a variable either in constructor or at declaration

(3) Logic Errors
   a. you didn't think carefully enough -- you assumed you KNEW the answer
   b. may cause crash (hopefully) or just plain wrong answer/result
   c. most dangerous and difficult to track down
   d. usually can't prove correct unless using exhaustive testing or mathematical proof
   e. common examples
      i. off-by-one error -- especially using arrays since start at subscript 0
      ii. incorrect formula -- Area = 2.0 * Math.PI * radius
      iii. incorrect variable usage (worse yet, Area = 2 * Math.PI * radius)
      iv. roundoff errors - fundamental limits of data storage

(4) Avoiding Errors
   a. Syntax Errors
      i. Know Java syntax!
         1) The Java Language Specifications (i.e. Java Syntax) for Version 8 (2015) has a Table of Contents that is 39 pages long!
         2) The Java Language Specifications (i.e. Java Syntax) for Version 8 (2015) is 788 pages long!
         3) This course condenses 788 pages of technical specifications into about 40 pages of lecture notes for the first half of the 1st semester of AP CSA. Some errors and over-simplifications WILL occur. If in doubt, read the Java Language Specification online at https://docs.oracle.com/javase/specs/
      ii. Use style conventions (indenting, commenting, variable names, etc)!!
   b. Runtime Errors
      i. Know OOP concepts!!!
      ii. Passing argument variables to parameter variables in method calls!!!
      iii. **Habits!!!** (always initialize all variables, know Class types, know instantiations)
   c. Logic Errors
      i. Plan FIRST!!!!... 10 years of education and fast food has taught you to "get working right away and get it over with"...this will be your downfall. Be prepared for 6 weeks of boredom and lecture notes.
      ii. Know defensive programming!!!!
         1) assume you are WRONG

   2) Use easy, known test data to check your BASIC understanding of problem
   3) **Test "boundary" conditions...be paranoid!!!!**

iii. Know debugging techniques!!!!
   1) Pinpoint -- determine which methods your program is running and in what order (don't just assume)
   2) Snapshot -- determine variable/object values at various times
   3) Use either System.out.println()
   4) **Learn & Use the IDE debugger**

Review: Hardware

Tuesday, May 31, 2022 11:15 AM

1. Basic Concepts:
    1. Digital computers are fundamentally made up of 2-state switches (think of a light switch that is either on or off) and something (CPU) that flips the states of the switches (either from on to off or off to on)
    2. At the lowest level, all digital computers store two states (or values)…0 and 1 (or on/off). These two options are stored in various ways as binary digits (i.e. *bits*).
    3. *Bits* can be grouped together to represent more than 2 options.
        a. One bit has 2 options (or 2 states) that can be described variously as 0 or 1, off or on, false or true, etc.
        b. Two bits have 4 options (or 4 states) that can be 00, 01, 10, or 11 (or both off, first off & 2nd on, first on & 2nd off, or both on)
        c. Three bits have 8 options (or 8 states)
        d. The mathematical relationship to the number of bits and the number of options : $2^{(\text{\# of bits})}$ = # of options
        e. This relationship can also be expressed as $\log_2$(# of options) = # of bits
        f. Groups of 8 bits is called a *byte* which has 2^8 = 256 states
    4. Historically, one byte has been used to represent (or map or quantify)
        a. A single *character* (or letter or key) on the most commonly-used keyboards
        b. A shade of grey on printers or monochrome screens (0 being black, 128 being middle grey, and 255 being white)
        c. A volume level on a speaker.
    5. Over time, various groupings of bits and bytes have been *encoded* to represent any type of data that is stored and manipulated by computers and digital devices. The speed of the flipping by the CPU and the number of bits has increased at an exponential rate over the last several decades…traditionally known as "Moore's Law".

2. Diagramatically, most digital computers have the following block diagram:

		External Memory (or Long-term Memory) • Hard drive or SSD • Flash drive • DVD, BluRay • Etc.		
		| |		
Input Devices • Keyboard • Mouse • Microphone • etc	----->	CPU ("brains") Central Processing Unit • X86 • M1 • Etc.	------->	Output Devices • Monitor • Printer • Speaker • etc
		| |		
		Internal Memory (or Short-term Memory) • registers • Cache (L1,L2,etc) • RAM • Etc.		

3. Data, Rates, and Realizations

Data : size of memory	**Data Realizations**
(from fewest states to greatest number of states) 1. 1 bit (0 or 1, off / on) 2. 1 Byte = 8 bits = 256 states, options, values 3. 1 kilobyte (KB) = 1024 bytes (sometimes 1000 bytes) 4. 1 megabyte (MB) = 1024KB 5. 1 gigabyte (GB) = 1024MB 6. 1 terabyte (TB) = 1024GB=8796093022240 bits	(i.e. human contexts or measurement) 1. 1bit = light switch, pit/no pit burned in DVD by laser 2. 1Byte is about single keyboard character or letter 3. 1KB is about half of a page of typewritten letter 4. 1MB is about the number of pixels on a typical computer monitor (1024 rows x 1024 columns resolution) a. 1MB is about 500 typewritten pages = 1 ream of paper 5. 1GB is about the amount of storage needed for about 1 hour of DVD video a. 1GB is 1000 reams of paper filling about 1 room of filing cabinets 6. 1TB is about 200 full length BlueRay movies a. 1TB is about 1000 rooms filled with filing cabinets. About the number of pages in a very large/national library

Rate : speed that CPU can flip bits (slowest to fastest) 1. 1 hertz (Hz) = 10^0 = 1 flip/second 2. 1 kiloHertz (KHz) = 10^3 = 1000 flips / second 1 megaHertz (MHz) = 10^6 = 1000000 flips / second 3. 1 gigaHertz (GHz) = 10^9 = 1,000,000,000 flips / second	**Speed Realizations (**"flip" may be a single bit or group of bits or bytes or instructions). 1. 1Hz is about the speed a human will flip a light switch on and off. a. 30Hz is about the speed at which a human will detect flickering on a screen 2. 1KHz is about the speed at which a human ear will hear a Soprano C (High C) 3. 1MHz is about the speed that a image on a monitor(1MB memory) can change within 1 second. In other words, to avoid flickering, a CPU must have at least a 30*1MHz=30MHz rate to update images on a monitor. 4. 1-5GHz is about the speed of your WiFi communications in a local network (i.e. among the devices in your home). Note: the human eye perceives color within the GHz range but human senses are not digital (i.e discrete) but, rather, analog (i.e. continuous).

Review: OOP: Concepts & Terminology

Tuesday, May 31, 2022 12:26 PM

Object Oriented Programming has 5 distinguishing Principles that make it a powerful Computer Science (CS) paradigm:

1. Divide & Conquer
    a. Break overall problem down into small, manageable tasks while modeling the "real world"
        i. Example: a PokerGame can be broken down into the following classes: Card, Deck, Player, Rules, UserInterface (i.e. Swing Graphics), etc.
    b. Allows for multiple programmers to solve one problem.
2. Encapsulation
    a. Each object "knows"
        i. what the object **IS** -- field variables
        ii. what the object can **DO** -- methods
        iii. how the object **INTERACT** with other objects
            1) data hiding = private
            2) method interfacing = public
3. Generality
    a. also known as "can be applied to a lot of situations or problems"
    b. A *Class* is a Type of problem -- the class solves a general problem
        i. Example: the quadratic function is a class because it solves a type of problem (equations of the form $ax^2 + bx + c = 0$)
    c. An *object* is a specific instance of a problem -- $2x^2 -3x +6 = 0$ is an instance of a quadratic equation
4. Extensibility
    a. The ability to add or include features easily
    b. Two types of *Class* relationships:
        i. Association
            1) also known as a "**has a**" relation**** . A Deck has Cards. Therefore, we can declare and instantiate as many (usually 52) Card objects by declaring objects from the Card class to use as needed.
            2) These objects are simply declared as **field variables.**
            3) Used for "code reuse"--why create another class when we can just "have" another object described by an already existing class.

        ii. Inheritance
            1) also know as an "**is a**" relation****. A Tiger is a Mammel. Therefore class `Tiger extends Mammal`
            2) allows for a "hierarchy" to organize classes into useful categories.
                a) Example: animal classification system (a Tiger **is a** Mammal) , Java Foundation Classes (JFC) where JButton is a Component; JLabel is a Component.
                b) extended classes
                    i) inherit all the abilities (fields & methods) of the **super** or ancestor classes
                    ii) may overload ancestor methods (polymorphism)
                    iii) may write additional methods and/or fields as needed
            3) When a class needs to solve a more-detailed question/problem than another class, then inheritance (i.e. extending) may be useful. For example, if I alredy have a class called Rectangle but I now want a class that deals only with

Squares, I might write `public class Square extends Rectangle` and make sure that both the length and width of a rectangle are restricted to the same value (i.e. the single side of a square).

4) A specialized version of inheritance is an ***Interface*** where a class can implement some other programmer's "agreed-upon" methods. The other programmer describes what needs to be written (i.e. method header) but not how to do it (i.e. method body).
    a) `class Poker ... implements ActionListener` forces the Poker programmer to write a method called `public void actionPerformed(ActionEvent e) {}` but doesn't say how or what to write inside the method body.

5. Abstraction : the overarching principle of the above 4
    a. "What makes a human different than a computer"? Humans are natural at abstraction!!
    b. Definition: Discovering the essential features (nouns & verbs) of a class (fields & methods) in order to solve a question/problem. This may entail several "layers" of abstraction (i.e. multiple related classes) and is best learned through experience (i.e. failure many times)

Review: Java Keywords ****

Wednesday, June 1, 2022 9:26 AM

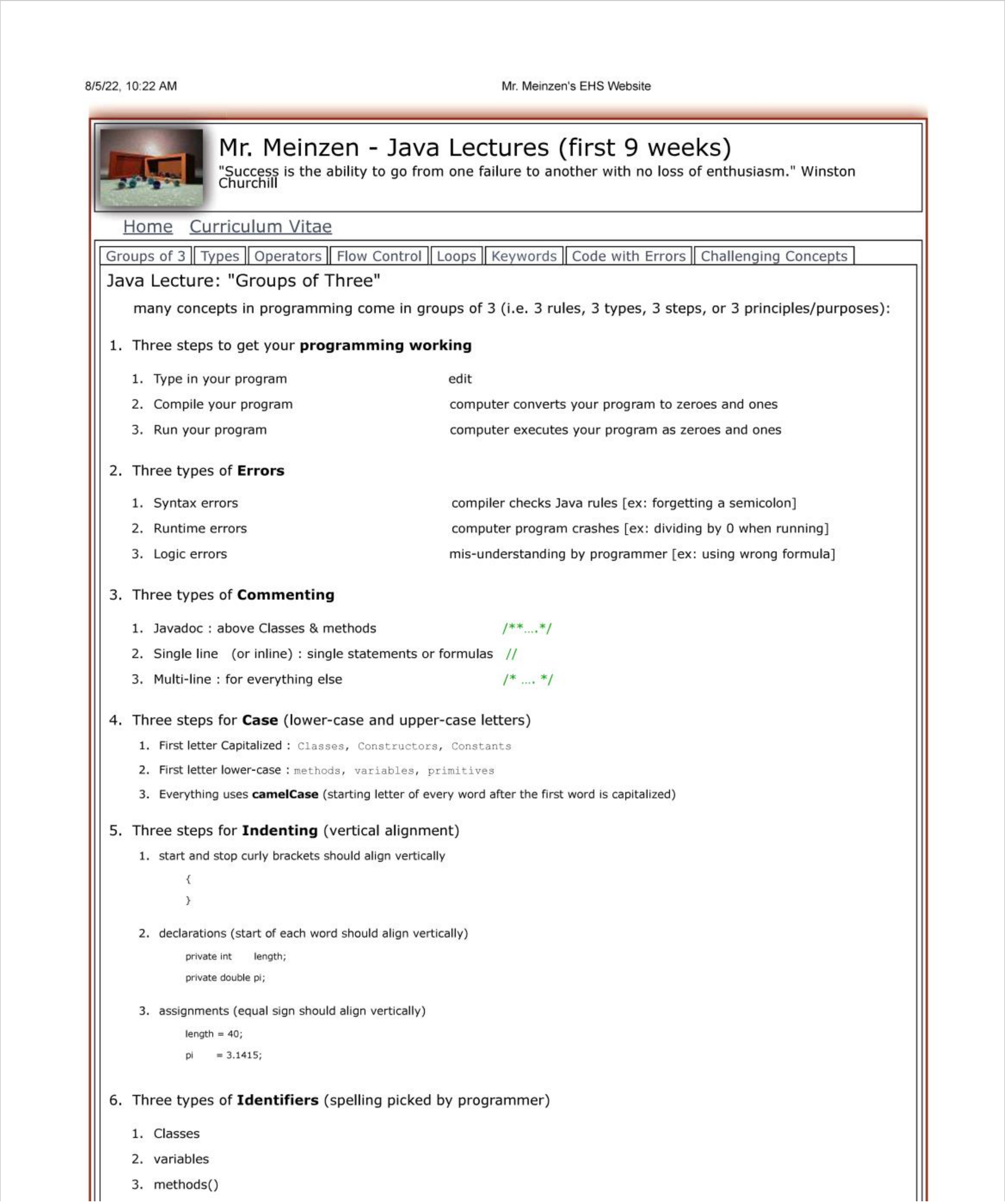
The following words have special meaning in the Java language and cannot be used as identifiers (i.e. names of variables, methods, or classes) by the programmer:

Note: "keywords" and "reserved words" are synonyms for practical purposes...the difference is that keywords have a purpose in the language whereas reserved words may or may not have a purpose. All keywords are reserved words but not vice-versa. Example: the "goto" reserved word cannot be used as a variable name and it serves no actual purpose in the Java language (other than prevents its use) and therefore is not a keyword.

Keywords introduced in (Honors) Programming in Java ****	Additional Keywords used in AP CSA	Other Java Keywords not explicitly taught in EHS courses.
boolean	abstract	assert
byte	catch	break
char	do	case
class	final	const
double	finally	continue
else	instanceof	default
extends	interface	enum
false	long	goto
float	package	native
for	short	protected
if	static	strictfp
implements	super	synchronized
int	switch	transient
import	throw	volatile
new	throws	
null	try	
private		
public		
return		
this		
true		
while		
void		

Review : Groups of 3 ****

Friday, August 5, 2022 10:39 AM



8/5/22, 10:22 AM Mr. Meinzen's EHS Website

Mr. Meinzen - Java Lectures (first 9 weeks)

"Success is the ability to go from one failure to another with no loss of enthusiasm." Winston Churchill

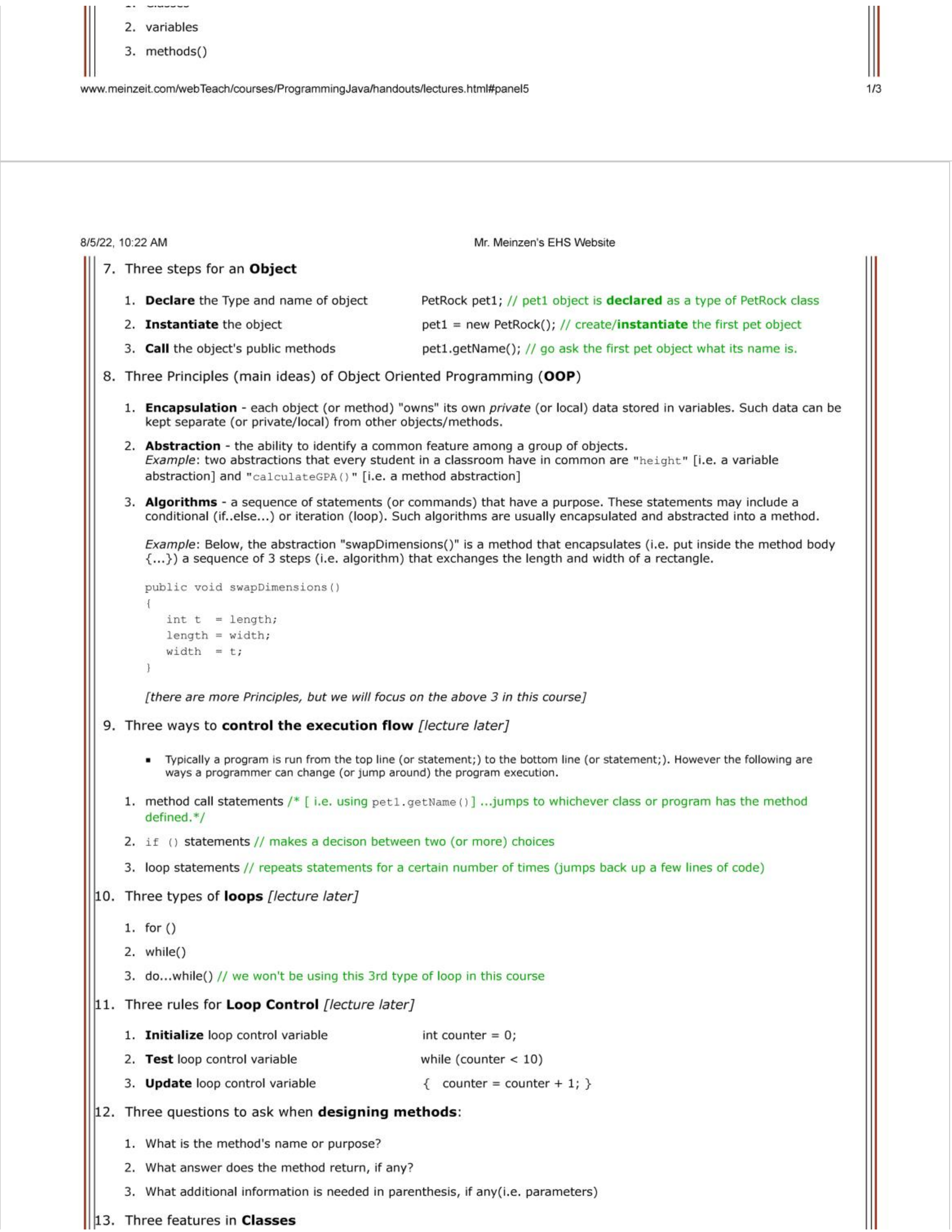
Home Curriculum Vitae

Groups of 3 | Types | Operators | Flow Control | Loops | Keywords | Code with Errors | Challenging Concepts

Java Lecture: "Groups of Three"

many concepts in programming come in groups of 3 (i.e. 3 rules, 3 types, 3 steps, or 3 principles/purposes):

1. Three steps to get your **programming working**
   1. Type in your program edit
   2. Compile your program computer converts your program to zeroes and ones
   3. Run your program computer executes your program as zeroes and ones
2. Three types of **Errors**
   1. Syntax errors compiler checks Java rules [ex: forgetting a semicolon]
   2. Runtime errors computer program crashes [ex: dividing by 0 when running]
   3. Logic errors mis-understanding by programmer [ex: using wrong formula]
3. Three types of **Commenting**
   1. Javadoc : above Classes & methods /**....*/
   2. Single line (or inline) : single statements or formulas //
   3. Multi-line : for everything else /* .... */
4. Three steps for **Case** (lower-case and upper-case letters)
   1. First letter Capitalized : `Classes, Constructors, Constants`
   2. First letter lower-case : `methods, variables, primitives`
   3. Everything uses **camelCase** (starting letter of every word after the first word is capitalized)
5. Three steps for **Indenting** (vertical alignment)
   1. start and stop curly brackets should align vertically
      ```
      {
      }
      ```
   2. declarations (start of each word should align vertically)
      ```
      private int    length;
      private double pi;
      ```
   3. assignments (equal sign should align vertically)
      ```
      length = 40;
      pi     = 3.1415;
      ```
6. Three types of **Identifiers** (spelling picked by programmer)
   1. Classes
   2. variables
   3. methods()



1. Classes
2. variables
3. methods()

7. Three steps for an **Object**

1. **Declare** the Type and name of object	PetRock pet1; // pet1 object is **declared** as a type of PetRock class
2. **Instantiate** the object	pet1 = new PetRock(); // create/**instantiate** the first pet object
3. **Call** the object's public methods	pet1.getName(); // go ask the first pet object what its name is.

8. Three Principles (main ideas) of Object Oriented Programming (**OOP**)

   1. **Encapsulation** - each object (or method) "owns" its own *private* (or local) data stored in variables. Such data can be kept separate (or private/local) from other objects/methods.
   2. **Abstraction** - the ability to identify a common feature among a group of objects. *Example*: two abstractions that every student in a classroom have in common are "`height`" [i.e. a variable abstraction] and "`calculateGPA()`" [i.e. a method abstraction]
   3. **Algorithms** - a sequence of statements (or commands) that have a purpose. These statements may include a conditional (if..else...) or iteration (loop). Such algorithms are usually encapsulated and abstracted into a method.

      Example: Below, the abstraction "swapDimensions()" is a method that encapsulates (i.e. put inside the method body {...}) a sequence of 3 steps (i.e. algorithm) that exchanges the length and width of a rectangle.

```
public void swapDimensions()
{
   int t  = length;
   length = width;
   width  = t;
}
```

   [there are more Principles, but we will focus on the above 3 in this course]

9. Three ways to **control the execution flow** *[lecture later]*

   - Typically a program is run from the top line (or statement;) to the bottom line (or statement;). However the following are ways a programmer can change (or jump around) the program execution.

   1. method call statements /* [i.e. using `pet1.getName()`] ...jumps to whichever class or program has the method defined.*/
   2. `if ()` statements // makes a decison between two (or more) choices
   3. loop statements // repeats statements for a certain number of times (jumps back up a few lines of code)

10. Three types of **loops** *[lecture later]*

    1. for ()
    2. while()
    3. do...while() // we won't be using this 3rd type of loop in this course

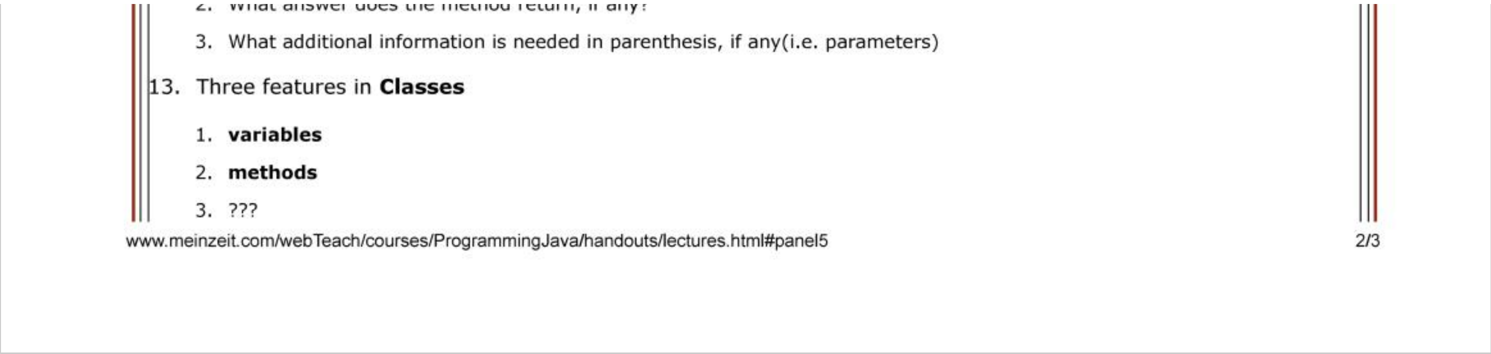
11. Three rules for **Loop Control** *[lecture later]*

1. **Initialize** loop control variable	int counter = 0;
2. **Test** loop control variable	while (counter < 10)
3. **Update** loop control variable	{ counter = counter + 1; }

12. Three questions to ask when **designing methods**:

    1. What is the method's name or purpose?
    2. What answer does the method return, if any?
    3. What additional information is needed in parenthesis, if any(i.e. parameters)

13. Three features in **Classes**



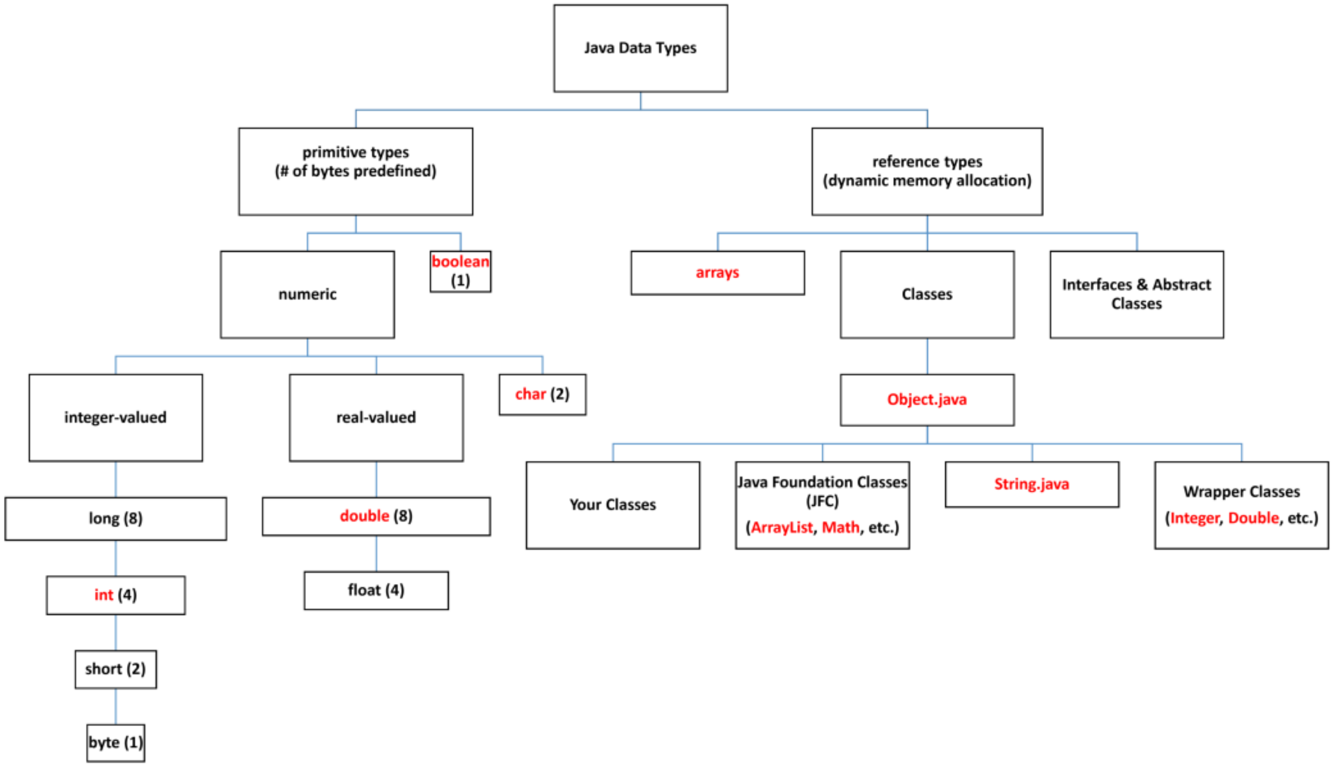
2. What answer does the method return, if any?
3. What additional information is needed in parenthesis, if any(i.e. parameters)

13. Three features in **Classes**

    1. **variables**
    2. **methods**
    3. ???

Syntax: Data Types****

Tuesday, May 31, 2022 1:04 PM



DataTypesHiearchy

Syntax: Variables

Tuesday, May 31, 2022 2:24 PM

A. Variables

1. 2 Types

a) **field variables that belong to the class**

b) **local variables that belong to a scope within a method (i.e. are declared inside a method body or parameters)**

2. Syntax

```
[final] Type variableName [= initializer];
```

where

a) `[...]` **is optional**

b) `Type` **is one of the Data Types chart**

c) `variableName` **are identifieres, are chosen by the Programmer, and must:**

- (1) consist only of digits, letters, and underscores
- (2) must begin with a letter or underscore
- (3) case sensitive
- (4) has a scope -- it exists from point of declaration to nearest end of block i.e { ... }
  - (a) a `final` variable must be initialized only once at point of declaration (which makes it a definition)
- (5) will be graded according to style:
  - (a) all letters lowercase
  - (b) except:
    - (i) `final` variables are all capitalized (also known as "constants")
    - (ii) the first letter of words following first word in variableName must be capitalized
  - (c) meaning or purpose in program

3. examples:

```
int    n;                 // declaration only
double x = 0;             // declaration & initialization (definition)

String harry = "Harry Handsome";
// object declaration & instantiation with assignment/initialization to String
literal

Rectangle box = new Rectangle(5,10,20,30);
// object declaration & instantiation (definition) using a constructor with
// 4 integer arguments passed by value to 4 integer parameters

int [] perfectSquares = {1,4,9,16,25};
// array reference/object declared & instantiated (automatically/implicitly to //
size 5) and initialized with 5 integer values by "block initializer".

final int MAXRANGE = 100;        // constant
final float _EPSILON = 0.0001;
// constant (the starting underscore implies it is a "system" variable)
```

4. variables : locals vs. fields -- scope and qualifiers

a) Definition & Usage

(1) a local variable is for temporary use in a method and therefore cannot be accessed outside its scope

(2) fields are variables declared in the class (outside of all methods)

(3) the scope of a variable is the region (usually denoted by {...}) of a program in which the variable can be referred to by its variableName.

(4) to access a field outside of its scope (i.e. outside it's class), the field variable must be qualified (using dot notation). Assuming the field is qualified as either `public` or `protected` (not `private`)

(5) within its scope (i.e. within its class) a field has an implicit qualifier, namely `this` (ex. `this.value`)

b) Notes

(1) It is a syntax error to have 2 identical local variables within the same scope or overlapping scopes

(2) variableNames should NOT be distinguished by case alone!

(3) duplicate fields & local variableNames:

(a) can overlap scopes

(b) cannot be both declared inside same scope -- why not?...ans: fields are class scope, locals are method scope

(c) Duplicate names are called "shadowing"... which can be useful but very dangerous if not done carefully!! If you intend to "shadow" your variable names, be prepared to explain why.

(4) it is a syntax error to have two duplicate field variableNames within one class scope.

5. Example ****

```
public class Test
{
    private int x;         // field variable,
                           // scope is the Test class,
                           // has implicit Test class qualifier this.x

    public Test (int x)    // local variable (aka parameter -- has method scope)
    {
         int    x;         // illegal as same scope as parameter

         String y;         // local variable object, method scope

         int      y;       // illegal, already declared local variable y

         for (int i=1; i<x; i++)    // ok, using local variable/parameter x, not field x
         {
              y = "hello";          // ok, just using local variable y
         }
     }


     public Test()        // default constructor
     {
         int x = 5;       // local variable x defined, shadows field x (hides the field)
                         // different scope the above constructor
         x = x + 1;       // OK since just modifying local variable
         x = x++;         // OK but what will x be after this?
                         // i.e. don't mix unary & binary expressions
         this.x = x;     // OK, private field this.x is being set equal to local variable x
     }
} // class Test
```

Syntax: Expressions

Tuesday, May 31, 2022 2:38 PM

A. Expressions

1. Dfn: An expression is one of the following:

a) **a literal (string or character or numeric) -- i.e. "this is a string literal", 'z', 23.0**

b) **a variable (local or field--perhaps qualified)**

c) **a method call (i.e. that returns a value or reference to an object)**

d) **combination of subExpressions acted on by operators**

2. Examples:

a) `'5'        // a character literal`

b) `x            // variable (type unknown)`

c) `Math.sin(x)     // method call (specifically a static class method)`

d) `x + Math.sin(x)        // two subexpressions acted upon by the + operator (Note: both subexpressions must be of compatible types)`

e) `x++                 // a variable acted upon by the ++ operator`

f) `x == y               // 2 variables acted upon by the == operator (returns a boolean value)`

g) `x == y && (z>0 || w<0)    // 4 variables, 5 operators`

h) `p.x                // a variable, specifically a "qualified field variable", p is the "name of the object" or "reference to the object" or "name of the instance" and x is the class field`

i) `v[i]                // variable, specifically an array element where v is the array, i is the index`

j) `e.getSalary()            // method call of an instance/object named e.`

3. Operators

a) **Three types of operators**

(1) unary -- acts upon only 1 variable

(a) prefix -- operation is done before all other operators

(i) `++x, --x, -x, +x` (Note: +x is same as x, -x is the negation of x if it exists)

(b) postfix - operation is done after all other operators are finished

`(i) x++, x--`

(c) other

(i) . access class feature ("qualifier operator")

(ii) [] array subscript

(iii) () method call

(iv) ! boolean NOT

(v) ~ bitwise not

(vi) `(TypeName)` Cast to another (super) class

(vii) `new` Object allocation/instantiation

(2) binary -- works on two expressions or two variables

(a) * multiplication

(b) / division or integer division

(c) % integer remainder (modulus)

(d) + addition

(e) - subtraction

(f) << shift left

(g) >> arithmetic shift right

(h) >>> bitwise shift right

(i) < less than

(j) <= less than or equal to

(k) > greater than

(l) >= greater than or equal to

(m) == equal to

(n) != not equal to

(o) **instanceof** tests whether an object's type is a given type or subtype thereof

(p) & bitwise and

(q) ^ bitwise exclusive or

(r) | bitwise or

(s) && bitwise "short circuit" and (also called a "logical and")

(t) || bitwise "short circuit" or (also called a "logical or")

(u) = assignment

(v) op= assignment with binary operator (op is one of +,-,*,/,%,&,|,^,<<,>>,>>>)

(3) ternary -- operates on 3expressions, Java has only 1 and is only rarely used

(a) testExpression ? expression1 : expression2

(i) Example:

```
(testExpression == true) ? return expression1 : return expression2;
```

(b) same as:

```
if (testExpression == true)
     return expression1;
else
     return expression2;
```

b) Precedence -- normal math precedence with ( ) as precedence override

c) Notes

(1) most operators are left-associated

(a) ex. x - y + z is same as (x - y) + z which is the same as mathematics

(2) exceptions (which are right-associated) are

(a) unary prefix and others -- examples include +x, -y, ++x, --x, !x, ~x, (int)x, new X()

(b) assignment and op= (ex. x = y = Math.sin(j) is same as x = (y = Math.sin(j))

Syntax: Class Declarations

Tuesday, May 31, 2022 2:43 PM

1. **Syntax of Class declarations**

```
[public | private] [abstract | final ] class ClassName [extends SuperClassName] [implements InterfaceName1,
Interface2, ...]
{
     feature1;
     feature2;
     ...
}

where
     feature is either:
          1) a declaration of the form:
                    modifiers field | constructor | method | inner class
     or
          2) an initialization block of the form:
                    [static] { body }
```

2. **Notes on Class declarations**

   a) **the underlined items were introduced in Honors Java Programming course but not always defined or explained**

   b) **the [...] is optional syntax**

   c) **the | means "exclusive or" (i.e. either choose one or the other but not both options)**

   d) `abstract, final, extends, implements, inner class` **and** `static` **will be discussed later.**

   e) **modifiers are: (may choose one of** `public, private,` **or** `protected` **along with either one or both of** `static` **and** `final`**)**

      (1) `public` -- make this feature available to all other classes

      (2) `private` -- make this feature only available to this class

      (3) `protected` -- make this feature only available to this class, subclasses (descendents) and classes in same package (directory).
      Note: protected, for historical reasons, is not used in this course.

      (4) `static` -- defined for this class, not for each instance of the class. (i.e. every object of this class has the exact same --1 and only 1--feature)

      (5) `final` -- (same as a constant) has the following three properties:

         (a) a field that cannot be changed once it is initialized at the point of declaration

         (b) a method that can't be overridden by another subclass   (i.e. defeats polymorphism)

         (c) a class that can't be extended (i.e. no subclasses)

3. **field declarations**

   a) **Syntax:** `[modifiers] Type variableName [= initializer];`

   b) **2 types of fields**

      (1) instance fields -- each object has its own copy of the field

      (2) static fields -- only 1 per-class copy. Also known as "class fields"

         (a) Note1: static fields are useful for "sharing" common information between all instances of a class

         (b) Note2: This breaks "encapsulation." So, to avoid accidental changes, use as `static final.` Therefore it can be set only once by original programmer

         (c) Note3: static variableNames use, by convention, all capital letters

         (d) Note 4: there are three ways to initialize static fields:

            (i) do nothing ... default values will be supplied by JVM... poor programming and will be major loss of points!!!

            What are default values for `int, long, float, double, char, boolean, byte, short`???

            (ii) explicitly at point of declaration/definition :
            i.e. `public static final double PI = 3.141592;`

         (e) static initialization block: (not used in this course)

```
public class Cake
{
    private static int COUNTER;    // field COUNTER is shared among all objects of type Cake
                                    // May be useful when trying to keep track of how many
                                    // Cake objects have been constructed

    public Test()        // constructor
    {
         ...
    }
    ...
    static {COUNTER=0;}  // static initialization block for field COUNTER can be anywhere.
    ...
}
```

4. constructor declaration : *Note: technically, constructors are NOT methods as there is no return type but they behave similar to methods*

a) Syntax

(1) **syntax for <u>declaring</u> a constructor**

```
[modifiers] ClassName ( [Type parameter1, Type parameter2, ...] )   [throws ExceptionType1, ExceptionType2, ...]
{
    body
}
```

(2) **syntax for <u>calling</u> a constructor** (or using the constructor from another class). AKA "invoking" or "instantiating"

```
new ClassName ( [argument1, argument2, ... ] );
```

b) Comments

(1) `throws` is used for "error handling" and will be discussed later

(2) when invoking a constructor, the <u>arguments</u> and <u>parameters</u> must match in:

(a) compatible Type (i.e. the same Type or can be "cast" to the same Type)

(b) same number of parameters as arguments

(c) same order of parameters and arguments

(3) when an object (instance) is constructed, the following actions take place:

(a) all fields are given default values

(i) `0` for numbers

(ii) `false` for booleans

(iii) `null` for reference/object fields)

(b) initializers & initializer blocks are executed in the order in which they appear

(c) body of constructor is executed/run

(4) when a class is loaded into the JVM the following actions take place:

(a) all static fields are initialized with default values

(b) initializers and initializer blocks are executed in the order in which they appear

(5) one constructor can call another constructor <u>in the same class</u> by the invocation:

(a) `this ( [ argument1, argument2, ... ] );`

(b) However, the invocation must be on the <u>first</u> line of the body of the calling constructor.

(c) Example:

```
private Coin(int v)     // no one can use this constructor directly
{
    value = v;
}

// second constructor which can be used by other classes
public Coin (int v, String n)
{
    this(v);           // calls the above constructor,  value now equals v (note: on first line)
    name = n;
}
```

(6) one constructor may call the constructor of the <u>"parent" or "super"</u> class (that the current class `extends`) by the invocation:

(a) `super ( [argument1, argument2, ... ] );`

(b) However, the invocation must be on the <u>first</u> line of the body of the calling constructor

Syntax: Special : arrays & Strings

Tuesday, May 31, 2022 2:53 PM

Special references (i.e. Specialized Java Classes)

1. ****** arrays**

    a. Syntax:
    ```
    new ArrayType[] [ = {initializer1, initializer2, ... } ];
    ```

    b. Example:
    ```
    // declare and instantiate (i.e. define) 52 empty slots that
    // can--but not yet--hold Card objects
    Card myDeck[] = new Card[52];
    ```

    c. Example:
    ```
    new int [ ] = {1,4,5, 16, 25};
    ```

    i. Note1: this Example is an "anonymous" array...no variable name is attached

    ii. Note2: the Examples "int [ ]" is part of the ArrayType whereas the "[= { ...}]" under Syntax is optional

    iii. Note3: anonymous array definitions sometimes occur when arguments are created and passed to parameter names

2. ****** String** class : effectively "has a" private array of char with methods declared to make Strings easeir to manipulate.

    a. **Construction**

    ```
    String name1 = "Rob";               // one String object referring to a literal
                                        // String "Rob", this is an implied constructor invocation
                                        // unique to String literals in Java

    String name2 = "R" + "o" + "b";     // one anonymous String object constructed from 3 literals
                                        // NOTE: JVM will recognize name2 as an object already
                                        //       constructed using same literal sequence as name1
                                        //       and therefore will NOT instantiate a new object.

    String name3 = new String("Rob");   // A new String object constructed from a
                                        // literal String "Rob" object that already exists.

    String name4 = name3;               // Not a construction of a new object instance!
                                        // name4 is a new reference to the exact same object name3
    ```

    b. Notes

    i. All Strings are <u>immutable</u> -- once instantiated & initialized (i.e. defined), a String object can never be changed. However it can be copied. In other words, after instantiation, there are only "accessor" methods but no "mutator" methods.

    ii. **Do NOT compare Strings using ==.** This leads to numerous problems as the "==" operator is meant for primitive Types.

    1) Example:

    a) `name1 == name2` *// returns true. Both constructied from same String* ***literal sequence :*** **`"Rob"=="R"+"o"+"b"`**

    b) `name3 == name4` *// returns true. Both are references (i.e. pointers) to the same String object*

    c) `name1 == name3` *// returns false!!! name1 and name3 are constructed differently...they are two different objects*

    // with the same characters in different arrays (i.e. same contents inside different objects)

    2) Why? Draw the **UML object Diagram!**

    3) to properly compare String objects use the `equals()` method for each String object i.e.
    ```
    if (name1.equals(name2))     // returns true since
    ```

```
                                        // equals() method compares
                                        // contents, not references
```

4) This warning holds true for comparison of any object. Ergo == operator compares if the references are the same, not contents.

5) In order to compare contents of the objects (i.e. field values the same), **must** use the `equals()` method.

6) However, comparison of contents is a complex topic and will be addressed only if we have enough time

7) Finally, the empty String `""` is not the same as the `null` String

```
String name1;
String name2="";
if (name1 == null)       // returns true
if (name2 == "")         // return true but dangerous! why?
if (name2 == null)      // returns false;
```

8) the "" is a valid String object (i.e. a size 0 String)

9) the `null` String means that a String variable has been declared but not instantiated or referenced properly (i.e. nullPointerException error)

c. ******** The most common String methods used are on the Java Quick Reference for the AP CSA Exam********. For example:

i. **String(String str)** // constructs a new String object with a duplicate array of characters

ii. **int length()** // returns the number of characters in a String object

iii. **String substring(int from, int to)** // returns a portion (i.e. substring) of the String beginning at index **from** and ending at index **to-1**

iv. **String substring(int from)** // returns a portion (i.e. substring) of the String beginning at index **from** and copying the remaining characters Same as substring(from, length())

v. **int indexOf(String str)** // returns the index of the first occurrence of **str**; returns **-1** if **str** is not found

vi. **boolean equals(String other)** // returns true if **this** is equal **other** (character by character comparision); returns false otherwise.

vii. **int compareTo(String other)** // Returns a value < 0 if **this** is less than **other**; returns zero if **this** is equal to **other**; returns > 0 if **this** is greater than **other**. Uses a first character to first character comparison. If first characters are equal, then checks 2nd characters for equality…and so on.

viii. Examples: Suppose we define: **String name = "Fred Generation";** // recall 'F' is at index=0

1) **String name2 = new String(name);** // create a new object with duplicate name contents.

2) **name.substring(1,7)** // returns "red Ge"

3) **name.substring(7)** // returns "neration"

4) **name.length()** // returns 15 which is the number of characters in the name

5) **name.indexOf("Fred")** // returns 0

6) **name.indexOf("n")** // returns 7

7) **name.equals(name2)** // returns true.

8) **name == name2** // returns false. Two different objects with same contents

9) **name.compareTo(name2)** // returns 0. Same as using name.equals(name2).

10) **name.compareTo("Frit")** // return integer < 0 since "Fre..." is less than "Fri..."

Example : String & array manipulations ****

Monday, August 8, 2022 8:27 AM

Scenario:

Suppose you are are writing a Java application for a new music player. You want users to be able to select numerous song names (i.e. file names) then drag-n-drop them into your application to be played in order.

Before you jump into the various "exciting" parts of the program, you need to make sure you can parse (take apart) the song names and verify that those songs are actually stored using the *.mp3* file extension at the end of the file name which can store music as opposed to *.mp4* file extensions which are used to store videos.

Another requirement for the music files is that the file name will always consist of the following :

1. title of the song followed by a hyphen,
2. artist(s)' name (firstName then LastName, if any) followed by a hyphen, and
3. a 4-digit year that the song was released.

Your assignment is to write the following code parts (in order of difficulty):

1. Write an application (i.e. main method) that parses the String array parameter. Note: see Starter Code below.
2. Write a private method to print out a each file name in any given array along with a prompt as to what the array contains. The method should use a for each loop and have a method header : `private static void printFileNames(String prompt, String fn[])` Note : see Starter Code below.
3. Write a method to scan through the parameter and generate a new array of Strings consisting of file names that ONLY have ".mp3" extensions. The method header will be : `public static String[] keepOnlyMP3(String[] args)` Note : see Starter Code below.
4. Write a method to parse each file name and place the name of the song in a new array of Strings.
5. Write a method to parse each file name and place the artist(s)' name in a new array of Strings.
6. Write a method to parse each file name and place the year released in a new array of Strings.
7. Challenges:
   i. sort the array of file names by year the song was released
   ii. sort the array of file names by the artist(s)' first name
   iii. identify file names that only have artist(s)' first name (i.e. no Last Name)

Some examples of file names:

- *dontStopBelieving-journey-1981.mp3*
- *hello-adele-2015.mp3*
- *hello-adele-2015.mp4* (Note: this is the music video that <u>cannot</u> be played by your new music player)
- *theTwist-chubbyChecker-1960.mp3*
- *sillyLoveSongs-paulMcCartney-1976.mp3*

The following Java concepts and reminders may be helpful:

- all Java applications have a main method with a String array as a parameter...the name of the parameter is typically `arg`
- The main method's signature is :
  `public static void main(String arg[])    // Note: arg is a variable/parameter name`
- The String class comes with many methods. The following are used on the AP CSA Exam and included in the Exam Reference Sheet : Red font is given and demonstrated in the starter code below

  i. **String(String str)** // constructs a new String object with a duplicate array of characters

  ii. **int length()** // returns the number of characters in a String object

  iii. **String substring(int from, int to)** // returns a portion (i.e. substring) of the String beginning at index **from** and ending at index **to-1**

  iv. **String substring(int from)** // returns a portion (i.e. substring) of the String beginning at index **from** and copying the remaining characters Same as substring(from, length())

  v. **int indexOf(String str)** // returns the index of the first occurrence of **str**; returns **-1** if **str** is not found

  vi. **boolean equals(String other)** // returns true if **this** is equal **other** (character by character comparision); returns false otherwise.

vii. **int compareTo(String other)** // Returns a value < 0 if **this** is less than **other**; returns zero if **this** is equal to **other**; returns > 0 if **this** is greater than **other**. Uses a first character to first character comparison. If first characters are equal, then checks 2nd characters for equality…and so on.

The following starter code may be helpful:

```
public class MusicFileNames {

    private static String testStrings[] = {
        "dontStopBelieving-journey-1981.mp3",
        "hello-adele-2015.mp3",
        "hello-adele-2015.mp4",
        "theTwist-chubbyChecker-1960.mp3",
        "sillyLoveSongs-paulMcCartney-1976.mp3",
    };

    /**
     * main method is needed for a Java application
     * @param args : the array of music file names that are to be drag-n-dropped by user
     */
    public static void main(String[] args) {

        // temporarily replace the user's file names with test file names
        args = testStrings;
        printFileNames("args for testing", args);

        // 1st generate a new array of Strings that ONLY have .mp3 file extensions
        String onlyMP3[] = keepOnlyMP3(args);
        printFileNames("onlyMP3", onlyMP3);

        // 2nd parse the file names into 3 separate categories according to 1. song name,
        // 2. song writer, and 3. year released.
    }

    /**
     * private method simply prints out a list of file names stored in an array
     * @param prompt : a message (usually the name of the array) to explain purpose
     * @param fn : the array of file names to be printed, one to a line.
     */
    private static void printFileNames(String prompt, String fn[])
    {
        System.out.println("File names in the array named: "+prompt);
        for (String n : fn) {
            System.out.println("  "+n);
        }
    }

    /**
     * This method takes an array of generic file names, counts how many file names
     * have an ".mp3" file extension, and then creates and returns a new array
     * of file names that include ONLY file names with ".mp3" extensions.
     * Note: this would be MUCH easier using ArrayLists rather than arrays
     * @param args an array with a variety of file names
     * @return an array consisting only of file names with ".mp3" extensions
     */
    public static String[] keepOnlyMP3(String[] args) {

        // determine number of files with ".mp3" extensions
        int countMP3 = 0; // assume 0 files with ".mp3" extensions in the array
        for (String n : args) {

            // grab last 4 characters of file name (i.e. the extension)
            String extension = n.substring(n.length()-4);

            // if the extension is ".mp3" then we have to add one to the mp3 counter
            if (extension.equals(".mp3"))
                countMP3++;
        }

        // create a new array and store only the file names with ".mp3" extensions
        int index = 0;
        String onlyMP3[] = new String[countMP3];
        for (String n : args) {

            // repeat above logic...find only the last 4 characters that have ".mp3"
            String extension = n.substring(n.length()-4);
            if (extension.equals(".mp3")) {
                onlyMP3[index] = n;
                index++;
            }
        }
        return onlyMP3;
    }
}
```

Syntax: Method Declarations

Wednesday, June 1, 2022 9:37 AM

1. method declaration

1. Syntax

```
modifiers ReturnType methodName (       [ [final] ParameterType parameter1,
                                          [final] ParameterType parameter2, ...
                                        ]
                                )
                                       [   throws ExceptionType1,
                                            ExceptionType2, ...
                                        ]
{
     body of methodName
}
```

where

a. **`ReturnType`** is any Java Type or **`void`** (i.e. no return value). See lecture on Syntax: Data Types

b. modifiers are: (may choose one of **`public, private,`** or **`protected`** along with either one or more of **`static, final,`** and **`abstract`**)

i. **`public`** -- make this method available to all other classes

ii. **`private`**-- make this method only available to this class

iii. **`protected`** -- make this method only available to this class, subclasses (descendents) and classes in same package (directory). Note: **`protected`** is not used in this course.

iv. **`static`**-- defined for this class, not for each instance of the class. (i.e. every object of this class has the exact same--1 and only 1--method).

1) also known as "class methods" as opposed to "instance methods" since the method belongs to the class as a whole

2) Example: Math.sqrt(x) // x must be either a static field or literal value

3) Note1: static methods can only affect static fields (not instance fields) and static parameters (cannot use the implicit instance qualifier "this")

4) Note2: the historical choice of term "static" is poor...as it implies that it cannot be changed, which is not true...it can change fields but they also must be "static"

v. **`final`** -- has the following properties:

1) the method cannot be overridden, (i.e. not overloaded or extended) by any subclass method (i.e. another programmer cannot "hide" your method behind his/her own method of the same signature)

a) **Dfn: a *"method signature"* consists of the *methodName, parameterTypes,* and *parameter order* but not the *returnType, modifiers,* or *ExceptionTypes.***

2) defeats polymorphism and therefore, in general, should be avoided

3) Example:

a)
```
public final String checkPassword()
{
...
}
```

b) the programmer did NOT want another programmer to be able to write a different checkPassword() method that might be used to allow "poor" passwords...minimizes attack vectors by nefarious programmers

vi. **`abstract`**-- has the following properties

(1) forces another programmer to <u>extend</u> the method (and class)

(2) has no method body (i.e. not even {...})

(3) any class with any abstract methods cannot be instantiated (since an object won't know how to execute a non-existent method body). Therefore abstract methods (& classes) can only be extended by concrete classes (eventually)

(4) throws and ExceptionTypes will be discussed later

(5) Example: *(see lecture Example: abstract, final, exceptions)*

b) Notes

(1) Generic method <u>declaration</u> is not covered in this class...only Generic method <u>call/invocationU</u>...see later lecture.

(2) if ReturnType (other than void) is specified, then the `return` statement must:

(a) be "guaranteed" to be executable (otherwise a compiler warning)

(b) include an expression whose Type is "compatible" with ReturnType

(c) the method exits as soon as the return statement is encountered.

(3) do NOT put final and abstract in same class since:

(a) `abstract` implies the class must be extended

(b) `final` implies the class cannot be extended

(4) `[final] ParameterType parameter`

(a) ** Numerous common mistakes!!!

(b) ** Explanation: All arguments from calling (invoking) method are passed to the corresponding parameter by value (i.e. the value of the argument is copied into a new memory location for use as a parameter)

(i) for primitives, cannot change argument via the parameter (i.e. can't change "backward")

(ii) for references (objects), cannot change original argument reference but can change original argument reference (object) fields via mutator methods (i.e. setXXX()) and public fields & methods (i.e. normal class interfaces)

(5) Example: *(see lecture Example: UML object Diagrams)*:

Example : Debug : abstract, final, exceptions

Wednesday, June 1, 2022 10:43 AM

Scenario:

Suppose several programmers are writing code for a bank. The lead programmer, Mr M, is a security expert and at least one other programmer, jmein, is a newbie trying to learn to be a security expert.

The lead programmer needs a base-level (i.e. parent or super) class called *BankAccount* to securely handle a single password for all other accounts such as *SavingsAccount*, *CheckingAccount*, etc.

The lead programmer will write the *BankAccount* class that will make sure passwords are "secure,"

The newbie programmers will write a sub-class called *SavingsAccount* that is a *BankAccount* with one required method, *createPassword()*, to see if each newbie programmer can create "secure enough" passwords for the bank.

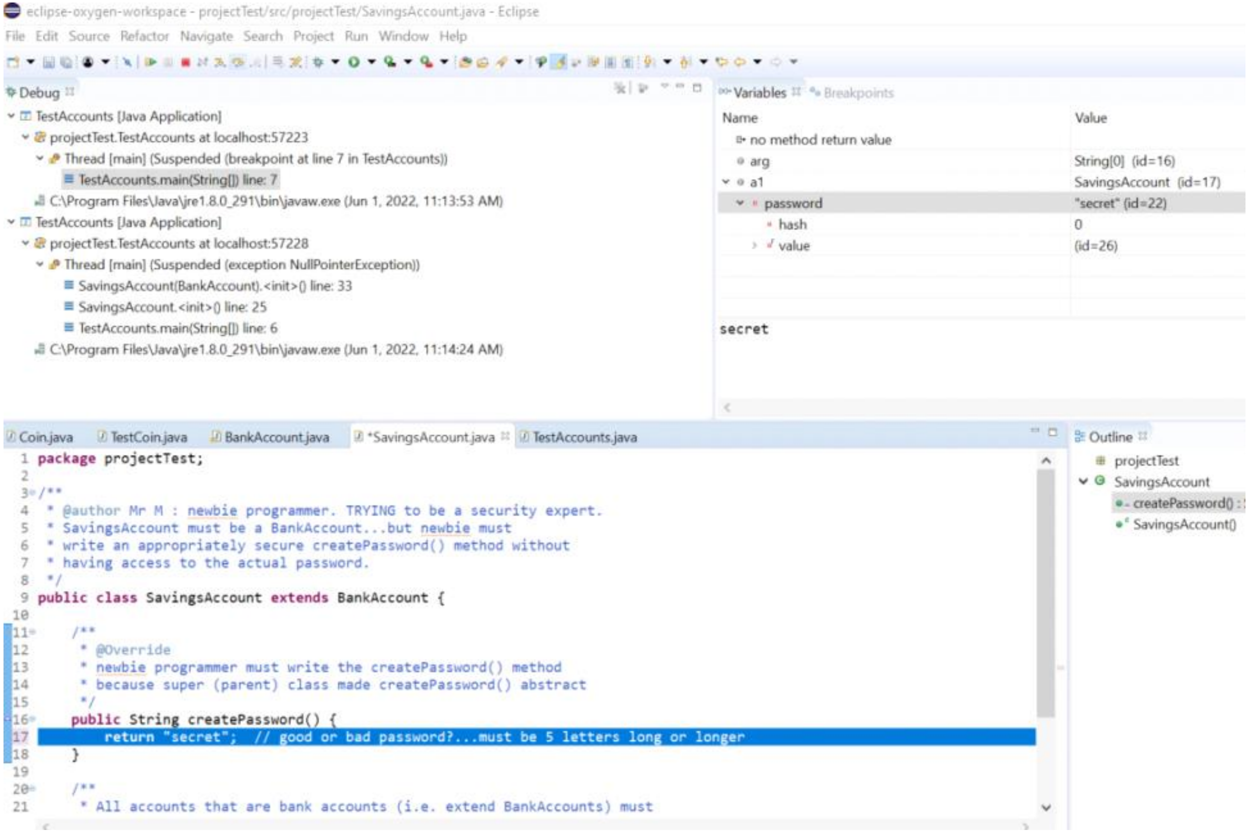
However, if the newbie programmer does not write the *createPassword()* method correctly, then the lead programmer will make sure no *BankAccount* (or any other account) can be constructed and a *NullPointerException* will be generated.

Here is one solution to the above scenario:

```
/**
 * @author Mr M: The security expert!
 * BankAccount is the base-level class for all accounts at a bank and stores the password.
 * All other accounts (SavingsAccount, CheckingAccount, etc.) must extend BankAccount.
 * However, no single instance of BankAccount can be created.
 */
public abstract class BankAccount {

    private String password = null;  // responsible for all passwords in the bank

    /**
     * Requires newbie security programmer to write the
     * createPassword() method in the sub-class (i.e in SavingsAccount)
     * @return the newbie's attempt at a password
     */
    public abstract String createPassword();  <--- NOTE THE SEMI-COLON.  NO METHOD BODY!

    /**
     * Method checkPassword() will verify the parameter, pwd, is "secure" enough.
     * The method is private so no other class can call/use checkPassword().
     * The method is final to prevent another sub-class from overriding checkPassword().
     * private final is a way to prevent child-classes (i.e. newbie programmers
     * from "seeing" or "hiding" parent method.
     * @param pwd : the password attempt that must be at least 5 letters long
     * @return true if password is long enough (5 letters); false otherwise.
     */
    private final boolean checkPassword(String pwd)
    {
        if (pwd.length() < 5) {
            return false;  // not long enough to be a "secure" password
        }
        password = pwd;  // set the password
        return true;
    }

    /**
     * Constructor allows for a new bank account to be created only if a
     * "secure" password can be created by a child class (SavingsAccount, CheckingAccount,etc.)
     * @throws NullPointerException if the newbie security programmer cannot create "secure" passwords
     */
    public BankAccount() throws NullPointerException {
        String tempPassword = createPassword();      // call the newbies attempt at a secure password
        if (this.checkPassword(tempPassword) == false) // check password
            throw new NullPointerException();
    }
}
```

```
/**
 * @author jmein : newbie programmer. TRYING to be a security expert.
 * SavingsAccount must be a BankAccount...but newbie must first
 * write an appropriately secure createPassword() method without
 * having access to the actual password.
 */
public class SavingsAccount extends BankAccount {

     /**
      * @Override
      * newbie programmer must write the createPassword() method
      * because super (parent) class made createPassword() abstract
      */
     public String createPassword() {
          return "secret";  // good or bad password? Must be five letters or longer
          // return "scrt"  // this is 2nd attempt...should generate NullPointerException
     }

     /**
      * All accounts that are bank accounts (i.e. extend BankAccounts) must
      * call the BankAccounts constructor method to set a password.
      */
     public SavingsAccount() {
          super(); //Note: unnecessary as compiler will call super() by default
     }
}
```

SCREENSHOT SHOWING password="secret" appears to work



eclipse-oxygen-workspace - projectTest/src/projectTest/SavingsAccount.java - Eclipse

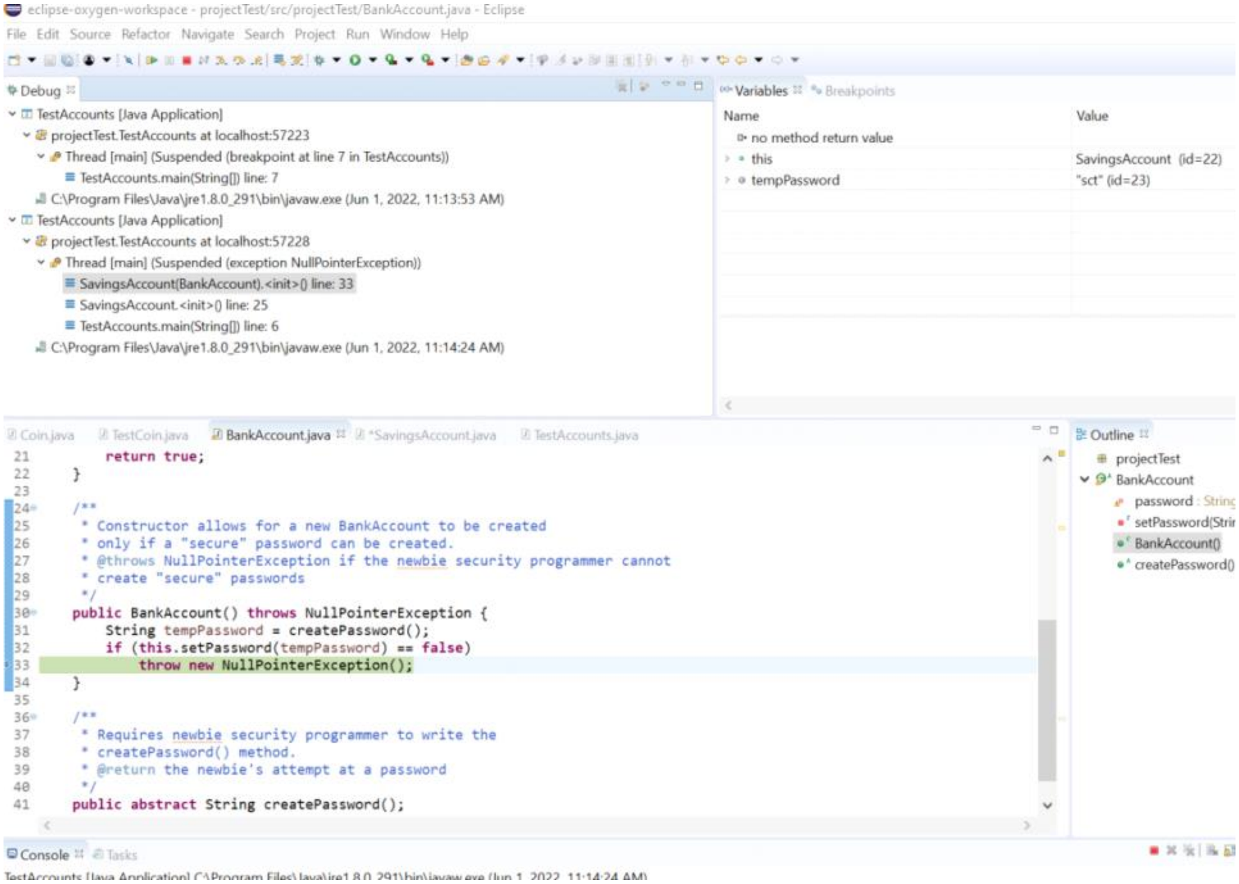
```
TestAccounts [Java Application]
  projectTest.TestAccounts at localhost:57223
    Thread [main] (Suspended (breakpoint at line 7 in TestAccounts))
      TestAccounts.main(String[]) line: 7
  C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (Jun 1, 2022, 11:13:53 AM)
TestAccounts [Java Application]
  projectTest.TestAccounts at localhost:57228
    Thread [main] (Suspended (exception NullPointerException))
      SavingsAccount(BankAccount).<init>() line: 33
      SavingsAccount.<init>() line: 25
      TestAccounts.main(String[]) line: 6
  C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (Jun 1, 2022, 11:14:24 AM)
```

Name	Value
no method return value	
arg	String[0] (id=16)
a1	SavingsAccount (id=17)
password	"secret" (id=22)
hash	0
value	(id=26)

```
secret
```

```
package projectTest;

/**
 * @author Mr M : newbie programmer. TRYING to be a security expert.
 * SavingsAccount must be a BankAccount...but newbie must
 * write an appropriately secure createPassword() method without
 * having access to the actual password.
 */
public class SavingsAccount extends BankAccount {

    /**
     * @Override
     * newbie programmer must write the createPassword() method
     * because super (parent) class made createPassword() abstract
     */
    public String createPassword() {
        return "secret";  // good or bad password?...must be 5 letters long or longer
    }

    /**
     * All accounts that are bank accounts (i.e. extend BankAccounts) must
```

SCREENSHOT SHOWING password="sct" generates NullPointerException


eclipse-oxygen-workspace - projectTest/src/projectTest/BankAccount.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Variables
Breakpoints
Coin.java
TestCoin.java
BankAccount.java
*SavingsAccount.java
TestAccounts.java
Outline
Console
Tasks

```
TestAccounts [Java Application]
  projectTest.TestAccounts at localhost:57223
    Thread [main] (Suspended (breakpoint at line 7 in TestAccounts))
      TestAccounts.main(String[]) line: 7
  C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (Jun 1, 2022, 11:13:53 AM)
TestAccounts [Java Application]
  projectTest.TestAccounts at localhost:57228
    Thread [main] (Suspended (exception NullPointerException))
      SavingsAccount(BankAccount).<init>() line: 33
      SavingsAccount.<init>() line: 25
      TestAccounts.main(String[]) line: 6
  C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (Jun 1, 2022, 11:14:24 AM)
```

Name	Value
no method return value	
this	SavingsAccount (id=22)
tempPassword	"sct" (id=23)

```
21            return true;
22        }
23
24        /**
25         * Constructor allows for a new BankAccount to be created
26         * only if a "secure" password can be created.
27         * @throws NullPointerException if the newbie security programmer cannot
28         * create "secure" passwords
29         */
30        public BankAccount() throws NullPointerException {
31            String tempPassword = createPassword();
32            if (this.setPassword(tempPassword) == false)
33                throw new NullPointerException();
34        }
35
36        /**
37         * Requires newbie security programmer to write the
38         * createPassword() method.
39         * @return the newbie's attempt at a password
40         */
41        public abstract String createPassword();
```

```
projectTest
BankAccount
  password : Strinc
  setPassword(Strir
  BankAccount()
  createPassword()
```

```
TestAccounts [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (Jun 1, 2022, 11:14:24 AM)
```

Screen clipping taken: 6/1/2022 11:22 AM

Examples: UML object Diagrams ****

Wednesday, June 1, 2022 11:05 AM

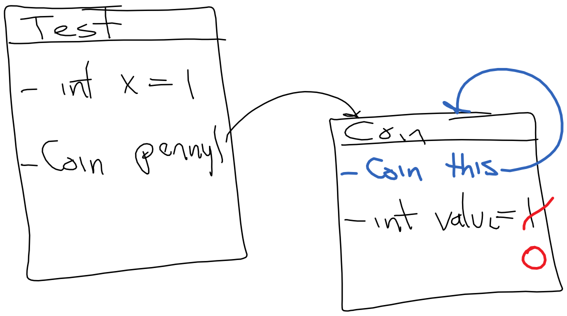
Example 1: Testing of a Coin object

calling program/method

```
public class TestCoin
{
     public static void main(String arg[])
     {
          // local variables (not fields) to main() method
          // private and public only apply to field variables, not local variables
          int  x      = 0;
          Coin penny1 = new Coin();

          // invoking penny1's object method setValue() with argument x.
          // The JVM will copy the value of argument x into parameter y declared in Coin class
          penny1.setValue(x);
     }
}
```

invoked or called program/method

```
public class Coin
{
     private int value = 0;          // field variable
                                     // other fields and methods not shown
                                     // default constructor provided by compiler
     public void setValue(int y)     // parameter y (local variable, method scope)
     {
          value = y;                // same as this.value = y;
     }
}
```



Example 2: (note: primitive variables not shown and lines are number for reference only)

```
public class TestPocketandCoins
{
     public static void main(String a[])
     {
          Pocket p;                   // holds up to 2 Coins
1)        p = new Pocket();           // create a new pocket
          Coin penny,dime;            // two coins
2)        penny = new Coin(1);        // the penny object has value 1
3)        dime  = new Coin(10);       // the dime object has value 10
         ...
4)        p.setCoin1(penny);          // put a penny into the pocket's first coin
5)        penny = null;
6)        p.setCoin2(dime);           // put a dime into the pocket's 2nd coin
         ...
     }
}
```

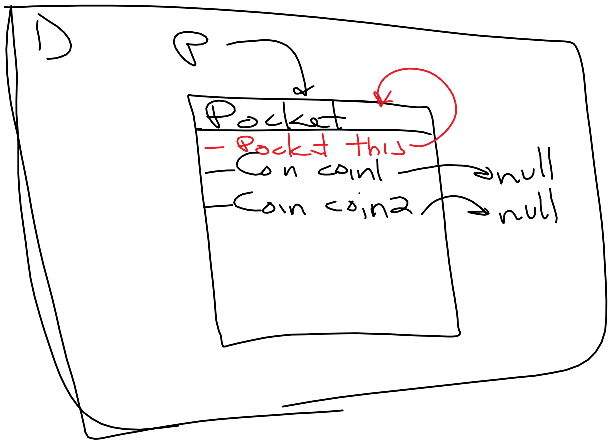
called program/method

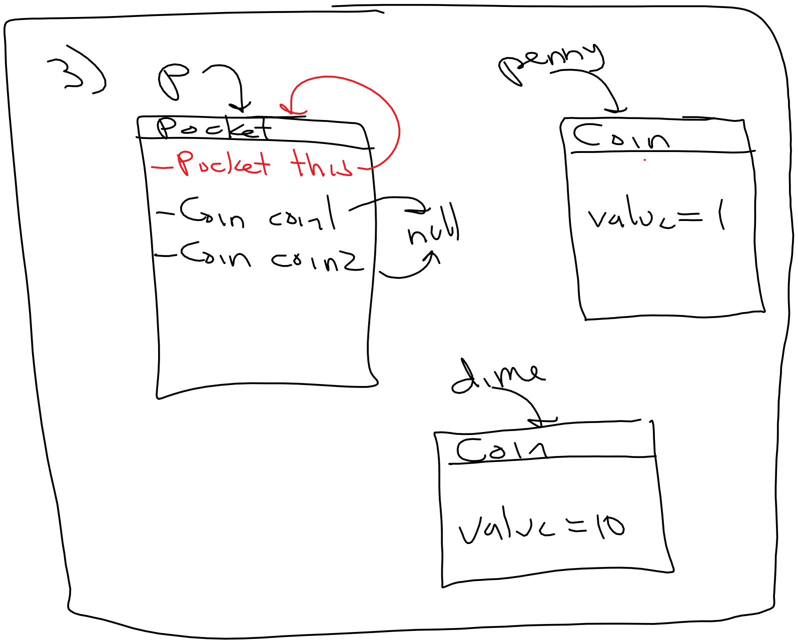
```
public class Pocket
{
     private Coin coin1 = null;     // first coin in pocket
     private Coin coin2 = null;
     ...
4)     public void setCoin1(Coin cn)
     {
4a)       cn.setValue(3);
4b)       coin1 = cn;               // same as this.coin1 = cn
4c)       coin1.setValue(2);
          coin1 = penny;            // syntax error (Pocket doesn't know about penny variable)
```

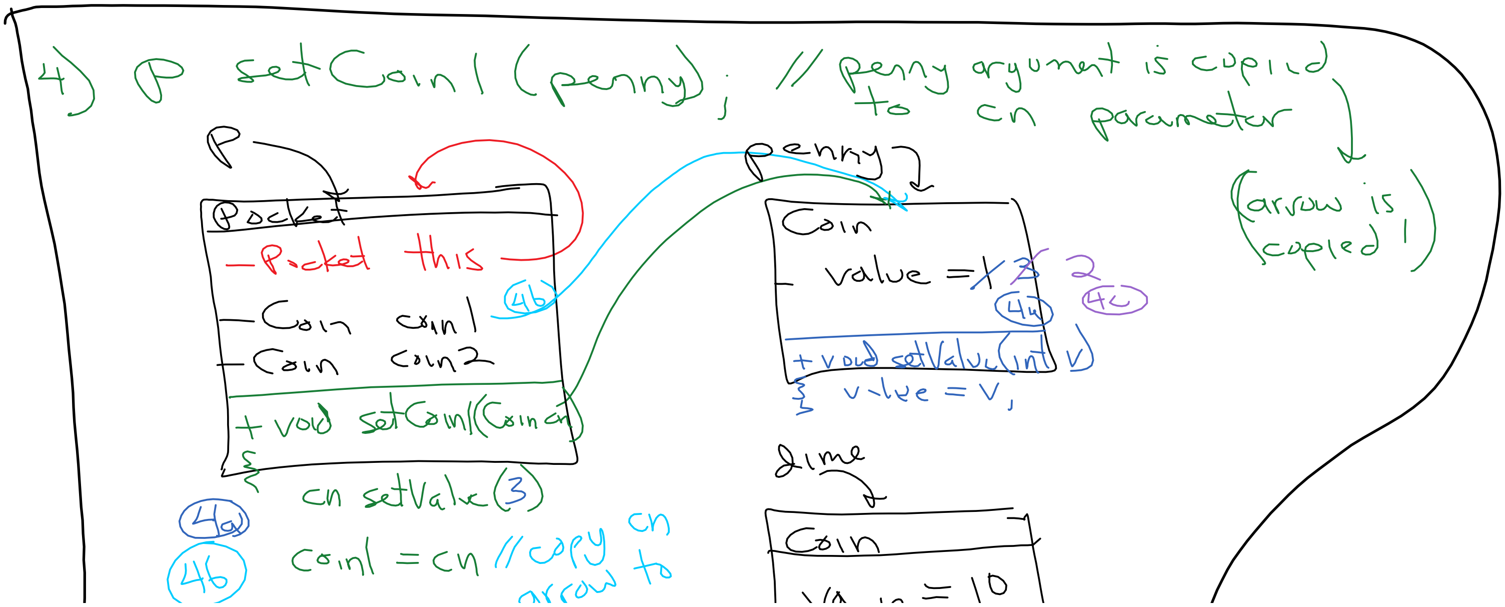
```
        coin1 = coin2;            // Ok, but dangerous as both slots refer to same coin
        coin1 = cn;               // Ok, corrected above problems
        cn    = null;             // OK since don't need parameter reference anymore
    }

6)  public void setCoin2(final Coin CN)
    {
6a)     CN.setValue(5);           // OK but dangerous as just changed the value of dime
6b)     coin2 = CN;               // same as this.coin2 = dime
6c)     coin2.setValue(7);        // OK but dangerous as just changed the value of dime
        coin2 = coin1;            // OK but now the dime is a penny;
        coin2 = CN;               // back to dime and coin2 having value 5
        CN    = null;             // illegal as CN can't reference anything other than dime
    }
}

public class Coin
{
    private int value;

    public boolean setValue(int x)    // the boolean return value can be ignored by
    {                                 // calling program (i.e. Pocket & Test classes)
        value = x;                   // same as this.value = x
        return true;                 // required since returning a boolean
    }
}
```

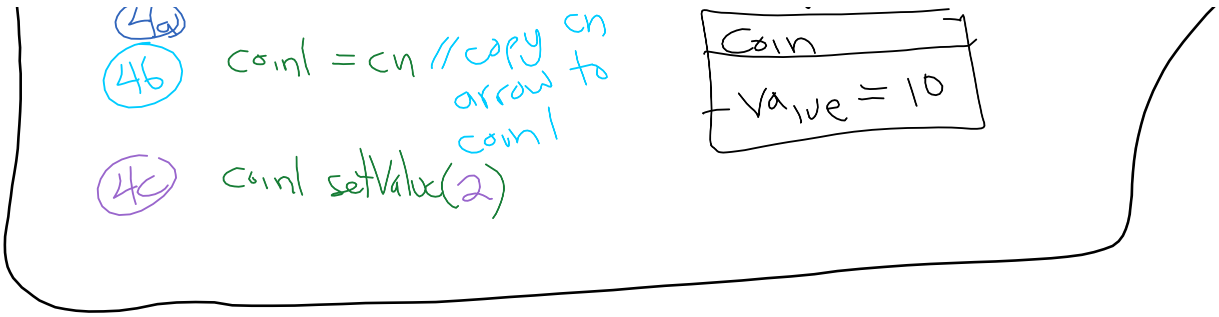
Note1: penny & dime are argument references. cn and CN are corresponding parameter references.

Note2: penny1, cn, and coin1 references are copies (i.e. they point to same place or address). But the object itself is NOT copied.









The above diagram does NOT include the final three statements of `setCoin1(Coin cn).` These 3 statements are left as classroom practice.

The final UML object Diagram for step 6 of the Pocket class (i.e. `public void setCoin2(final Coin CN)` is left as a classroom practice as well.

Syntax: Interfaces

Wednesday, June 1, 2022 12:08 PM

1. Interface declaration & usage

1. Syntax for declaration:

```
[public] interface IntefaceName [extends InterfaceName1, InterfaceName2, ..]
{
     feature1;
     feature2;
     ...
}
```

where each `feature` has the form: *modifiers method | field*

a. modifiers are **`public`**, **`static`**, and/or **`final`**
   i. modifiers are never necessary because:
      1) methods are always **`public`**
      2) fields are always **`public static final`**

b. method declaration has the form: **`Type methodName (parameter1, parameter2,…);`**

c. field declaration has the form: **`Type variableName = initializer;`**

2. Example

```
public interface Measurable
{
     double CM_PER_INCH = 2.54;         // assume public static final

     int getMeasure();                  // assume public
}
```

3. Implementing an Interface

```
public class InchRuler implements Measurable
{
     private int length_in_inches;

     public int getMeasure()                  // dont forget to make public
     {
          return length_in_inches * CM_PER_INCH; // downcast to int!!!!
     }
}
```

4. Notes

a. Interfaces can be thought of as “Abstract classes on steroids”

b. Interfaces are “agreed upon” set of method signatures that will be implemented by the individual programmers.

c. Converting from Class to Interface is legal (and automatic) as long as the Class implements the Interface….”upcasting is automatic”

d. Casting or converting from an Interface to a Class is legal (but not automatic) as long as the Class implements the Interface and the programmer KNOWS the object is of the desired class. “downcasting must be down explicitly and carefully”

e. See API List Interface – a List is a Collection of items where the items have a position. (i.e arrays, ArrayLists, LinkedLists, etc.)

5. Using and casting Interfaces and Classes

```
public class Test
{
     InchRuler ruler = new InchRuler(12);     // an 12-inch ruler object
     Measurable x = ruler;                    // ruler can be “upcasted” to Measurable
```

```
        System.out.println(x.getMeasure());      // ok
        System.out.println(x.getUnit());         // illegal, can only use getMeasure()
        InchRuler ruler2 = (InchRuler) x;        // OK, since we KNOW x is an InchRuler
        InchRuler ruler3 = (FootRuler) x;        // illegal unless FootRuler extends InchRuler
        Measurable y = new Measurable();         // illegal (can’t instantiate an Interface)
    }
```

Syntax: Generics (Java5)

Wednesday, June 1, 2022   2:04 PM

1. Generics Lecture (Java5—jre1.5)

Why:

Generics (aka Parameratized Types) solved a long-standing problem in Java when a project has multiple programmers over a long period of time (i.e. maintaining *legacy* code). Java 5 (aka jdk1.5) introduced the concept of Generics. This improvement moved a potential *runtime error* (i.e. ClassCastException) to a *compile-time error.*

Example:

When storing items (objects) in an array or ArrayList or any other JFC data structures (Maps, Lists, etc), the data is stored as type **Object.** However, other Java programmers may not know what Type of objects you may have really stored. As long as YOU know what you are storing and retrieving, then you can be safe when you CAST your data. However, if you are the one responsible for storing data but another programmer somewhere or sometime else is in charge of retrieving your data, then both programmers have a problem. How does the other programmer know what to cast?

Example using the pre-Java5 way that potentially generates a *runtime* error.

```
/** Method expurgate() is to remove a name (i.e. "Fred") from an ArrayList named c.  */
public void expurgate(ArrayList c)      //we don't know what Type of objects are in "c"
{
     for (int i = 0; i<c.size(); i++) {
       Object element = c.get(i);      // we know that the elements of c are Objects
       String name = (String) element; // are we SURE we can cast the element to a String?
       if ((name.equals("Fred"))       // If not, then we won't know until we run the program.
          c.set(i, null);              // code will compile as Object.java has an equals() method
     }
}
```

Example using the Java5 Generics (aka parameterized Types):

```
/** Method expurgate() is to remove a name (i.e. "Fred") from an ArrayList named c.  */
public void expurgate(ArrayList<String> c) //we now know what objects in c are Strings.
{
     for (int i = 0; i<c.size(); i++) {
       String name = c.get(i);     // we know that the elements of c are Strings. No cast needed
       if ((name.equals("Fred"))   // String.java has an overloaded equals(String) method!
          c.set(i, null);
     }
}
```

Notes:

1. When you see the code `<Type>`, read it as "of `Type`"; the declaration reads as "an `ArrayList` of `String c`."
2. The main advantages are:
   a. Declarations take longer to code but using the variables takes less typing.
   b. Using generics makes code clearer and safer, since compiler checks rather than runtime exception
   c. We have eliminated an unsafe cast and a number of extra parentheses
   d. We have moved part of the specification of the method from a comment to its signature, so the compiler can verify at compile time that the type constraints are not violated at run time.
   e. If the program compiles without warnings, we can state with certainty that it will not throw a `ClassCastException` at run time.
   f. The net effect of using generics, especially in large programs, is improved readability and robustness.
3. The main disadvantage are:
   a. It makes a bit more programming. We must add `<String>` to our declarations.
   b. Parameter type information is not available at run time...only at compile-time.
   c. Automatically generated casts may fail when interoperating with ill-behaved legacy (old) code

Examples for Using**** Generics (we won't cover Declaring/Designing Generic classes in APCS)

old way (legacy) *new Generic way*

```
-
private ArrayList   animals;        private ArrayList<Animal>     animals;
private Map         environment;    private Map<Location, Animal> environment;
```

Syntax: Java Statements

Wednesday, June 1, 2022 2:28 PM

Java Statements – Java comes with the following statements

1. ****** Declarations – already covered in previous sections**

   a) **Constants : `public static final int PI = 3.14159267;`**

   b) **Variables : locals and fields (or primitives and references)**

   c) **Classes**

   d) **Interfaces**

   e) **Methods**

   f) **Parameters**

2. ****** Expression / Assignments - already covered in previous sections**

   a) **Use of = or other expression (unary, binary, or ternary) followed by a semicolon**

3. **Flow Control**

   a) ****** Default is Sequential execution – one line after another from top of source code to bottom of source code**

   b) **Conditional**

      (2) ****** if** statement

      ```
      if (boolean expression) {...}
      ```

      (1) ****** if…else** statement

      ```
      if (boolean expression)
           {...}
      else
           {...}
      ```

      (2) **switch** statement – not tested on AP exam or this course (often has unexpected effects)

      ```
      switch (integer expression)
      {
                case intValue1: statement;
                case intValue2: statement;
                ...
                default: statement;
      }
      ```

   a) ****** Return statement**

   ```
   return primitiveValue or reference;
   ```

   d) **Throw statement – abruptly terminates current method and resumes control to closest (scope-wise) catch block**

   ```
   throws expression; // expression must evaluate an object of class Throwable
   ```

   a) **Break statement - not used on AP Exam or this course**

   ```
   break;     // exits out of local block
   ```

   e) **Continue statement**

   ```
   continue;    // skips to end of block
   ```

4. **Loop / iteration - recall *initialize Loop Control Variable, test LCV, update LCV* ******

   a) ****** For loop**

   ```
   for (init LCV; test LCV; update LCV) {...}
   ```

b) ** While loop**

```
init LCV;
while (test LCV)
{
             ...;
      update LCV;
}
```

c) Do…while loop – not used on AP Exam or this course

```
init LCV;
do
{
            update LCV;
             ...;
}
while (test LCV);
```

d) Iterator (interface) – demonstrated later if needed. Not used on AP Exam or this course

```
// iterators
ArrayList<String> names = ...;
Iterator it = names.iterator();  // it methods are : next() and hasNext()
while (it.hasNext())
      System.out.println((i.next());
```

a) ** Enhanced For-Loop (Java5—jre1.5) (aka "for each")**

```
Why:
      There are times when the "for loop" and "while loop" are more complex than needed,
      specially for JFC Collections (ArrayLists, Maps, Sets, etc.)
```

The pre-Java5 "For" looping over primitive arrays		**Java5 "For Each" looping over primitive arrays**
`// MUST explicitly initialize, test and update LCV`		`// no need to use an index`

`int values[] = ....;`
`int sum;`
`for (int i=0; i<values.length; i++)`
`      sum += value[i];`
		`int values[] = ...;`
		`int sum;`
		`for (int v : values)`
		`      sum += v;`

The pre-Java5 "For" looping over ArrayList		**Java5 "For Each" looping over ArrayList of Strings**

`ArrayList names;`
`for (int i=0; i<names.size(); i++) {`
`    String name = (String) names.get(i);`
`    System.out.println(name);`
`}`
		`ArrayList<String> names;`
		`for (String name : names)`
		`      System.out.println(name);`

The pre-Java5 "For" looping over Sets of names (unknown data structure)...???? means we don't know the correct method.		**Java5 "For Each" looping over a Set of names (unknown data structure)**

`Set names;`
`for (int i=0; i<names.????; i++) {`
`    String name = (String) names.????;`
`    System.out.println(name);`
`}`
		`Set<String> names;`
		`for (String name : names)`
		`      System.out.println(name);`

Notes on "For Each" loops:

(1) Code is easier to read (once you get used to it) and you wont need to scan the loop for index errors

(2) Good for "simple" and "common" looping constructs (i.e. WILL be given on the AP Exam)

(3) Don't use if need to do complex or unusual situations (i.e. traversing an array in reverse order, removing elements, etc.).

1. **** Method calls

a) Direct calls

b) Recursion or self-referencing (often unstable and results in infinite loop). Requires a terminating "base case"

2. Other statements

a) Block statement (using {…}). Implies another level of scoping rules

b) Import

c) Package

d) Try blocks (i.e. Exception handling) – not test on AP Exam

B. Special Topics

1. Exception Handling - to deal with runtime errors to avoid system crashes

a) Checked – must be "caught"

b) Unchecked

c) Example

```
try
{
     // statements that might cause a crash
}
catch (ExceptionType e)
{
     // handling of possible problem
     System.out.printl("error: "+e)
}
```

d) **** **Common Exceptions to be known for AP Exam**

```
NullPointerException;             // object not instantiated
ArrayIndexOutOfBoundsException;   // tried to access an element not within an array
ArithmeticException;              // divide by zero
ClassCastException;               // wrong class for downcasting
IllegalArgumentException;         // wrong argument for method call
IllegalStateException;            // JRE not ready (network error)
NoSuchElementException;           // iterator error
```

2. Event Handling – implementing ActionListener interface (i.e. actionPerformed(ActionEvent e) method)

3. Graphics

a) AWT – original graphics, OS dependant (use of Panel, Button, TextField, etc.)

b) Swing – 2nd generation graphics, less OS dependant, "lightweight" (use of JPanel, JButton, etc.)

(1) User Interfaces

a) Components : `JButtons`, `JTextFields`, etc

b) Containers : `JFrames, JPanels,` etc. (note Containers extend Components and therefore "are components")

c) Layouts : `GridBagLayout, FlowLayou`t, etc.

c) JavaFX - 3rd generation graphics. Separate library from standard edition (SE) of Java. Includes classes to be used with Internet/web pages (HTML, CSS, etc). Not used on AP Exam or this course.

4. **** Class hierarchy – inheritance (extending)****

a) **** **Overloading** – same method name but different signatures, can be determined at compile time

b) **** **Overridding** – same method signature as parent class, may be determined at runtime

c) **** **Polymorphism** – the ability to decide at runtime which method is to be used based upon the object's class at runtime. Also related to casting between levels within the hierarchy. (see `getClass()` method of `Object` class). The `this` object reference is critical to understanding polymorphism and require **UML object Diagrams**. The `super` references are set at compile-time and can be shown in **UML Class Diagrams** (see *lecture on Challenge: OOP in Java)*

5. Threads – not on AP Exam or this course… useful for animation and multiple programs running concurrently

Challenge: OOP in Java****

Thursday, June 2, 2022 9:27 AM

a. **Challenging OOP concepts:** `Object.java, this`**, and** `super`

i. The root (or top-level parent) object of a class hieararchy is always `Object.java`

1) If a class does not extend any other class (i.e. does not have a parent class), then the compiler will, by default, append `extends Object` to the class header. This has the effect of making `Object.java` the root (or parent, grandparent, great-grandparent, etc) of every class.
   a) Example: `public class Pet` will become `public class Pet extends Object`
   b) Example: `public class Pet extends Animal` where `Animal` may, by default, extend `Object`

2) `Object.java` has several methods including:
   a) `boolean equals(Object other)` compares (`this==other`) The references must point to the same object. Note that overriding and overloading the `equals()` method is difficult to do properly.
   b) `int hashCode()` converts the object to a (hopefully) unique integer. Sometimes interpreted as the index of the object stored in a Map or its location in memory)
   c) `String toString()` returns the name of the object's class, @, and the hashCode(). The toString() method is designed to be overridden to support programmers and users.

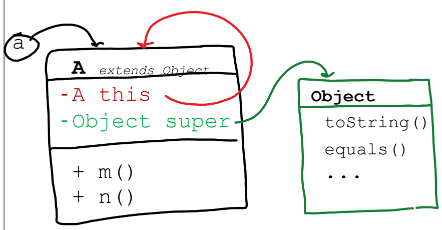
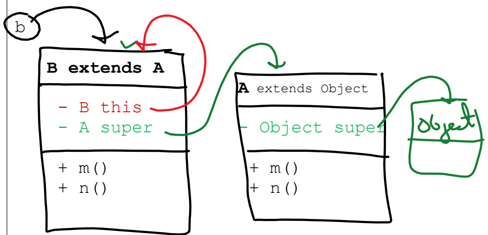
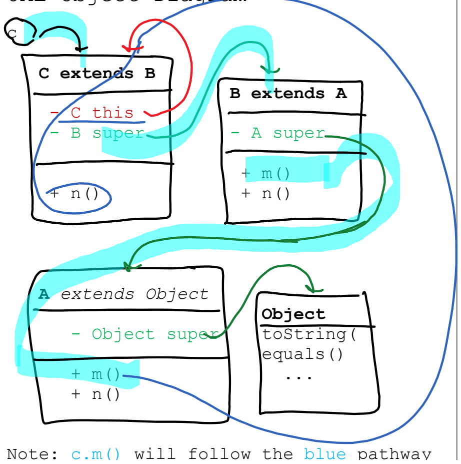
ii. The keyword (i.e. reference) **`this`** refers to the current object upon instantiation.

1) **`this`** is determined at runtime (i.e. dynamically) based on the actual class of the object instantiated.
2) For any instantiation (i.e. call to a classname's constructor), there is only one this referenced generated.
3) The feature is called *POLYMORPHISM : the ability at RUNTIME to determine which object is referenced by* **`this`.**
4) Code can be traced using a **UML object Diagram.**

iii. The keyword word (i.e. reference) **`super`** refers to the current instantiated object's parent..

1) The meaning of **`super`** is fixed statically, when the method is compiled. **`super`** refers to the superclass of the class the method is textually contained in.
2) Since **`super`** is fixed at *compile-time (as opposed to runtime*), code can be traced using **UML Class Diagrams**

public class **A** {
public void **m()** {
System.out.println("m in A");
this.n();} *//call to 1 instantiated object*
public void **n()** {
System.out.println("n in A");}
}
	public class **B extends A** {
	public void **m()** { *// c.m() starts here*
	super.m(); *// call to **A***
	System.out.println("m in B");
	}
	public void **n()** {
	System.out.println("n in B");}
	}
		public class **C extends B** {
		public void **n()** {
		System.out.println
		("n in C");
		}
		}

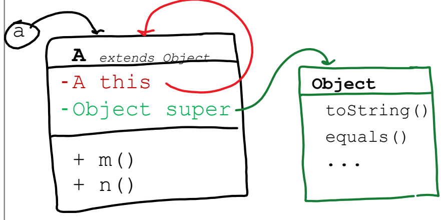
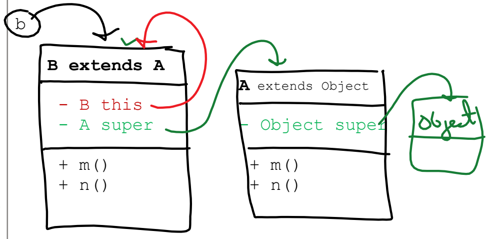
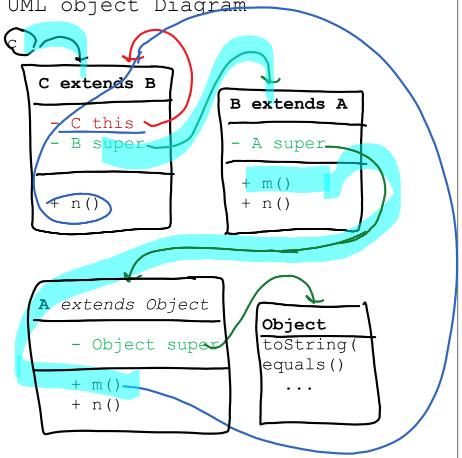
A a = new A(); a.m() results in output: m in A n in A	B b = new B(); b.m() results in output as: m in B n in C m in C	C c = new C(); c.m() results in output as: m in A n in C m in B *most interesting!!*
UML object Diagram  a A *extends Object* -A this -Object super + m() + n() Object toString() equals() ...	UML object Diagram (Object superclass not shown)  b B extends A - B this - A super + m() + n() A extends Object - Object super + m() + n() Object Note: since only object ***b*** was instantiated, there is only one ***this*** reference but several ***super*** references	UML object Diagram  c C extends B - C this - B super + n() B extends A - A super + m() + n() A *extends Object* - Object super + m() + n() Object toString(equals() ... Note: c.m() will follow the blue pathway

		Note: c.m() will follow the blue pathway

scrap paper : images+text

Thursday, June 2, 2022 9:22 AM

public class **A** {
public void **m()** {
System.out.println("m in A");
***this*.*n*();**} *//call to 1 instantiated object*
public void **n()** {
System.out.println("n in A");}
}
	public class **B extends A** {
	public void **m()** { *// c.m() starts here*
	super.m(); *// call to* ***A***
	System.out.println("m in B");
	}
	public void **n()** {
	System.out.println("n in B");}
	}
		public class **C extends B** {
		public void **n()** {
		System.out.println
		("n in C");
		}
		}

A a = new A(); a.m() results in output: m in A n in A	B b = new B(); b.m() results in output as: m in B n in C m in C	C c = new C(); c.m() results in output as: m in A n in C m in B *most interesting!!*
UML object Diagram  a A extends Object -A this -Object super + m() + n() Object toString() equals() ...	UML object Diagram (Object superclass not shown)  b B extends A - B this - A super + m() + n() A extends Object - Object super + m() + n() Object Note: since only object ***b*** was instantiated, there is only one ***this*** reference but several ***super*** references	UML object Diagram  c C extends B - C this - B super + n() B extends A - A super + m() + n() A extends Object - Object super + m() + n() Object toString(equals() ... Note: c.m() will follow the blue pathway