AP Computer Science: the Textbook

or

‘How I Learned to Stop Worrying and Love `Java`’

Written by Jason Eiben, Edited by his AP CompSci Students

Table of Contents:

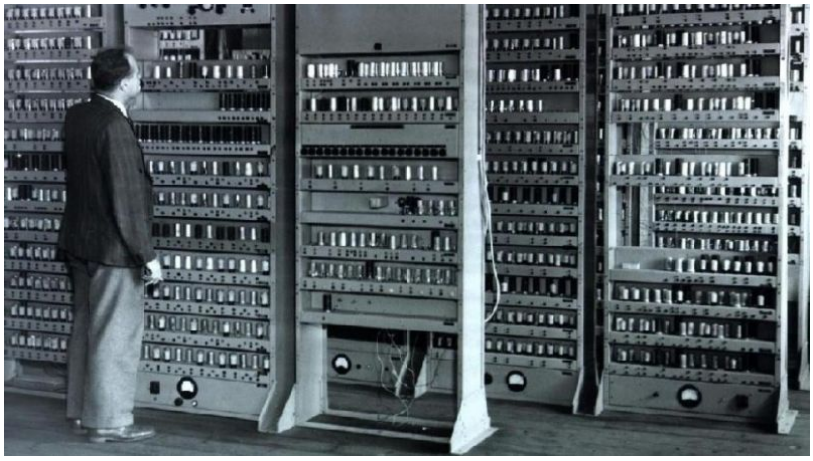
Unit 1 Cycle 0: Computers, Hardware & Software, Binary

Computers, Historically

Doing math by hand can be hard, or at the very least, annoying. Imagine that you are the captain of a 1700's ship that is sailing from Europe to the 'New World' across the Atlantic. Some smart people realized that, using the stars as indicators, a navigator can use a logarithm to determine a ship's latitude and longitude. If you haven't taken calculus yet, you probably aren't familiar with logarithms, but they can be complicated to computer. Unfortunately, most ship captains didn't take calculus either, so they were out of luck. Instead, they had to rely on huge books that people wrote that contained the math for entire oceans worth of calculations that mathematicians had already written. Double unfortunately, doing this complex math by hand can easily lead to errors, and if your calculations are off while you are sailing, you could easily sink your ship.

In an attempt to remove some of the complicated and tedious aspects of doing this type of math, and to remove as many errors as possible, people started to invent devices that could help a regular person do very difficult calculations. These devices included simple personal devices such as the abacus, and incredibly large and complicated machines such as the gear-based "difference engine" created by Charles Babbage and Ada Lovelace in 1849. On this machine, a person could physically input information, turn a crank many times, and the machine would display the result of calculating the equation. Despite how impressive this machine was, it was cheaper and easier to employ teams of people to do calculations, such as the team of women working in the Harvard Astronomy department known as "Pickering's Harem." For a very long time, 'computers' were actually just people who did complex math by hand.

During World War II, English mathematician Alan Turing was part of a team tasked with breaking the German Enigma code, which was being used to encrypt german messages. Unfortunately, this code changed every day using a device that made it nearly impossible to decipher by hand. Instead of using traditional methods, Turing built a machine which turned cylindrical rotors representing different letter combinations. This machine could try fully-decrypted solutions thousands of times faster than a human could by hand, and the eventual machine could crack the German code each day. Many historians suggest that the intelligence gained by this breakthrough allowed the Allies to win WWII. This would be the start of the electronic, digital age of computation.



**note: Alan Turing's contributions were incredible, but his life was filled with tragedy and struggle. He was persecuted for being gay, and committed suicide before the world would know the impact of his work.*

Hardware vs Software

Alan Turing's machine was built out of relatively standard electrical components that were being used in other applications, such as motors, switches, and lights. Putting all of these components together does not necessarily give you a computer that can actually do something, such as cracking the Enigma code. Instead, Turing needed to combine the components, and then teach the system rules that it could follow to do the calculations that he needed. Here we see an important difference when discussing computers - the distinction between *hardware* and *software*. Hardware is a term for the physical components that make up a computer system, including things like the CPU (central processing unit), HDD (hard drive, for storing data), RAM (random access memory, which lets the computer store information quickly and temporarily), a power source, keyboards, and monitors. In Turing's example, the hardware was built of motors, rotors, switches and wires. Without hardware, you can't do anything. On the other hand, just wiring up these components won't crack the code - Turing needed to give his system rules to follow in order to do the actual work. These rules comprise the *software* of his system.

One of the most important pieces of software on a modern computer is the *operating system*. An operating system makes it easy for users to interact with a computer. Early operating systems were simple interactive text windows, often referred to as a 'terminal', that a user could type in commands for the computer to follow. Such an operating system would have rules for how the computer should open files, run programs, and generally use the hardware in the system. You are probably more familiar with a GUI (graphical user interface) that is provided by a modern operating system such as Windows 10, Mac OSX, or Linux (a free, open-source operating system). A GUI uses images on the screen that a user can click on, which is usually more intuitive for the average computer user - everything that you see on a computer screen is designed as a GUI for you to interact with.

In addition to the operating system, software also includes programs or apps that you might install on your system. A game that you might install isn't part of the operating system (OS), but the OS allows you to install other software that the computer can run. These programs are written using different *programming languages*, such as Java. As long as a computer has the language installed, it can run a program written with it!

Consider your cell phone: It has hardware, such as the screen, processor, memory, and RAM. It also has software, including the operating system and any apps that you have installed. These apps are rules that the hardware will follow in order to do the things you want your phone to do, like making calls or playing games.

Hardware Specifics - Transistors & Memory

While the AP Computer Science test will not specifically ask you about hardware and software in general, it often does ask questions about the way that a program stores information in the memory of your computer. In order to understand the way a computer remembers, you first need to know how a computer thinks. The CPU of a computer is built of millions of very small *transistors*. A transistor is a very small switch that turns on and off - just like a light switch, there are only two options: on and off, 1 and 0. Every year, engineers and electricians figure out ways to make transistors smaller and pack more of them into smaller spaces, but the reason that a computer from today is faster than a computer made 10 years ago is that newer computers have more transistors. These transistors are used as the computer thinks, or calculates, following the rules defined in the software. What does this have to do with the computer's memory? Well, if the computer 'thinks' using 1's and 0's, it must remember information using this same format.

Memory Continued - Binary Numbers

Due to the on/off nature of transistors, computers store information in a **binary** way. In a binary system, there are only two working digits: 1 and 0 (think *bi*cycle, with two wheels). Unlike the *decimal* system, which uses 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9 - *dec* often denotes 10, such as in *dec*ade), binary numbers only consist of 0 and 1. Despite this extreme difference, you can still represent the same range of values in both systems. Consider the way that we already know to count in the decimal system:

0… 1… 2… 3… 4… 5… 6… 7… 8… 9…

What do we do when we run out of digits to use? We add a new digit in front and return the lowest digit to 0.

10… 11… 12… 13… 14… 15… 16… 17… 18… 19…

We ran out of digits again, so we increase the next level digit and continue.

20… 21… 22… 23… … 29… 30… 31… … 39… 40… 41… … 98… 99…

Once again we have run out of digits to use in both places, so what do we do? We add a new place and set the lower digits to 0!

99… 100… 101… … 198… 199… 200… … 998… 999… 1000

Binary numbers work exactly the same way, but we only have two digits to work with: 1 and 0

0… 1…

whoops, we already ran out of digits, so we need to add a new one and set the lower to 0:

0… 1… 10… 11…

we ran out of digits again, so we follow the same process: add a new digit, set the others to 0:

0… 1… 10… 11… 100...

Notice that 0 and 1 are the same values in both decimal and binary (often denoted with $_{10}$ and $_{2}$). The following chart illustrates the relationship between binary and decimal numbers:

Decimal	Binary
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Notice that each decimal place represents a power of 2:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	=	128	64	32	16	8	4	2	1

To convert from a binary number to a decimal number, you simply add the values listed below each 1:

0	1	0	1	0	1	1	0	=
~~128~~	64	~~32~~	16	~~8~~	4	2	~~1~~	~~128~~ + 64 + ~~32~~ + 16 + ~~8~~ + 4 + 2 + ~~1~~ = 86

What is 11111111_2 in decimal? 128+64+32+16+8+4+2+1 = 255. To go larger, we would need to add a new digit: 100000000_2 = 256, or 2^8.

Memory Continued - bits, bytes … gigabytes

So a computer stores information as 0's and 1's, but that might not explain how you computer does things like show you pictures or let you play Pokemon Go. Why don't you just see 0's and 1's on the screen like in *The Matrix*? Well, consider the colors that make up a picture on your screen. Your computer stores the color of each *pixel* (tiny squares that make up your screen) as a binary number. Let's imagine that you have 256 colors to work with in your picture, so each pixel could be represented as a binary number with 8 digits (because 00000000 = 0, and 11111111 = 255, which is a range of 256 numbers). 00000000 might represent **black**, which 11111111 represents *white*, and numbers in between represent other shades of the rainbow. In this case, each pixel is represented with 8 *bits* of information (a single binary number = 1 bit). When you have 8 bits together, this is known as a *byte* of information. If you have a big enough picture with enough pixels, it might get crazy to describe it with bytes - if each pixel is 1 byte, what if your picture has thousands of pixels (and it probably does)? You can describe it with *kilobytes* (1 KB = 1024 B), *megabytes* (1 MB = 1024 KB), *gigabytes* (1 GB = 1024 MB), or even *terabytes* (1 TB = 1024 GB). Why 1024? Well, 1024 = 2^{10}, so each level is one level of magnitude higher than the next.

Why is this relevant? Well, remember that each piece of information is stored in your computer's memory as a 1 or a 0. Consider a 1GB movie on your phone - how many 1's and 0's is that? Well, 1 GB = 2^{10} MB = 2^{100} KB = 2^{1000} B =
107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837881569585812759467291755314682518714528569231404359845775746985748039345677748242309854210746050623711418779541821530464749835819412673987675591655439460770629145711964776865421676604298316526243868372056680693 76 bytes. Multiply this giant number by 8, and you have the number of bits in a gigabyte. That's a lot of 1's and 0's!

This means that somewhere on your phone, a location in the memory is holding 2^{1003} values, either 1 or 0. To watch the movie, your phone *reads* this combination and converts each pixel into a color, over and over to display the full movie on your screen!

Memory Continued Again - Hexadecimal Numbers

Similar to binary numbers, another counting system is sometimes used when working with computers: hexadecimal. Where $binary_2$ has 2 digits and $decimal_{10}$ has 10 digits, $hexadecimal_{16}$ has 16 digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F G

The counting works the same way with decimal and binary - you use digits until you run out of them, and then you go back to zero and increase the next position:

0... 1… 2… 3… 4… 5… 6… 7… 8… 9… 0… A… B… C… D.... E… F… 11… 12… 13…

This means that $10_{16} = 17_{10} = 10001_2$

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
0	0	11	17	...	...
1	1	12	18	AE	10*16 + 14 = 174
2	2	13	19	AF	10*16 + 15 = 175
3	3	...	...	B0	11*16 + 0 = 176
4	4	1A	16 + 10 = 26	B1	11*16 + 1 = 177
5	5	1B	16 + 11 = 27	...	...
6	6	1C	28	F9	15*16 + 9 = 249
7	7	1D	29	FA	15*16 + 10 = 250
8	8	1E	30	FB	15 * 16 + 11 = 251
9	9	1F	31	...	...
A	10	20	2*16 + 0 = 32	FF	15*16 + 15 = 255
B	11	21	33	100	1*256 + 0 + 0 = 256
C	12	...	...	101	1*256 + 0 + 1 = 256
D	13	9E	9*16 + 14 = 158	...	...
E	14	9F	9*16 + 15 = 159	10E	256 + 0 + 14 = 270
F	15	A0	10*16 + 0 = 160	10F	256 + 0 + 15 = 271
10	16	A1	10*16 + 1 = 161	110	256 + 16 + 0 = 272

Notice the digit pattern:

____	____	____	____	=	__2__	_C_	__7__	__B__
16^3	16^2	16^1	16^0		4096	256	16	1

$2C7B_{16}$ = 2*4096 + C*256 + 7*16 + B*1 = 8192 + 3072 + 112 + 11 = 11387_{10}

Binary questions do appear on the AP Test, and on the 2016 test there was a question that required students to know the difference between notation for binary and hexadecimal numbers. More frequently, you will see hexadecimal used when adding styles to websites, games, and graphical programs, because color codes are often expressed as hexadecimal (notice that 4 digits of hexadecimal can represent 16^4 - 1 colors, or 65535. This allows for many more shades of color than 4 digits of binary, which would only be 2^4 - 1, or 255 colors).

The information in this packet provides a *very* brief introduction to the way that computers work. As you study computer science, you will find that computers are great at doing many things, such as: performing millions of calculations in a short time, remembering vast amounts of information with no errors, sorting and combining information to produce new results. At the same time, computers are exceptionally bad at other tasks that you can do with virtually no effort: understanding human language, identifying implied meanings, identifying faces in a crowd, participating in "creative" endeavors, just to name a few. The challenges that we face as computer scientists will be to leverage the strengths that computing systems have, while also finding new ways to *teach* our computers to do things that they couldn't do before. It was not long ago that it seemed impossible for a computer to hear, understand, and react to the human voice, but today you can speak to your phone with generally accurate results. While there is still a long way to go, starting your journey with Java is an excellent way to gain the tools that will be required for future discoveries!

The following resources might be helpful if the descriptions below are confusing or you want further explanations for any topic:

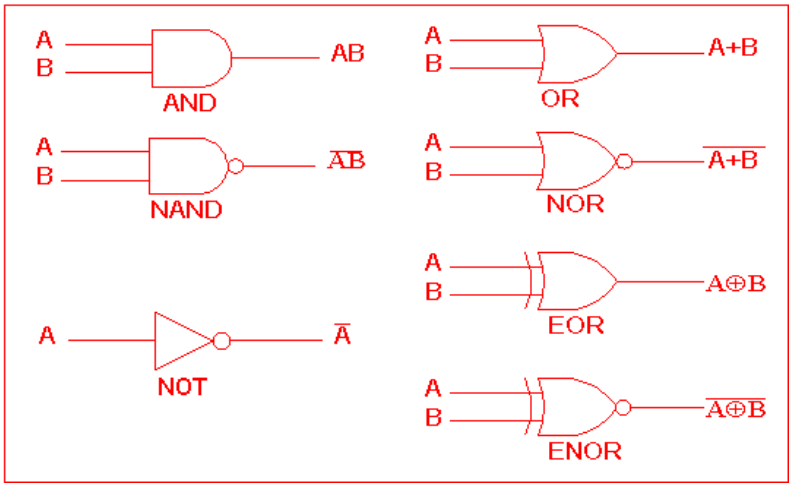
https://en.wikipedia.org/wiki/History_of_computing
http://www.diffen.com/difference/Hardware_vs_Software
https://www.mathsisfun.com/binary-number-system.html
http://www.bbc.co.uk/education/guides/zwsbwmn/revision/1
http://seggleston.com/1/web-development/hex-numbers
http://www.w3schools.com/colors/colors_picker.asp

Unit 1 Cycle 1: Java, the Java Compiler & Eclipse

Welcome to AP Computer Science! The majority of this course will focus on writing programs using the Java programming language. While most of the structures and concepts that we will cover are applicable in many other programming languages, such as Python or C++, the AP test consists only of code written in Java. Before we can actually start reading and writing Java code, however, there are a few things that you need to know about this language and the programs that run it.

Machine Language & Programming Languages

In the bootcamp readings, we learned about the physical processes that transistors and other hardware experience when a computer is in use. We saw that a CPU can respond to an input of 1's and 0's and follow the instructions in a program as a result. You might be wondering, however, how the computer even knows how to follow instructions at all - that's where *machine language* comes in. On the earliest computers, engineers wrote instructions in the only language that the computers could understand: binary. Using binary, they could construct what is known as a **logic gate**, which takes in input and, depending on the logic, spits out some output. For example, an AND logic gate takes in two pieces of data as input. If both of those inputs are a 1, it also spits out a 1. If one or both of the inputs are a 0, the gate spits out a 0.



Other gates were also constructed, such as OR gates, which return a 1 if either of the inputs are a 1, and NOT gates, which return the opposite of the input. By combining these operations, programmers were able to recreate the most basic mathematical and operational tasks that computers do today.

This was great, but reading and writing instructions in binary is nearly impossible. So programmers developed a tool called *assembly language*. These languages basically are wrappers that look more like human language, and were easier to write. When executing code from an assembly language, the computer uses an interpreter to translate this code back into machine language.

To the right is an example of a program written in an assembly language for a Linux computer. While this is definitely easier to understand than a string of 0's and 1's, it isn't particularly easy to use either.

```
.global _start

.text
_start:
    movl   $4, %eax
    movl   $1, %ebx
    movl   $msg, %ecx
    movl   $len, %edx
    int    $0x80

    movl   $1, %eax
    movl   $0, %ebx
    int    $0x80
.data
msg:
    .ascii  "Hello, world!\n"
    len =   . - msg
```

In an effort to make programming easier, *high-level programming languages* were created. Any modern programming language that you might learn is considered 'high-level' (even something that is considered very old today, such as FORTRAN, which was created in 1956), since they act at a higher, more abstract level than assembly language and machine code. Basically, you install the rules governing a high-level programming language, such as Java, on your computer, so that your computer knows how to translate this language into assembly code, and then finally into machine language. All that you need to do is install the languages that you want to write with and your computer can execute those programs!

Interpreted vs. Compiled Languages:

There are several different kinds of high-level languages. In order to talk about these, you'll find it useful to bear in mind that the *source code* of a program (the human-created, editable version) has to go through some kind of translation into machine code that the machine can actually run.

Compiled languages

The most conventional kind of language is a *compiled language*. Compiled languages get translated into runnable files of binary machine code by a special program called (logically enough) a *compiler*. Once the binary has been generated, you can run it directly without looking at the source code again. (Most software is delivered as compiled binaries made from code you don't see.)

Compiled languages tend to give excellent performance and have the most complete access to the OS, but also tend to be difficult to program in. A language called C is by far the most important of these (with its variant C++). FORTRAN is another compiled language still used among engineers and scientists but years older and much more primitive. COBOL is very widely used for financial and business software. There used to be many other compiler languages, but most of them have either gone extinct or are strictly research tools. If you are a new Unix developer using a compiled language, it is overwhelmingly likely to be C or C++.

Because a compiled language needs to be fully compiled before it can be run, the compiler can find errors that make a program un-executable. You may be familiar with the .exe file format on Windows computers. A .exe file is 'executable', and usually opens a program when you run it. You can't see the source code for such a problem, because it is already compiled and ready to run.

Interpreted languages

An *interpreted language* depends on an interpreter program that reads the source code and translates it on the fly into computations and system calls. The source has to be re-interpreted (and the interpreter present) each time the code is executed. Interpreted languages tend to be slower than compiled languages, and often have limited access to the underlying operating system and hardware. On the other hand, they tend to be easier to program and more forgiving of coding errors than compiled languages. Historically, the most important interpretive language has been LISP (a major improvement over most of its successors). Some popular and newer interpreted languages include Python, Ruby, and JavaScript (which is not the same as Java!)

Reflection: *Based on this reading, what is the difference between a compiled and an interpreted language?*

The Java Programming Language

http://www.oracle.com/technetwork/java/compile-136656.html#platform

The Java Programming Language is owned and maintained by a company named Oracle. In order to run programs written in Java, you will need to have the following components installed on your computer:

- The Java Application Programming Interface (API)
  These are libraries of pre-compiled code that you can use in your own programs. This lets you add ready-made functionality to your programs. One example is the function that lets you 'print' messages on the screen. If you didn't use the Java API, you would need to teach the computer to 'print' yourself!

- The Java Compiler
  Java is a compiled language, so your computer needs this compiler so that it can translate your high-level code into machine-readable code. You can specifically ask the compiler to check and compile files on your computer before running them.

- The Java Virtual Machine (JVM)
  *Once the compiler has compiled your program, the result is a collection of **bytecode**, which is basically the term for the machine-readable code for Java. Instead of having your computer run this bytecode, the Java Virtual Machine runs it - your computer has a virtual computer inside of it, and this virtual computer executes your program! One benefit of this is that, regardless of what type of computer you are using, the JVM is the same for everyone. This means that a program written in Java, compiled, and run in the JVM will act the same way on a Windows PC, Mac, or Linux computer!*

Installing Java

Java is free to download and use on your computer. You can visit https://java.com/en/download/ to download and install the most up-to-date version. The school computers will automatically be updated, and many computers are sold with Java pre-installed. If you are using a computer that is not yours, be sure to ask for permission from the owner before installing any software!

You might be wondering what it means for a programming language to be "up-to-date" - how can it change? Well, frequently the groups that maintain a language will constantly be working to make the language better. New versions of Java, for instance, often have more efficient compilers, or have new built-in tools in the Java API that programmers can use in their own code. If you are using an older version of Java, don't worry - the AP test uses a subset of the Java API that is included all currently-running versions of Java!

***Reflection:** Imagine that you have written a program in Java. Based on this article, how does your computer know what to do with the instructions in your code? Be sure to explain the role of the JVM!*

Writing Java Code

So you have Java installed and you are ready to start programming - but how do you even do that? Well, you could technically write code following the rules of the Java language in any text editor and save your file as `filename.java` (notice the font change - this is a monospace font, which means that every letter and symble takes up the same width. All examples of code and file names will be written in a monospace font, which are easier to read for programming!). By attaching the `.java` file extension to your file, you are indicating that this is a Java file and should be compiled using the **javac** compiler and run on the JVM. This is similar to the way that a `.doc` file is a document file that should be opened by a text editor like MicroSoft Word.

More often, programmers use a tool called an Integrated Development Environment (IDE). An IDE is basically a text editor that is designed for writing code - it will color-code different features of your programs, help you to organize your code, and even have built-in tools for finding and fixing bugs. We will be using the Eclipse IDE (https://eclipse.org/downloads/), which is a free, professional, and powerful tool for writing Java code. Eclipse makes the process of writing, testing, and debugging code more efficient, and even includes built-in tools to quickly compile and test your code all in one program window.

Basic Java Syntax

In the next cycle, we will dive into the specifics of Java **syntax**, or the rules that you must follow to write working code. Unlike spoken languages, where breaking the rules of a language won't necessarily cause your conversation to break completely, failing to follow the syntax of a programming language will completely stop your program. The following syntax rules will be needed for our first lab:

1) A Java document (`.java`) starts and ends with the definition of a `class`:

```
class MyFile{

}
```

This class name MUST match the name of your Java file, in this case `MyFile.java`. In addition, notice the opening and closing curly braces `{}` - everything between these braces will be included in your program!

2) If a Java program does not have a 'main' method, it won't do anything:

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

      }
}
```

Don't worry about what this all means - just include it in your programs for now. Any code that you write needs to be inside of this main method if you want to to run when you compile!

3) In Java, individual instructions are called ‘statements’, and each complete statement ends with a semicolon (`;`)

```
class MyFile{
      public static void main(String[] args){
            System.out.println(“boo!”);
      }
}
```

Every time you tell your program to DO something, put a semicolon at the end!

4) In Java, text following two forward slashes `//` are called comments.

```
class MyFile{
      public static void main(String[] args){
            System.out.println(“boo!”); //this prints ‘boo!’ to the screen
      }
}
```

Comments can be used to leave reminders for yourself, outline a program, organize your code, and ask other programmers for help on shared projects. You can also quickly ‘comment out’ a line of code by putting `//` in front of a line - this would disable the line of code without deleting your work!

5) Java is ‘white-space agnostic’, which means that spaces and new lines don’t actually have an impact on the way that your program compiles or executes.

```
class MyFile{
      public static void main(String[] args){
            System.out.println(“boo!”); //this prints ‘boo!’ to the screen
      }
}
//runs the same as:

class MyFile{ public static void main(String[] args){
System.out.println(“boo!”); //this prints ‘boo!’ to the screen}}
```

Nevertheless, you should use indentations to keep your code organized and easy to read. Eclipse will automatically help you indent blocks of code!

Unit 1 Cycle 2: Java Syntax Basics

Reserved Words

While you will generally be able to create your own names for variables, methods, and classes in your programs, certain words are considered 'keywords' in Java and cannot be used outside of their prescribed use. For example, you might want to name a variable `long` because it describes the longest object in a group, but this word is reserved in Java to describe a specific data type. If you tried to name your variable this way, it would create an error because the Java Compiler would think that you were trying to name the data type instead of creating a new variable name (instead, you might name your variable `longest` to avoid this issue!)

The most common reserved words include:

```
abstract   assert     boolean    breakbyte       case       catch      char
class      const      continue   default         do         double     else
enum       extends    final      finally         float      for        goto
if         implements import     instanceof      int        interface  long
native     new        package    private         protected  public     return
short      static     strictfp   super           switch     synchronized
this       throw      throws     transient       try        void       while
```

Statements

A Java program is comprised of *statements*. A statement is a line of code that the program will execute, and each statement is an instruction to the program to do something. All statements in Java end with a semicolon (`;`), which indicates that the statement is complete. Following a semicolon, your next line of code should start on a new line. Note that you can include comments after the end of a statement!

Examples:

```
1     String name = "Stuart";     // a complete statement
2     int age = 17;
3     if (age >= 18){             //this is not a statement
4          System.out.println(name + " is getting old!");
5     }
```

In the example above, complete statements can be found on lines 1, 2 and 4. Notice that lines 3 and 5 are not complete statements because they do not end in a semicolon. The `if` structure on line 3 creates a condition that might allow the statement on line 4 to run, depending on the value of `age`.

Reflection: *Write a statement in Java, and underline any reserved words*

Data Types

Java is a 'strongly-typed' language!

This means that you will need to specify what type of information your program will take in, manipulate, create, or return. The following chart describes the most common data types that you will deal with on the AP Computer Science A exam:

Data Type	Example(s)	Notes
`int` *	0, 1, 5; a whole number in the range { 2^{-31}-1 … 2^{31} }, or { -2147483647… 2147483648 }	Integers have NO decimal place
`float` *	0.0, 1.23, 397.000001; represents up to 32 bits of accuracy. Decimal values are 'floating-point' numbers	not commonly used since it doesn't store values as detailed as `double`
`double` *	0.0, 1.23, 397.0000000000000001; represents up to 64 bits of accuracy.	most commonly used decimal value data type.
`boolean` *	`true` or `false` (no other values can be represented by a boolean!)	notice lowercase! Comparisons evaluate as boolean values
`char` *	'a', 'b', '4', '9', '!', '>'; a single graphical symbol. Notice they are represented between quotation marks	a `String` that contains a single letter is NOT a `char` !
`String`	"Hello", "What a nice day!", "365", "Q", "It doesn't matter how long it goes. Or how many sentences it contains. This is all one String!"	Not a primitive data type, so items of the `String` data type are objects, and have special methods: `.length()`, `.substring()`, `.toUpperCase()`
***** *denotes a primitive data type, which has no built-in methods. Primitive data types are not considered objects, unlike all other types of data in Java.*		

The Assignment Operator

When you want to store information, you will need to instantiate a variable of the correct data type:

```
<data type> <variable name> = <expression>;
```

Example:

```
int age = 99;
double price = 104.32
boolean hasMoney = false;
String name = "Stuart";
int num;   //notice you can instantiate a variable without a value
num = name.length(); //assigns the integer value 6 to the variable
```

Reflection: create two variables of different data types

Arithmetic

You can process information by using basic arithmetic symbols with your data:

`+`	addition	*combines numerical data types, concatenates text-based data types*
`-`	subtraction	*finds the difference between numerical data types*
`/`	division	*returns the result of dividing the first term by the second*
`*`	multiplication	*returns the result of multiplying the terms*
`%`	modulus	*returns the remainder that results from dividing the first term by the second*
`()`	parenthesis	*can be used to influence the order of operations*

Examples:

```
int numA = 7;
int numB = 15;
int numC = numA - numB;                 //this would make numC = _______
int numD = numB % numA;                 //this would make numD = _______
int numE = numA % numB;                 //this would make numD = _______
int numF = ((numB - 3) * 2) + numC);    //this would make numF = _______
String wordA = "Hello";
String comboWord = wordA + wordA;       //this would make comboWord = ________
```

Integer Division

While the examples above show some logical arithmetical behavior, you might be surprised by the following:

```
int x = 5;
int y = 2;
int z = x/y;
System.out.println(z);      //this would print the int 2 to the console
```

You might have expected `z` to hold the value 2.5, this is not the case. When you divide two `int` values, the result is also an `int` value. When executed, your program will round down to the nearest `int` value when you divide two integers. To avoid this you could do either of the following:

```
double q = 5.0;                  or          z = ((Double)x)/y;
z = q/y;  // z now = _______                 // z now = _______
```

In the first example, when either the numerator or denominator are `double`s, the division results in a `double`. In the second example, 'type casting' is used to convert `x` into a `double` before executing the division. Notice that we are casting `x` as a `Double` with a capital 'D'.

- Casting does not change an object, just the type used to reference it.
- Casting from one primitive type to another may cause some loss of precision.
- When casting an object, if the cast is not legal then a `ClassCastException` is thrown.

Increment and Decrement

```
int x = 12;       int y = 4;
```

Operator	Description	Example	X or Y now equals...
++	Adds one to the variable and stores and stores the new value in that variable	`x++`	
--	Removes one from the variable and stores the new value in that variable	`y--`	
+=	Adds the specified number to the variable and stores the new value in that variable	`x += 5`	
--	Removes the specified number to the variable and stores the new value in that variable	`y -= 5`	
***=**	Multiplies the variable by the specified number and stores the new value in that variable	`x *= 2`	
/=	Divides the variable by the specified number and stores the integer result in that variable	`y /= 2`	

Reflection: if `int x = 10;` and `int y = 3;` what does `double z` evaluate to below? Why?

```
y++;
x/=y;
double z = x / y;
```

Methods / Functions

Instead of repeating yourself, use methods to practice 'abstraction'

A method is a block of code that can be executed more than once inside of your program. This allows you to define a complicated set of steps that the program will follow, and then reuse them without having to write them over and over again. While we will cover methods in much greater detail in the next cycle, for now it is enough to know that method definitions, as seen below, include the following: 1) the keywords `public` and `static`, a data type (such as `int`, `String` or even `void`), a method name, and some optional parameters. Once a method is defined, it can be used again as many times as you would like in your program.

```
public static boolean isEven(int num){
      return (num % 2 == 0);
}
//a method named isEven, which is public, static, returns a boolean value,
//and takes in a single parameter called num
```

With this method defined, we can now *'call'* the method by stating it's name and providing any parameters:

```
//a static function can be called any time after it is created
boolean answer1 = isEven(34); //calling the method defined above
boolean answer2 = isEven(33);
```

What would be the values stored in `answer1`? Well, `answer1` returns whatever the value is for the expression `(num % 2 == 0)`, where `num` is equal to 34. 34 % 2 **is** 0, so this method call would return true, which means that answer1 will store the value true.

```
public static void sayHello(String name){
      System.out.println("Say hello to " + name + "!");
}

sayHello("Mr. Eiben");
sayHello("Adelle");
```

In the `isEven` method above, notice the keyword `return`. In a method, if a statement starting with `return` is reached, the value that follows the keyword return becomes the *output* of the method. For example, calling `isEven(33)` *returns* the value false. That means if we call:

```
System.out.println(answer2);
```

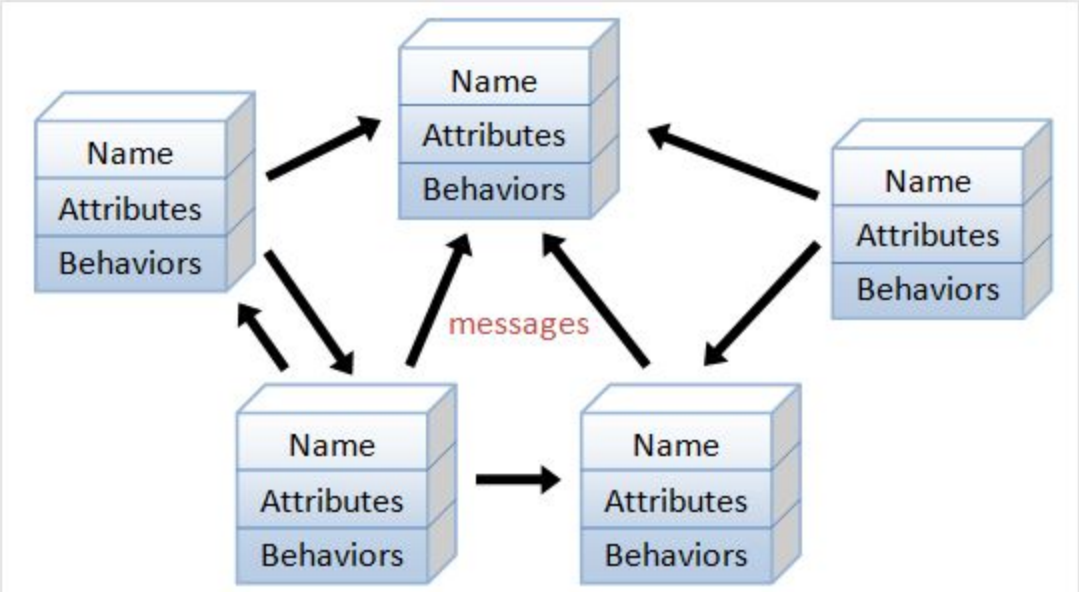
The value "false" will be printed to the console. In other words, a method call can be considered equal to whatever value it returns! Notice, however, that not all methods return a value, such as `sayHello` above, which is a `static` method. This means that nothing will be returned by the method. If you write a method that does not return the type of data specified in the method definition, you will get an error!

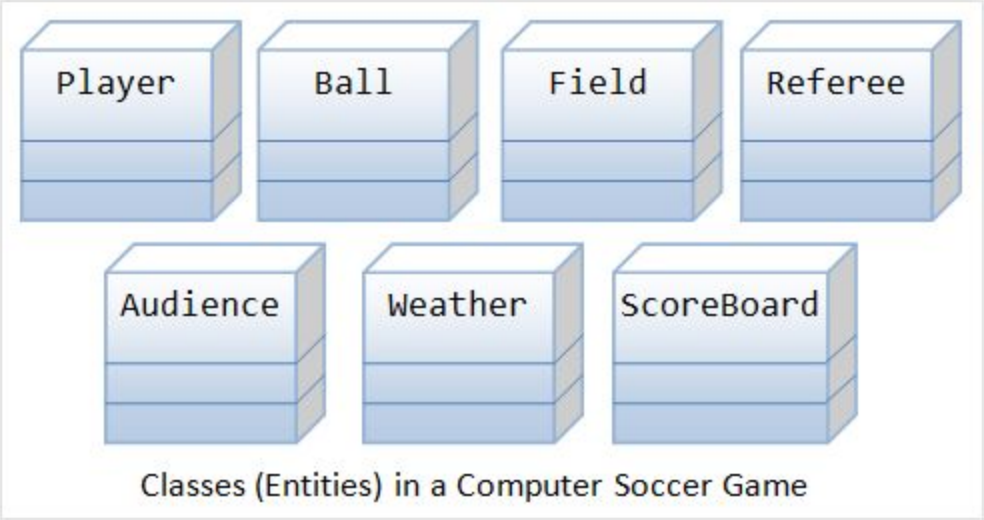
Unit 1 Cycle 3: Object-Oriented Programming

OOP: Object-Oriented Programming

Suppose you have a very specific problem: you have five lists of values, each with a length between 100 and 1000 elements. You want to write a program that takes all of these values and sorts them in order from smallest to largest. There are many strategies that a computer scientist might take to solve this problem. In one strategy, you could write instructions for the first list of numbers that sorts it in ascending order, then you could write similar instructions for the other four lists. Now with these sorted, you could write instructions that would compare the first items of each newly-sorted list and find the smallest value, and put that value in a final list. Then you could write instructions to repeat this comparison over and over until each number has been placed in the final list. Our basic strategy here is to treat the data and the operations as two unrelated ideas: on one side we have the lists (the data), and the other we have the instructions for sorting the items (the operations).

This approach closely resembles what is known as 'procedural programming', where specific functions are written and executed in a specific order. While this approach can lead to working solutions, it is also difficult to plan and visualize. Even worse, it is hard to take this solution and adjust it to solve other similar problems. Instead, you might consider taking the 'object-oriented' approach. In Object Oriented Programming (OOP), programmers create *classes*, which contain both data and operations. Based on the example above, we might create the `NumberList` class, which contains a list of numbers as a *parameter* (data), and some *methods* (operations or abilities) that things of this type can perform. By approaching the problem this way, it becomes much easier to reuse and adapt this code to solve other, different, or larger problems.





Classes (Entities) in a Computer Soccer Game

In the example above, you might be creating a computer program that simulates an entire soccer game. In this program, you will be creating many different objects from many different classes, such as the `Player` and `ScoreBoard` classes. Imagine that objects of the `Player` class have attributes (known as *fields* in Java) such as *name* and *number*, and a method such as *shootBall()*. When this method is executed, an object of the `ScoreBoard` class might update one of its attributes, *team1Score*. In this example, the full program is made up of objects that interact with each other. Each object belongs to one or more classes, and these classes define how the objects work.

When we write programs in Java, we will be following the OOP approach, which means that you will need to be very familiar with the way that classes are defined, created, and used so that you can solve complicated problems!

Class Overview

In Java, *everything* is an object. What this really means is that all of the documents that you write, all of the code that belongs to the Java language, and all of the code that you might import or borrow into your programs, belongs to the `Object` class. We won't get into the specific details now of the Java class hierarchies, but at this moment you simply need to know the following facts:

- Information is stored and manipulated in *classes*, such as the String class
- Objects that belong to a class have that class as a data type
  - `String word = new String("hello");` *`//word`* *is an object of the String class*
- If a class has fields, any object belonging to that class also has that field
  - `word` has one attribute/field, which is its value, "hello"
- If a class has any methods, any object belonging to that class can call those methods
  - `word.length()` *//this calls the length() method, which is defined in the* *`String`* *class*

As you can see, the code that we have been writing already has classes in it, and now we will learn how to define and use our own classes!

Class Definitions

In Java, each class is its own document. This means that in an IDE like Eclipse, a single program can be made up of many different files, each representing a class. In order to define a class, you must first declare it as `public` or `private` (all of our classes will be public, otherwise we could not use them in our programs), then you provide the name of the class starting with a capital letter, followed by a curly bracket (`{`):

```
public class Dog {
      //all of the the information defining the dog class will go here
}
```

Fields / Member Variables

Often, you will be creating classes in order to store and use information. In a Java class, we call these pieces of information *fields*, but you might also see them referred to as *member variables*. When you declare a field for a class, you are saying that any object that belongs to this class will store such a value:

```
public class Dog {
      private String name;
      public int age;
      private boolean hasFleas = false;

      //other information defining the dog class will go here
}
```

In the example above, three fields are defined. The first field, `name`, will have the type `String`. Notice that this field is declared as `private`, which means that only instances of this class, `Dog`, have access to this information, as it is not publically available. Also notice that this field has no default value (since each dog will have its own name.)

On the other hand, the `age` field is `public`, which means that any other class can ask a `Dog` object for its `age`, and it will give that information away. Lastly, notice the field `hasFleas`: it is `private`, like `name` was, but it also has a default value defined. This means that each `Dog` object will automatically have this value when it is created.

Constructors

In order to actually create instances (examples) of an object that belong to a class, you need a way to construct them. Earlier in this reading, the example `String word = new String("hello");` was used. In this example, we are calling the *constructor* for the `String` class in order to create a new `String` object. If we want to make our own classes, we will need to write our own constructors, which are the rules that define how instances of a class are created:

```
public class Dog {
      private String name;
      public int age;
      private boolean hasFleas = false;

      public Dog(String n, int i){
            name = n;
            age = i;
      }
      //other information defining the dog class will go here
}
```

Here we have added a constructor to the `Dog` class. To use this constructor, we would need to provide a `String` for the dog's `name` field, and an `int` for the dog's `age` field. When we create a new `Dog` object, we will pass these values in as *parameters*. Note that in order to use this constructor, we must provide exactly the same number, type, and order of parameters that are listed in the constructor:

```
Dog doggy1 = new Dog("Scooby Doo", 9); //these parameters will be stored as fields
```

This line might be called inside of another class, such as `TestDog`. But what if we want to add a new dog to our program but we don't know its name or age? We might want another constructor to deal with this possibility:

```
public class Dog {
      private String name;
      public int age;
      private boolean hasFleas = false;

      public Dog(){
            name = "Spot";
            age = 3;
      }
      public Dog(String n, int i){
            name = n;
            age = i;
      }
      //other information defining the dog class will go here
}
```

This means that we can state `Dog doggy2 = new Dog();` This is called a *"no-args" constructor*.

With these constructors defined, we have created two Dog objects. Imagine that you tried to call the following statements: *(note: these lines would be called in another class, such as `TestDog`, not in `Dog`)*

```
System.out.println(doggy1.age); //should print 9
System.out.println(doggy2.age); //should print 3
System.out.println(doggy1.name); //error
System.out.println(doggy2.hasFleas); //error
```

The last two statements will result in an error because the `name` and `hasFleas` fields were defined as `private`, which means that we simply can't ask for that information. Since the `age` field was `public`, we can ask for it by saying the name of the object, plus a period, plus the field we are asking for. This format is known as "dot notation", and allows you to access the fields and methods of an object.

But what is the point of having fields if they can't be accessed by other classes? You might want to use that information in some way, without it being publically available. Using information in a specific way is known as a *method* in Java. (You might also call these structures 'functions' in other languages)

Methods

This is where the complicated (and interesting) parts of programming come in. When you are defining a class, often you will want members of a class to *do* something. Things that a class can *do* are its *methods*. Let's extend our `Dog` example:

```
public class Dog {
      private String name;
      public int age;
      private boolean hasFleas = false;
      //constructors not shown to save space

      //returns the value stored in the hasFleas field
      public boolean getFleas(){
            return hasFleas;
      }

      //changes the value stored in the age field
      public void setAge(int newAge){
            age = newAge;
      }

      //prints a statement including the value stored in the name field
      public void speak(){
            System.out.println("Woof woof, my name is " + name + "!");
      }
      //other information defining the dog class will go here
}
```

//It is common to declare 'get' and 'set' methods for private fields

We have defined three new methods that objects of the `Dog` class can execute. Most of your methods will be `public`, because you want to be able to use them. Notice that after the declaration of `public` or `private`, you also declare what type of data this method will `return`. If a method `return`s a value, the result of calling the method is equal to the returned value. The `getFleas()` method returns a `boolean` value, so `boolean` is declared before the method name. The `setAge()` method does not return any data, it just stores the input parameter, `newAge`, in the `age` field. Methods that don't return any information are declared as `void` methods. Notice that methods do not have to take in parameters.

With these methods declared, we could do the following:

```
Dog doggy3 = new Dog("Snoopy", 5);
Dog doggy4 = new Dog("Snoop Dogg", 44);

System.out.println(doggy3.getFleas()); //should print true
doggy3.setAge(77); //nothing printed or returned
System.out.println(doggy3.age); //should print 77

doggy4.speak(); //should print "Woof woof, my name is Snoop Dogg!"
```

Instantiation

You might be wondering where exactly we might be typing lines such as the `doggy3` and `doggy4` examples above. Since this is Java, and Java is an object-oriented language, we actually would need to write these statements in another whole class. Often, such a class can be used to create objects of other classes or test the definition of a class. We might call this class `TestDog`:

```
public class TestDog{
      public static void main(String[] args){    //ignore this for now...
            Dog doggy5 = new Dog("Astro", 6);
            Dog doggy6 = new Dog();
            System.out.println(doggy6.getFleas()); //should print false
            doggy5.speak();//should print "Woof woof, my name is Astro!"
      }
}
```

If we were to include both the definition for the Dog class and this `TestDog` class in the same folder in our Eclipse IDE, the program would behave as expected. Basically, we are creating a class that *instantiates* (creates examples of) another known class. This only works if there is a definition for the `Dog` class that the `TestDog` class can access, but in this way we can test and use the `Dog` class from this shorter test class.

The **`static`** method named **`main`** is special. You should only have one of these in any project that you are trying to run, and these reserved words indicate that this method is the main method in your program that will organize and execute the data and methods from your other classes. `static` indicates that this method can run even if we create no instances of the `TestDog` class - unlike the `speak()` method, we don't need to create a `TestDog` object to run the main method.

static methods vs. instance methods

So far, we have mostly been looking at examples of *instance methods*, which are methods that can be run by instances of a class. For example, the `speak()` method is an instance method for the Dog class, which means that only instances of the Dog class can call this method. In other words, you need to have created a `Dog` in order for this method to be called, such as the call above to `doggy4.speak()`. If we tried to call speak() on its own, the program wouldn't know what to do - who should speak? This doesn't mean, however, that you need to create an instance of a class to run any method. Instead, you can create a ***static*** method that does not need an instance to call it.

Think of the classes that we have defined so far in our study of Java. Some of them, such as `Dog`, were designed to be instantiated. Others, such as `TestDog`, were not - we never tried to create an instance of the `TestDog` class (and we didn't make a constructor, so we couldn't even if we wanted to.)

```
public class TestDog2{
      dog1 = new Dog("Scrappy", 4);
      dog2 = new Dog("Scooby", 8);

      public static boolean isOldDog(Dog d){
            return d.age >= 7; //note: age is a public field
      }

      if(isOldDog(dog1)){  System.out.println(dog1.speak());  }
}
//notice that isOldDog() is a static method, so it can be called even though
//no specific object is calling it! By comparison, the speak() method above
//can only be called by an instance of the Dog class, such as dog1 above!
```

One prominent example of a static method is the main method, which we use to execute our programs. Remember in Eclipse when we go to test a class that we have created, we often create a test class, which always includes a method such as:

```
public static void main(String[] args){
      //lines of code to execute
}
```

This method is `static` because we aren't creating an instance of the test class in order to run it - we are running it from the class itself.

Unit 2 Cycle 1: Conditional Logic & Control Flow

In Unit 1, we learned about the basic structure and rules of Java statements and classes. While this is an important foundation for constructing solutions to complex problems, we are still limited on one important way: so far, all of our programs would execute the exact same lines of code in the same way, every time time they are executed. This isn't ideal, as we want our computer programs to be able to behave differently based on many factors. Consider a basic text-based game, where the program asks the user questions, and reacts to their answers. To accomplish something like this functionality, we need to be able to train our programs to react to specific conditions, which is the basis of Unit 2.

Boolean Expressions

A boolean expression results in a boolean value (either true or false.) The most common boolean expressions, listed below, can be used to compare data.

>	greater than	*boolean foo = (10 > 9); foo is true*
<	less than	*foo = (10 < 10); foo is _____*
>=	greater than or equal to	*foo = (6 >= (3*2)); foo is _____*
<=	less than or equal to	*foo = (2 <= (7%3)); foo is _____*
==	equal to	*foo = (2 == (5/2)); foo is _____*
!=	not equal to	*foo = (true != false); foo is _____*

Note that when testing equality of objects (as opposed to primitive data types), use the equals method. E.g. `object1.equals(object2)`. This will test whether the objects are equal in terms of their attributes, whereas using **==** will only test whether the two variables are actually references to the same object.

Example:

```
int x = 17;
int y = 17;
System.out.println(x == y); //prints true, primitives can be compared this way

String word1 = new String("hello");
String word2 = new String("hello");
System.out.println(word1.equals(word2)); //prints true
System.out.println(word1 == word2); //prints false!
// == between two objects checks to see if they refer to the same piece of data
String word3 = word1;
System.out.println(word1 == word3); //prints true!
```

In the example above, `word1` and `word2` store their values in different locations on the hardware. When we ask if they are equal, we are asking if they point at the same location. By setting `word3 = word1`, we are saying that `word3` points at the same location as `word1`, so they *are* equal.

Reflection: Imagine that you had two Book objects, how would you check to see if they were equal?

Logical Operators

Operator	Description	Example	Evaluates as true if...
&&	logical AND	`A && B`	*both values are `true`*
||	logical OR	`A || B`	*short-circuit, `true` as soon as one value is `true` ***** *
!	logical NOT (logical negation)	`!A`	*`A` is false*
^	boolean logical exclusive OR	`A ^ B`	*Only one of values is `true`*
|	boolean logical inclusive OR	`A | B`	*At least one is `true`, not short-circuit*

(Note that the data types of `A` and `B` should be `boolean`s)

***** Short-circuit means that the compiler will not even look at the second term if the first term is true. This means that your program could contain a fatal error (such as trying to divide by 0), or have an incompatible data type, and you might miss it!

Example:

```
boolean apple = true;
int orange = 77;
boolean compareFruit = apple || orange;    //no error
boolean moreCompare == apple | orange;          //causes error
```

You can combine many logical operators to create very complex conditions. These operators will follow the normal order of operations from left to right, though you can group operators using parenthesis:

```
boolean tricky = ( 10 < 12 || ((5 % 3 >= 3 % 2) && !(true && false)));
```

Here, the two boolean comparators in ( ) are evaluated first, then we can apply the logical operators. This results in: `( false || (true && true) );` which is equivalent to `false || true` which is `true`!

Now that we can make complex comparisons and logical rules, we can apply them to influence the flow of our programs.

DeMorgan's Law

Consider the following complex, compound boolean expression:

```
!(true || (false && false));
```

While you could evaluate and simplify the expressions, then negate the results of the parentheses, you could also use a mathematical logic trick known as **DeMorgan's Law**. This law states that if you have a ! in front of a boolean expression with a logical operator, you can 'switch' the logical operator and remove the !, while maintaining the same logic:

`!(true || (false && false));` is the same as `(true && (false || false));`

!(true || (false)) → *!(true)* → *false* *(true && (false))* → *false*

It can be helpful to think of **DeMorgan's Law** as being similar to distributing a negative sign when you are simplifying inequalities in algebra:

-2x -10 > 2 → -2x > 12 → x < -6

When solving for an inequality, dividing by a negative switches the inequality. Similarly, when distributing a *not* sign (!) in a boolean expression, you switch `||` to `&&` and vice versa.

DeMorgan's Law has appeared on the AP Computer Science A test every year for the past 10 years, mostly in multiple choice questions. Not only is it important in terms of testing, but it is also often a helpful way to simplify complicated boolean expressions that are not immediately clear.

Control Flow: If

Often, you don't want every statement of your program to execute every time you run your program. You might want a certain statement to execute, but only if a specific condition is met. Consider the example:

```
int num = 10;
int randNum = (int)(Math.random()*20) + 1; //random int from 1 to 20;
if (randNum > num){
      System.out.println("The random number was greater than 10!");
}
```

In the statements above, the keyword `if` creates *conditional logic*, which controls the flow of the program. If the condition, which is in parentheses, evaluates to be `true`, then the statement(s) between the curly braces will execute. In this example, it is unclear weather this statement will be true, because our condition is comparing a random integer between 1 and 20 to the integer 10. Approximately half of the time the print statement should occur, but in cases where the random number is < 10, nothing will happen!

Control Flow: If/Else

What if you have two possible paths to follow: one if a condition is met, and another if it is not met. You can do the following:

```
int num = 10;
int randNum = (int)(Math.random()*20) + 1; //random int from 1 to 20;
if (randNum > num){
      System.out.println("The random number was greater than 10!");
} else {
      System.out.println("The random number was not less than 10!");
}
```

Notice that there is no condition following the **`else`** keyword, because it accounts for all other conditions other than a true condition. But what if you had multiple conditions to check, with unique responses for each condition?

Control Flow: If/Else If/Else

In the example below, we add a second condition with its own response:

```
int num = 10;
int randNum = (int)(Math.random()*20) + 1; //random int from 1 to 20;
if (randNum > num){
      System.out.println("The random number was greater than 10!");
} else if(randNum < num){
      System.out.println("The random number was less than 10!");
} else {
      System.out.println("The random number is equal to 10!");
}
 // here the else if is followed by a second condition, if neither are true then the else will execute
// notice that in a single if/else if/else structure, only one of the blocks will execute
```

Adding It All Together

As we saw earlier, you can combine boolean expressions into a single boolean value by using logical operators. You can use these complex boolean statements in conditional logic:

```
int num1 = 10, num2 = 7
int randNum = (int)(Math.random()* 5) + 1;

if(randNum < num1 && randNum < num2){
      //do something
} else {
      //do something else
}
```

Unit 2 Cycle 2: Nested If/Else, Switch

In the last cycle we studied the structure and functionality of conditional logic. This allowed us to make our programs react based on specific conditions, and that is a good thing. While the structures were relatively straight-forward, you might be surprised at how nuanced the differences can be between similar uses of conditional logic. Consider the two blocks below:

`int x = 8;`

```
if( x > 6 ){
  //do A
}
else if( x%2 == 0 ){
  //do B
}
else if( x + 2 == 10 ){
  //do C
}
else{
  //do D
}
```

`int x = 8;`

```
if( x > 6 ){
  //do A
}
if( x%2 == 0 ){
  //do B
}
if( x + 2 == 10 ){
  //do C
}
else{
  //do D
}
```

If we ran these two blocks, what would be the result? On the left hand side, notice that only one of the internal *do* blocks will be executed, because in an if/else if/else structure as soon as one condition is true, the remainder of the logical stack is ignored. In this case, the program would *do A*, and stop. On the right hand side, each `if` will be evaluated, so *do A*, *do B*, and *do C* will all be executed. The final, *do D*, will not, because once the condition for C was met, the `else` statement will not execute.

Similarly, the keyword `return` in a method definition can cause some conditional logic to be skipped:

```
public String foo(int x){
      if( x > 6 ){
        return "A";
      }
      if( x%2 == 0 ){
        return "B";
      }
      if( x + 2 == 10 ){
        return "C";
      }
      else{
        return "D";
      }
}
```

In this case, a call to `foo(8)` would result in a returned value of "A". Even though each `if` statement would be evaluated, all methods end as soon as they `return` a value! Similarly, a call to `foo(4)` would return "B", because the first condition would be false and the second would be true, resulting in the end of the method.

So simple `if/else if/else` structures can be complicated, but we are only scratching the surface...

Nested if/else

```
1 int num = 24;
2 if(num > 20){
3     if(num % 7 == 0){
4           if(num % 3 == 0){
5                 System.out.println("Yay!");
6           }
7     }
8     else if(num % 3 == 0){
9           System.out.println("Yikes!");
10    }
11    else{
12          System.out.println("Surprise!");
13    }
14 }
15 else{
16    System.out.println("Wowza!");
17 }
```

What would be printed to the console if this line of code was executed? It would print the string "Yikes!". Let's trace the logic: first, the condition (num > 20) is checked, resulting in true, which executes the block following line 2. Then the condition on line 3 is checked. Since 7 does not divide evenly into 24, this condition fails and lines 3 through 7 do not execute. Then the condition on line 8 is checked, which is true, so "Yikes!" is printed. Notice that neither else statement will execute, since an if condition was met for each.

Is there a less complicated way to do something like this? Possibly:

```
if( ((num > 20) && (num % 7 == 0)) && (num % 3 == 0)){
      System.out.println("Yay!");
}
else if((num > 20) && (num % 3 == 0)){
      System.out.println("Yikes!");
}
else if(!(num > 20)){
      System.out.println("Wowza!");
}
else{
      System.out.println("Surprise!");
}
```

This is logically equivalent to the nested if statements above. Notice that an if statement nested inside another if statement is similar to using the && logical operator. You may or may not prefer this type of structure, and the AP test will not require you to use one over another in the free response section. For MC questions, any structure might be present, so it is good to be able to understand both.

Variable Scope

What if you saw the following:

```
int y = 6;
if(y > 5){
      int x = 2 * y;
      y += x;
}
System.out.println(y);
System.out.println(x);
```

Many people would expect these statements to print 17 and 12, but it will actually result in an error. Why? Well, consider the scope of the variables in these statements. `y` has been defined, and as a result it can be accessed by the blocks within the if statement. But what about `x`? `x` only gets a definition inside of the if block, which means that `x` only exists in the short space between the `{ }` following the keyword if. When we try to print `x` later, there is an error, because there is no such variable that the print statement can access!

This concept, that variables defined within specific levels of an `if/else` block also applies to nested `if/else` statements, as well as methods and class definitions.

Switch

Suppose you were in a situation where you had 6 conditions to check, each with its own resulting statements to execute. You already know how you might accomplish this using if/else structures, or nested if/else structures. But it can be pretty tedious to write the conditions over and over again - isn't there an easier way? This is where the switch statement comes in:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

In this structure, the keyword `switch` is used. Following this keyword, you can put an expression, which is evaluated against each `case`. If the value listed after the case matches the expression, the statement(s) for that case are executed. Notice the keyword break included above as well - if this is executed, the switch statement is immediately closed and no further cases are checked. If there is no break statement, multiple cases could match.

Switch Example:

```
int num = (int)(Math.random()*10 + 1); //a random int from 1 to 10
switch(num){
      case 1 :
            System.out.println("blah");
      case 2 :
            System.out.println("bleh");
      case 3 :
      case 4 :
      case 5 :
            System.out.println("blurg");
      case 6 :
            System.out.println("bling");
      default :
            System.out.println("I ran out of b sounds");
}
```

Here, we do not need to write multiple conditions checking the value of `num` (such as `if(num == 1)`, `if(num == 2)`, etc), instead these comparisons are implied through each `case`. Notice cases 3, 4 and 5 - this is equivalent to the condition `if( num == 3 || num == 4 || num == 5 )`. In addition, notice that the `default` case means that any number that does not match 1 through 6 will result in the `default` statement.

The if/else structure below recreates the logic above:

```
if(num == 1){
      System.out.println("blah");
}
else if(num == 2){
      System.out.println("bleh");
}
else if(num == 3){
      System.out.println("blurg");
}
else if(num == 4 || num == 5 || num == 6){
      System.out.println("bling");
}
else{
      System.out.println("I ran out of b sounds");
}
```

There is no operational difference between these approaches, but some people prefer one more than the other. On the AP test, you will not be required to write code in any specific structure, but you should be prepared to see both `if/else` structures and `switch` statements on the test.

Unit 2 Cycle 3: Loops

In our previous lab, `TechSupport`, we created a conversation between our program and a user. This was nice, because it allowed us to collect information and react conditionally to the user's input. But there were some shortcomings to this program - what if the user had three problems that needed to be solved, or what if they accidentally typed in an incorrect response and wanted to start over? It would be terribly annoying and inefficient to write out conditional if/else if/else structures for each repeated interaction, so we will need to find a better way to reuse our code. This is where loops, the topic of this cycle, come in.

Loops

Loops are one of the most fundamental components in the programs that you will write. They are important because they allow us to efficiently make use of a computer's primary strength: its ability to follow many instructions *very* quickly and accurately. Instead of explicitly telling the program to execute a block of code many times in a row, we can simplify our programs by wrapping that block with a loop:

Without a loop	*With a loop*

`System.out.print("hello");`
`System.out.print("hello");`
`System.out.print("hello");`
`System.out.print("hello");`
`System.out.print("hello");`
	`for(int i=0; i<5; i++){`
	`  System.out.print("hello");`
	`}`

Both of the examples above would result in the string "hellohellohellohellohello" being printed to the console, but notice how much more concise the example with a loop is. Moreover, what if you wanted to change what the program printed to "goodbye"? On the left, you would need to replace all five instances of the string, whereas on the right you would only need to replace it once. But most importantly, what if you wanted to print "goodbye" 1000 times? It would be ridiculous to copy and paste 995 more print statements following the pattern on the left, when you could accomplish the same work just by changing the 5 in the loop to the right to 1000!

There are multiple types of loops that you can write in most programming languages, but for the AP Computer Science exam you need to be very familiar with three structures: **for loops**, **while loops**, and **for-each loops**. We will focus on the first two for this unit, as the last type of loop is not useful until we have studied arrays in Unit 3.

Reflection: In your own words, why might you want to use a loop in a program? Describe an example of a program that might benefit from using a loop.

For Loops

A 'for loop' is one of the most common structures that allows you to *iterate* over a block of code many times. When you *iterate* over a block, you execute it, and a loop allows you to execute many iterations with very few lines of code. Each *iteration* is a single execution of the code inside of the loop.

Consider the example from the introduction above:

```
for(int i=0; i<5; i++){
   System.out.print("hello");
}
```

The block of code that is being iterated is the print statement between the `{` curly braces `}`, and the rules for the number of iterations are defined in the `( )` after the keyword `for`. A `for` loop takes in three pieces of information: a variable with a starting value, a continuing condition, and a statement that changes the variable.

variable and starting value	`int i = 0;`	*creates an int variable named* `i` *and sets it to 0*
continuing condition	`i < 5;`	*the loop will repeat as long as* `i` *is less than 5*
changing statement	`i++;`	`i` *will increase by 1 each time the loop has executed*

Notice that in this example, if the changing statement had been `i--`, we would be in trouble. The trouble would be that our loop will continue as long as `i` is less than 5, but if `i` is going to *decrement* (get smaller), the continuing condition will never be false and as a result we will have created an infinite loop, which will crash our program!

Loop Example	**Outputs**	**Because...**

`for(int i=1; i<4; i++){`
 	
`System.out.println("goodbye");`
`}`
	goodbye
	goodbye
	goodbye
		Note that `i` starts with a value of 1, and the loop stops once `i` is >= 4.

`for(int i=10; i>6; i--){`
`  System.out.println(i);`
`}`
	10
	9
	8
	7
		`i` starts at a value of 10, and the loop continues as long as `i` is greater than 6. `i` gets smaller with each iteration.

`for(int i=0; i<10; i+=2){`
`  System.out.println(i);`
`}`
	0
	2
	4
	6
	8
		`i` starts at a value of 0, and the loop continues as long as `i` is less than 10. `i` increases by 2 with each iteration.

You can, of course, include multiple statements inside of the curly braces `{ }`, and each will be executed in order.

‘break’ in a loop

The loops that we have seen so far will all execute until the condition (the second part of the for loop declaration) is false. But in some instances, we might want to stop the loop early. You can use the keword `break` to accomplish this:

```
public static void breakAtX(int x, int y){
      for(int i = 0; i < y; i++){
            if(i == x){
                  break;
            }
            else{
                  System.out.println(i);
            }
      }
}
```

In the method above, inside of the loop we have added an if/else block. As we iterate through values of `i`, if our `i` value is equal to the `x` parameter, the loop will `break` and stop:

`breakAtX(6,10);`	`breakAtX(2,100);`	`breakAtX(9,5);`
0 1 2 3 4 5 6	0 1	0 1 2 3 4 5

Notice that in these calls to the `breakAtX()` method, the loops only `break` once the iteration reaches the value of `x`. In the second call, even though the loop was set to end at 100, the condition of `i==2` was met, so the loop encountered a `break`. In the last case, the loop never reaches a condition where `i==9`, so the full range of 0 through 5 is printed.

Note - if a method is going to `return` a value, if `return` is reached in a loop, the loop will immediately break and the method will `return` that value and stop!

```
public static int returnFive(int x){
      for(int i = 0; i < x; i++){
            System.out.println(i);
            if(i == 5){  return i; }        //return ends the loop and method
      }
}
```

If we called `returnFive(100)`, the values 0,1,2,3,4,5 would be printed, and 5 would be returned.

While Loops

For loops are useful when you know how many times a block should be executed, but this isn't always the case when you are creating a program. Maybe you need a block of code to execute 5 times in one situation, but 7 times in another. A while loop can be used to solve this problem:

```
int i = 0;
while(i<3){
   System.out.print("hey there!");
   i++;
}
```

This loop would result in "hey there!hey there!hey there!" to be printed to the console. Notice that after the keyword `while`, a condition is set in parenthesis. As long as this condition is true, the block of code inside of the `{ }` will execute. In the example above, the first time through the loop, `i` is less than 3, so the condition is true and the print statement executes. Then, `i` gets incremented. The condition is then checked again - is `i` less than 3? Yep, `i` is 1, so the condition is true and the block will execute again, printing and incrementing. This will continue until `i` is set to 3, in which case `i < 3` evaluates as false and the loop stops.

You might be wondering, 'how is this any different from a *for loop*?', since in both cases we are specifying how many times the code should execute. But consider the following situation: You have $100 in a bank account that grows by 5% every year. You want to know how many years it will take you to have $140. You know that each year you just need to multiply your current balance by 1.05, but you don't know how many times this will need to happen to reach $140. A while loop can do this for you:

```
double money = 100.00;
int years = 0;
while(money < 140.00){
  money = money * 1.05;
  years++;
}
System.out.println("It took " + years + "years!"); // It took 6 years!
```

The chart below tracks the values of each variable through each iteration:

iteration	money	money < 140?	block execution (rounded)	years++
0	100	true	100 * 1.05 = 105	1
1	105	true	105 * 1.05 = 110.25	2
2	110.25	true	110.25 * 1.05 = 115.76	3
3	115.76	true	115.76 * 1.05 = 121.55	4
4	121.55	true	121.55 * 1.05 = 127.63	5
5	127.63	true	127.63 * 1.05 = 134.01	6
6	134.01	true	134.01 * 1.05 = **140.71**	stop!

Unit 2 Cycle 4: Break & Nested Loops

So far this unit, we have learned how to use conditional logic and loops to control the flow of our programs. This has allowed us to do create more complicated structures to solve more complicated problems. Remember when we learned how to created nested conditions by inserting if statements inside of each other? That allowed us to create a tree of conditions that could branch in many complex directions, which is often useful. It turns out that we can do a similar trick with loops by inserting loops into other loops, resulting repeated repetitions that can help us to do even more complicated work! But before we get to nested loops, it might be useful if we know how to put on the breaks under some conditions, just in case we want to end a loop early.

Break

In some situations, you might want to stop a loop before it has reached its stopping condition. For example, you might be writing a program that will ask a user 10 questions, but if at any time their answer is “exit”, you want the loop to end. In this case, you can use the keyword `break` to immediately end the loop and move on to the next line of code.

```
Scanner scanner = new Scanner(System.in);
for(int i = 0; i < 10; i++){
  System.out.println("Tell me something:");
  String answer = scanner.nextLine();
  if(answer == “exit”){
    break;
  }
}
```

In this example, the program will ask the user for input 10 times, unless the user types “exit”, in which case the loop will end. Notice that this break statement only ends its nearest *parent* loop (in this case, the for loop that contains it). If another loop were to follow, it would still execute, because the scope of the break statement only applies to its closest containing loop.

Nested Loops

Just as we saw that a programmer can nest conditional logic (an `if` statement inside of an `if` statement, for example), you can also nest loops inside of each other:

```
for(int i = 0; i < 3; i++){
      for(int k = 0; k < 5; k++){
            System.out.print("* ");
      }
      System.out.println(); //puts the next iteration in a new line
}
```

In this example, the outer loop will execute 3 times. Each time it executes, the full internal loop will execute five times. The results of this nested loop would look like:

```
* * * * *       //here i = 0, k iterates from 0 to 4
* * * * *       //here i = 1, k iterates from 0 to 4
* * * * *       //here i = 2, k iterates from 0 to 4
```

Notice that the second print statement is outside of the internal loop, because we only wanted to go to a new line after the internal loop has completed. For patterns like this, you might consider the iterations of an inner loop to represent the *columns* of the shape, and the iterations of the outer loop to represent the *rows* of the shape.

```
for(int x = 0; x > 3; x++){
      for(int k = x; k < 5; k++){
            System.out.print(k);
            System.out.println(); //puts the next iteration in a new line
      }
}
```

Notice in this example, the internal loop is different in each iteration, since `k` gets set equal to the current value of `x`. This would result in:

01234
1234
234

Nested Loops & Break

```
for(int x = 0; x < 3; x++){
      while(1 < 2){
            int num = (int)(Math.random()*100 + 1);
            if(num == 77){
                  System.out.println("broke out of it!");
                  break;
            }
      }
}
```

Notice in this example you might think the `while` loop would result in an infinite loop, since its condition is always true. Nevertheless, as soon as the `num` gets randomly assigned the value 77, the internal loop will break. Then the external loop will start again, and repeat a total of three times.

Shape Examples

One common way to practice with nested loops is to use them to print specific shapes to the console. Consider the following:

```
int x = 5;
while(x > 0){
      for(int k = 0; k < x; k++){
            System.out.print("* ");
            System.out.println(); //puts the next iteration in a new line
      }
      x--;
}
```

Here, we will repeat the outer loop 5 times, and each inner loop will repeat from 0 to the current value of x, resulting in the following shape:

```
* * * * *       // outer loop: 1st iteration, inner runs while k < 5
* * * *         // outer loop: 2nd iteration, inner runs while k < 4
* * *           // outer loop: 3rd iteration, inner runs while k < 3
* *             // outer loop: 4th iteration, inner runs while k < 2
*               // outer loop: 5th iteration, inner runs while k < 1
```

```
for(int r = 0; r < 5; r++){
      for(int s = 5-r; s > 0; s--){  //prints blank spaces
            System.out.print(" ");
      }
      for(int t = r; t > 0; t--){
            System.out.print("* ");  //prints stars
      }
      System.out.println();         //starts new line
}
```

Output: *In the space below, trace the loops - at each row, what are the values of `r`, `s` and `t`?*

```
    *
   * *
  * * *
 * * * *
```

Unit 3 Cycle 1: String Methods

In Unit 2, you learned about ways to control the flow of the statements in your programs. Whether you were creating if/else trees, switch statements, loops, or even nesting many of these things together at once, you used these tools to make specific events occur in a specific (or sometimes even random) order. But what 'events' have we been executing in our programs? Well, mostly things like arithmetic operations, generating random numbers, and printing statements to the console. While these are often very important parts of programming, they are mainly number-based (with the exception of print statements, which can print non-numbers). Frequently, you will need to write programs that deal with the type of data that humans are most comfortable working with: words. In this cycle, you will study the `String` data type, with a specific focus on the methods that are provided in this class, both in the AP Java subset, and in the broader Java world.

The `String` Class and 'Reference Types'

If you look back to your notes from Unit 1, you might remember a discussion about the difference between *primitive* data types, and instances of a *class*. We said that data types such as `boolean`, `int`, `double`, and `char` were all *primitive*, which meant that they did not exist as instances of a class with discrete methods. For example, if we wanted to cube an integer we might be tempted to try something like:

```
int x = 5;
int y = x
int z = x.cubed(); //error!
x++;
System.out.println(y);     //prints 5, not 6!
```

If you tried to run this program in Eclipse, you will get a warning that integers do not have a method called `cubed()`. Further, notice that when you assign values to primitive data types, they are stored as separate values (if they were saved to the same value, changing `x` would also change `y`!) Even worse, since `int` isn't a class, we can't even try to write such a method.

The `String` data type, on the other hand, is not primitive - like any other class, it is considered a 'Reference Type'. String is a special class built into Java. Like other classes that we have created, it has a constructor:

```
String myWord = "banana";
String yourWord = new String("apple");
```

So far you have seen examples of the first 'constructor' for `String` objects - you simply say the data type, give the variable a name, and set it equal to something inside quotation marks. You can't do this for other classes, but **`String` is special** because it is so often used, so you can create a new `String` much like you can store primitive data types like `int`s and `boolean`s.

The second statement uses a constructor shape that you should be more familiar with from other classes. Here we are saying that `yourWord` will be a new instance of the `String` class. You might guess that the text that goes into the parenthesis will be the text that is stored in this object.

`Strings` are 'immutable'

It is important to remember that **the value that is stored in a `String` cannot be simply changed.** Consider the statements below:

```
1 String word = "yellow";
2 System.out.println(word + " submarine"); //prints "yellow submarine"
3 System.out.println(word); //what would this print?
```

*The second print statement would only print "yellow". Sometimes, students learning Java might think that the `+` operation in `word + " submarine"` combined values into one `String`, but this is incorrect! When we say a `String` is immutable, we mean that the value of a `String` object can only change if you explicitly **replace** the value that it stores:*

```
1 String word2 = "green";
2 word2 + " apple";
3 word2 = word2 + " goblin";   //could have been written word2 += " goblin"
4 System.out.println(word2);   //prints "green goblin"
```

In the example above, note that on line 2 when you add `word2` to `" apple"`, the value of `word2` does not actually change! This statement evaluates to `"green apple"`, but nothing is being done with that information. It is only when we explicitly set the value of word2 to a new value using the `=` sign that the value for this `String` has changed!

Don't forget that when you use the `=` sign, you are setting one location in the computer's memory equal to the value stored *at another location*. For example:

```
1 String word3 = "orange";
2 String word4 = word3;
3 System.out.println(word4);    //prints "orange"
4
5 word3 = "agent " + word3;
6 System.out.println(word4);    //prints "agent orange"
```

*Why does line 6 above print "agent orange" even though we did not explicitly change the value of `word4`? Because when we instantiated `word4` we set it equal to the memory location for `word3`, which means that `word4` **references** `word3`, and if the value of `word3` changes, so does the value at `word4`.*

Even more confusing, when we did this with integers above, they didn't change. Why? Because instances of a primitive data type store values at different locations when they are created, while reference objects can be instantiated to point at a single shared location.

This is a key difference between <u>primitive</u> types such as `int`, `char`, and `boolean`, compared to <u>reference</u> types, such as `String` and other classes that we define!

String Methods

The `String` class comes packed with methods that you can use to do very useful (and sometimes complicated) things with `String` objects. We won't cover them all, but you can find a more exhaustive list of the provided methods, their syntax, and how they work, in the Java Documentation or at http://www.tutorialspoint.com/java/java_strings.htm. The methods below are the most commonly used on the AP test, and you will need to be able to use all of them!

method	description	example
`int length(){...}`	*returns the number of characters, including spaces, in the String.*	`String x = "Hey you!";` `x.length();//returns 8`

`int compareTo(String anotherString);`	*compares lexicographically*	`String y = "abc";`
		`x.compareTo(y);`
		`//returns a positive int`

`String concat(String anotherString);`	*returns the result of adding two Strings together*	`String z = x.concat(y);` `//z now is "Hey You!abc"`
`boolean equals(Object anObject);`	*returns true if the **values** are equal*	`x.equals(y);` `//returns false`
`boolean equalsIgnoreCase(String wrd);`	*like equals, but ignores capitalization*	`z.equalsIgnoreCase("AbC");` `//returns true`
`int indexOf(String str);`	*returns the **index position** of the first instance of the input*	`x.indexOf("y"); //returns 2, the index of the first "y". Index counting starts at zero! -1 returned if no instance found.`
`int lastIndexOf(String str);`	*returns the index position of the last instance of the input*	`x.lastIndexOf("y");` `//returns 4, the second "y". -1 returned if no instance found.`
`String replace(char oldChar, char newChar);`	*replace all occurrences of oldChar with newChar*	`x.replace('y','X');` `//returns "Hex xou!"`
`String replaceAll(String oldStr, String newStr);`	*replaces all occurrences of oldStr with newStr*	`x.replaceAll("you","friend");` `//returns "Hey friend!"`

method	description	example

`String substring(int beginIndex);`	*returns the String from beginIndex to the end of the STring*	`x.substring(3); //returns " you!", including the space`
		`//returns error if index is out of bounds`

`String substring(int beginIndex, endIndex);`	*returns the String starting at beginIndex and ending **before** endIndex*	`x.substring(2,5);`
		`//returns "y y"`
		`//returns error if index is out of bounds`

`String toString();`	*the value is returned*	`q = x.toString();` `//now q references the value, not the location!`
`String toLowerCase();`	*makes all capitalized letters lowercase*	`"YELL!".toLowerCase();` `//returns "yell!"`
`String toUpperCase();`	*makes all lowercase letters capitalized*	`"whisper".toUpperCase();` `//returns "WHISPER"`
`String trim();`	*removes any leading or trailing whitespace*	`"  hat  ".trim();` `//returns "hat"`

`boolean contains(String str);`	*returns true if the String contains the input parameter*	`x.contains("yo"); //true`
		`x.contains("YO"); //false`

These probably seem like a lot, but we have only scratched the surface here. There are also some String methods that we will not study until we have completed other units, such as methods that turn strings into arrays or other data structures. For now, you need to become familiar with these methods above!

Interacting With Other Data Types

While you can do basic addition with integers (`5 + 6`) and concatenate Strings (`"Happy " + "Birthday!"`), it is less intuitive to see a statement such as `7 + "Eleven"`. In this case, it is hard to tell what Java will do - will the `7` get converted to the `String "seven"`? Will the `"Eleven"` be converted to the `int 11`? Neither - in this case, Java will turn the integer into a `String` as `"7"`. This means that you will need to be careful when dealing with user input, which is often stored as a String. For example, if you used the Scanner class to ask the user for a number, you might accidentally store their answer as a String. To do math with their answer, you would need to convert the answer into an integer first:

```
Scanner sc = new Scanner(System.in);
System.out.println("How many times should I say boo?");
String a = sc.nextLine();

//Imagine that the user entered "5" as their answer

System.out.println(a + a); //this will print "55", not 10!
int num = Integer.parseInt(a);

//this method is important: it takes a string and turns it into an integer!
```

*While primitive types like `int` have no methods, there are "**wrapper**" classes such as `Integer`, which basically allow you to run specific methods on the data in your program. The lines above take the user's answer and turn it into an `int`, stored in the variable `num`. Now we can call:*

```
for(int i = 0; i < num; i++){ System.out.println("Boo!"); }
```

If we had tried to use `a` instead of `num` as the limit for our loop, we would have an error since `"5"` is not a number!

Using String Methods

Remember that you can call an object's methods by writing *object.method()*.

```
String word = new String("hoolahoop");
String afterHoop = word.substring(word.lastIndexOf("hoo"));
```

What will be stored in `afterHoop`? First, we run the `word.lastIndexOf("hoo")` method, which returns 5 (start counting at zero for index positions!). Then, we run `word.substring(5)`, which is "hoop".

You can also chain multiple methods in a row - this isn't specific to String methods, but it is common when using Strings:

```
word.replace("o","x").toUpperCase().concat(" party");
//  "hxxlahxxp"       "HXXLAHXXP"         "HXXLAHXXP party"
```

Here, each method gets called in order. The next method is run on the value returned by the method before it!

Common Errors With String Methods

Programmers often make the following errors when working with `String` methods:

1) == versus `.equals()`

Remember that when we are working with reference types, such as instances of `String` objects, using the boolean comparator == behaves differently than it does with primitive data types. Consider the following:

```
int x = 7; y = 7;
boolean areEqual = x == y;  //here areEqual is true

String s1 = "apple";
String s2 = "apple";
String s3 = s2;

areEqual = s1 == s2;  //here areEqual is false
areEqual = s1.equals(s2);  //here areEqual is true
areEqual = s1 == s3;  //here areEqual is true
```

2) Chaining Methods & Modifying Strings

Remember that Strings are immutable, which means that simply calling String methods does not change the value stored in a String - you need to explicitly re-assign a value to the String if you want it to change:

```
String journalEntry = "Today I ate two pizzas!");
journalEntry.replaceAll("pizzas","apples").toUpperCase();
```

Did the value stored at `journalEntry` change? NO! Calling the .replaceAll() method doesn't change the String, it just returns a String based on the original. To change the string, we would need to re-assign it's value:

```
journalEntry = journalEntry.replaceAll("pizzas","oranges").toUpperCase();
```

Now `journalEntry` holds the `String`: "TODAY I ATE TWO ORANGES!"

Note: Assignment operators such as += *do* assign a new value to a data point, so…

```
journalEntry += " Yum!";
```

***would* change the value of the variable:** "TODAY I ATE TWO ORANGES! Yum!"

3) `IndexOutOfBounds` errors:

Index positions are important and useful, but it is easy to cause an error. What if you wanted to write a method that returns the last character of a String that is put into it? You might try to do the following:

```
public static String lastLetter(String inString){  //ignore 'static' for now!
      int wordLength = inString.length();
      return inString.substring(wordLength);
}
```

The intention here would be to find out how long a word is, let's call that length n, and then find the n^{th} letter in the String. But what happens when we call `lastLetter("doggy");`*?*

In the method, wordLength would be set equal to "doggy".length(), which is 5;
Then, it would attempt to return the substring of "doggy".substring(5)
***But** remember that we start counting at 0 for index positions. What is the 5th index position of "doggy"?*

d	*o*	*g*	*g*	*y*
0	*1*	*2*	*3*	*4*

There is no letter at the 5th index of "doggy", even though the String has a length of 5! Remember that you need to subtract 1 from the length if you are planning to use the length in conjunction with the index!

To fix this, we could update the return statement to `return inString.substring(wordLength - 1);`

This also applies to the `.substring(int start, int end)` *method. Note, however, that the end parameter is NOT included in the returned string, so even if this index would be out of bounds, the method can still run:*

```
String lastFew = inString.substring(wordLength-2,wordLength); //"gy"
```

Even though wordLength is out of bounds above, the substring ends BEFORE the second parameter, so it does not cause an error!

Unit 3 Cycle 2: Arrays

So far in studying Java, we have spent a lot of time figuring out ways to store, interpret, and create single pieces of data. For example, we might write a function that takes in a single piece of data, and returns a value based on this data:

```
public int countF(String word){
      word = word.toLowerCase();
      int fCount = 0;
      for(int i = 0; i < word.length(); i++){
            if(word.substring(i, i+1).equals("f")){
                  fCount++;
            }
      }
      return fCount;
}
```

If we called countF("Falafel"), the integer 2 would be returned. This is nice, but what if we had a list of 5 words words and we wanted to call this method on them? Well, we would have to call the method explicitly each time:

```
countF("Falafel");
countF("FYI");
countF("Frisbee");
countF("Baffle");
countF("Shut the front door!");
```

At a certain point, it gets repetitive to continue calling the method explicitly. That's not great, because we try to keep our code "DRY" ('Don't Repeat Yourself'). Imagine you had a list of 100 words, do you really want to write out the call to `countF()` each time? Even worse, if you stored each of those words in a variable, you would need to create 100 named `String` variables to keep track of those values in your program. Luckily, Java has a data structure that allows you to store, access, and modify groups of data objects, known as an "Array".

The `Array` Data Type

An array is a specific collection of data reference points. In Java, when you create an array, you specify how many pieces of data the array will hold (this is known as the array's *length*). In addition, when you create a new array you must specify the *type* of data that the array will contain.

```
int[] myNums = new int[5]; //here we create an array of 5 integers
String[] someWords = new String[3]; //here we create an array of 3 Strings
double[] thoseDoubles = {1.23, 4.56, 7.89, 10.0}; //an array of 4 doubles
```

In each of the examples above, we declare the data type that the array will hold, followed by `[]`. Notice that you can *instantiate* a new array using the keyword `new`. Above, `myNums` and `someWords` are both currently empty arrays, since we created them but stored no data in them yet; think of them as empty boxes that can hold a specific number of items.

The array `thoseDoubles`, on the other hand, does hold values. If we were to call `System.out.println(thoseDoubles)`, the following would be printed to the console:

```
[1.23, 4.56, 7.89, 10.0]
```

Notice that in this array, all of the values that are stored are of the data type `double`. Unfortunately, you can't mix in other data types (if you tried to put 10 in instead of 10.0, you would get an error!) In addition, now that we have instantiated this array, we can't change the size of it - unless we completely re-define the variable, `thoseDoubles` will represent exactly four `double` values.

Accessing Array Values

Just like with Strings, the values that are stored in arrays are ordered using *index positions*. If you wanted to access the first item in the thoseDoubles array, you could write:

```
thoseDoubles[0]; //this statement equals 1.23, the first value in the array
                 //it means 'find the value at index 0 of the array'
```

Remember that when using index positions, you start counting at zero! What would happen if you called:

```
thoseDoubles[thoseDoubles.length];
```

There are two things to track here: first, look at the inner statement `thoseDoubles.length`. You might have expected to see `()` at the end of length, because it seems like we are calling a method, but that isn't the case. Array objects store the length of the array as a public field, just like you can include a field in the classes that you define. As a result, you can access a public field by saying `Object.field`, or in this case, `Array.length`.

Second, what is the length of this array? 4 (there are four doubles in the array). So what happens when we call `thoseDoubles[4]`? We get an IndexOutOfBoundsError, because index position 4 is the ***5th*** item, and there is no 5th item in this array! As a result, to access the last item in an array, you need to access the index at position (`Array.length - 1`):

```
thoseDoubles[thoseDoubles.length - 1] ;  //this will return 10.0
```

What if we tried to access the third index position of one of the other arrays?

```
System.out.println(myNums[2]); //remember that index 2 is the 3rd item
```

What is actually sitting in each position if we created empty arrays using the `new` keyword? Java has a default value for each data type: `int` → 0, `double` → 0.0, `boolean` → false, `String` → `null`. As a result, the print statement above would print 0, since `myNums` is an array of integers (`int[]`).

Changing Array Values

Because the empty arrays actually have some values, when we go to 'add' values to an array we are actually just changing the values at specific positions:

```
myNums[0] = 14;
myNums[1] = 27;
myNums[3] = myNums[0] + myNums[1];
```

So what is in the `myNums` array now? *[14, 27, 41, 0, 0]*

Notice that each of the value assignments above match the data type for the array. If we tried to say `myNums[4] = 3.7;` we would get an error, because `myNums` can only hold integers and 3.7 is a `double`.

Arrays & Loops (`for` and `for-each`)

So let's go back to the example from the start of the reading. If we have an array of 5 strings, and we want to count how many "f"s are in each string, we could do the following:

```
String[] words = {"Falafel","FYI","Frisbee","Baffle","Shut the front door!"};

for(int i = 0; i < 5; i++){ //fancier: int i = 0; i < words.length; i++
      System.out.print(countF(words[i]) + " ");
}
//this would print 2 1 1 2 1
```

This for loop will execute 5 times (for `i` values 0, 1, 2, 3, 4) and each time through the loop we are running the `countF()` method on a String in the array (`words[0]`, then `words[1]`, then `words[2]`, etc…). Also notice the comment next to the constraints for the for loop - you can use the length of an array as a part of the condition to make sure that your loop goes over every value in the array. This is often better than hard-coding a value such as 5, because you might later add values to your array!

There is a special kind of a loop that is useful for arrays, known as the `for-each` loop:

```
for(String s : words){
      System.out.print(countF(s) + " ");
}
```

This loop will accomplish the same output as the regular for loop above. The statement can be read as *'for each String, known as `s`, inside of the array words'*. Notice that this gets rid of the counting mechanism of a regular for loop - instead, this loop creates a variable called `s` that will represent the members of the array. The loop will execute the statement once for each item in the array, each time replacing `s` with each member of the array:

```
String[] words = {"Falafel","FYI","Frisbee","Baffle","Shut the front door!"};
```

iteration	`s` is...	statement called...
1st	1st item of words, "Falafel"	`System.out.print(countF("Falafel") + " ")`
2nd	2nd item of words, "FYI"	`System.out.print(countF("FYI") + " ")`
3rd	3rd item of words, "Frisbee"	`System.out.print(countF("Frisbee") + " ")`
4th	4th item of words, "Baffle"	`System.out.print(countF("Baffle") + " ")`
5th	5th item of words, "Shut the front door!"	`System.out.print(countF("Shut the front door!") + " ")`

Just as you could use a loop to access the different values in an array, you could use a loop to *change* the values in an array:

```
int[] otherNums = new int[100];
for(int i = 0; i < otherNums.length; i++){
      otherNums[i] = (i*3)/4;
}
```

What would be the first five values stored in `otherNums`?

```
[(0*3)/4, (1*3)/4, (2*3)/4, (3*3)/4, (4*3)/4, (5*3)/4... ]
  0/4=0    3/4=0    6/4=1    9/4=2    12/4=3   15/4=3

[0, 0, 1, 2, 3...] //remember integer division rounds down!
```

--Warning--
You might think, then, that you could do the following:

```
String[] animals = {"dog", "cat", "frog"};
for(String animal : animals){
      animal += "**";
}
```

While this would compile and run without an error, you might (incorrectly) expect the animals array to contain "dog**", "cat**", "frog**". Note that inside of this for-each loop, we are not explicitly re-assigning a value to a specific position in the array. When we say `animal`, we are just grabbing a reference to the Strings stored in the array, not saving the values to the array itself. As a result, the array has not changed by the end of these statements!

On the AP test, you will frequently need to write loops that will do some work on every item in an array. It is very useful to be able to write a few statements that can process information for hundreds or thousands of *different* items!

Arrays & Methods

Remember that when we define a method we use the following structure:

```
public typeToReturn methodName(typeOfParameter parameterName){
      //do something to return the correct type
}
```

Since arrays are a data type, you can use an array as the type that will be returned, and/or the type of an input parameter:

```
public int[] squareItems(int[] arr){
      int[] final = int[arr.length];
      for(int i = 0; i < arr.length; i++){
            final[i] = arr[i]*arr[i];
      }
      return final;
}
```

This method takes in an array of integers as a parameter, and returns an array of integers. Consider the following statements:

```
int[] nums2 = {1, 2, 3, 4, 5, 6};
int[] nums2Squared = squareItems(nums2);
System.out.println(nums2Squared); //prints [1, 4, 9, 25, 36]
```

Notice that the `squareItems()` method could take in an array of any size, and it will return an array of that same size.

Arrays of Objects

You might be thinking: "*OK, I see why I might want to make an array of simple things like integers or strings, but what if I want to make a collection of a bunch of different types of data?*" For example, maybe you want to store names, ID numbers, and account balances for a bunch of users at your bank. Well, you *could* make three different arrays:

```
String[] userNames = {"David", "Erika", "Isaac", "Ashley"};
int[] userIDs = {2411, 1784, 1893, 9231};
double[] userBalances = {0.50, 210.45, 99.99, 105.00};
```

This would work, but it would become difficult to manage this data. For example, what if the bank adds a new customer? We would need to add data to each of the arrays. It would be nice if there was just a single array that contained this data. Luckily, we can do this using classes!

Consider the class definition below:

```
public class Customer{
      public String name;
      public int userID;
      public double balance;

      public Customer(String n, int i, double b){
            name = n;
            userID = i;
            balance = b;
      }

      public void printUserInfo(){
            System.out.println("Name: " + name +
                               ",ID: " + userID +
                               ", Balance: " + balance);
      } //indentation used for readability

      public void addToBalance(double amnt){
            balance += amnt;
      }
}
```

We can use instances of this class to represent the information stored in those three separate arrays:

```
Customer david = new Customer("David", 2411, 0.50);
Customer erika = new Customer("Erika", 1784, 210.45);
Customer isaac = new Customer("Isaac", 1893, 99.99);
Customer ashley = new Customer("Ashley", 9231, 105.00);
```

Now, instead of having three different arrays to hold this data, we can store them all in one array:

```
Customer[] customers = {david, erika, isaac, ashley};
```

Notice that the data type for this array is `Customer[]`, which means that it will be an array of `Customer` objects. This works because each of the variables in the array is a reference to a `Customer` object. This is even more useful when we consider running methods using this array, such as:

```
for(Customer c : customers){
     c.addToBalance(50.00);     //adds 50.00 to each customer's balance
}

for(Customer c : customers){
     c.printUserInfo();    //calls the printUserInfo() method for each customer:
}                          //Name: David, ID: 2411, Balance: 50.50
                           //Name: Erkia, ID: 1784, Balance: 260.45
                           //**continued for each member of the array**
```

Printing Arrays

At times, it might be useful to print an array to the console to check its values. You might think you could do the following:

```
String[] names = {"The Doctor", "Tardis", "Dalek"};
System.out.println(names);
```

But instead of printing out *["The Doctor", "Tardis", "Dalek"]* as you might expect, you will instead get something like: `java.lang.String;@677327b6`

This output means that when you try to print an Array, Java will simply print a code that represents the *memory location* on your computer for the object that you are printing. In this case, the names array is stored at the memory location @677327b6. That's not very useful. Instead, you can include the following statement to the top of your class:

```
import java.utils.Arrays;
```

and then print by calling:

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

When you import a module into your code, you are borrowing functionality from the larger Java library. Here, we are importing special methods for Arrays, including the `.toString()` method. This method allows us to make an array more print-friendly, and in this case we would get our expected *["The Doctor, "Tardis", "Dalek"]* output. On the AP test, it is assumed that modules such as this are already imported, so you do not need to include them in your answers, but in Eclipse you will need to import them if you want to print out the contents of an array!

Changing the Length of Arrays

This is all great, but what happens when someone new wants to open an account at our bank? Well, it's easy enough to create an instance of the Customer class to represent this person:

```
Customer william = new Customer("William", 6666, 1000000);
```

But how do we get this new customer into our array of customers? Well, this is where arrays get tricky: when we created the `customers` array, we specified that it would have a length of 4. We can't just add a new value, because the following error will occur:

```
customers[4] = william; //error - IndexOutOfBoundsException
```

We could, however, make a new, larger array, and add all of the previous customers to it, and then add our new customer to the end of it:

```
newCustomers = new Customers[5];
for(int i = 0; i < customers.length; i++){  //notice the use of i instead of
      newCustomers[i] = customers[i];       //a for-each loop
}
customers[4] = william;
```

At this point, we can now replace the old array stored as `customers` with the new array:

```
customers = newCustomers;
```

The fact that you can't simply add new values to an array is one of the limitations of this data structure, and in the next cycle we will cover another tool that avoids this problem (ArrayLists). Still, questions on the AP exam will specifically test your knowledge of Arrays and the ways that we can store, retrieve, and manipulate the data inside of them!

Unit 3 Cycle 3: 2-Dimensional Arrays

In the last cycle, we studied a data structure that allows us to store multiple data pieces with the same data type. This is clearly useful - there are many situations where we might want to store a list of names, numbers, or objects. But the world doesn't always fit into a nice, single-file line of information. Consider one of the questions from our last quiz: we used an array to represent a roster of students in a class, and that class had a specific number of seats in each row.

```
String[] roster = {"David", "Dajour", "Tiemoko", "Jessica", "Anthony"}
int seatsPerRow = 3;
```

In this case, we can visualize that the first three students sit in the first row, and the next two students sit in the second row:

David	Dajour	Tiemoko
Jessica	Anthony	

While this visualization works for that question, it isn't ideal. We are forcing our brains to take a single row of information and break it into multiple rows. Wouldn't it be nice if our data structure simply matched the shape of the data?

Visualizing a 2-Dimensional Array

It turns out that you *can* create data structures such as the one above. In order to create this shape in code, consider what makes up the shape itself: Instead of a single collection of names, we have many smaller collections. In fact, we have a collection of collections - in other words, we have an array of arrays.

`String[] arr1`

`String[] arr2`

`String[] arr3`

David	Dajour	Tiemoko
Jessica	Anthony	

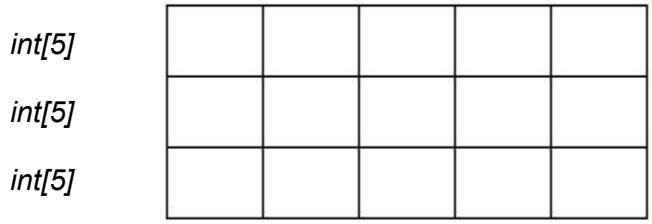
Instead of a single array of strings, what we really want is an array that holds arrays of strings! Luckily, we can do this. Think back to how we create an array of strings in the first place. First, we write the data type, then we write `[]`, then we create a name for the object. That worked fine with Strings, but what is the data type of an array of strings? `String[]` is the data type!

Instantiating 2-D Arrays

So, to create a 2-Dimensional array, you need to write *`datatype`*`[][]`. When you add brackets after a datatype, you are saying that you are creating an array of that type of data. So when you put two brackets in a row, you are creating an array of arrays of that type of data. The first bracket refers to the smaller arrays, and the second refers to the larger array. For example:

```
int[][] myNums = new int[3][5];
```

Here, we are instantiating an array that is 3 items long, and each item in this array is an array of integers that is 5 items long. The shape of this data is:

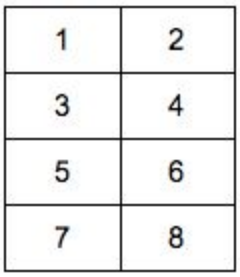


So here we have an array of 3 `int[5]` structures, which is denoted as `int[3][5]`.

Just as you can hard-code in the values in a 1-D array, you can do the same when creating a 2-D array:

```
int[][] yourNums = {{1,2},{3,4},{5,6},{7,8}};
```

If we were to draw out this data, it would look like:



Notice that this is an int[2][4] shape. This means that the there are four rows, each with two columns. While the construction above works, it is more likely that you will use the `new` keyword to create your arrays, and then you will fill in values each position.

Accessing & Changing Values in a 2-D Array

Just as we were able to access and edit the contents of a 1-D array, we can easily do the same for 2-D arrays. Remember that to access the 3rd value in an array named `words`, we would call `words[2]`, because array index positions start counting at 0. How would we find the value 6 in the example above? Well, the value 6 is in the 2nd column, and the 3rd row. Since our structure is `2dArrName[rows][columns]`, we would want to insert the values for the index positions for the 3rd row and 2nd column:

```
yourNums[2][1]; //index 2 = row 3, index 1 = column 2
```

Note that `yourNums` refers to the larger array, so calling `yourNums.length` will tell you the length of the larger array, also known as the number of rows: `yourNums.length` would return 4, and `yourNums[1]` would return the 2nd row of the array, [3,4]. Calling `yourNums[1].length` would tell you how long the sub-array is, in this case it is a length of 2.

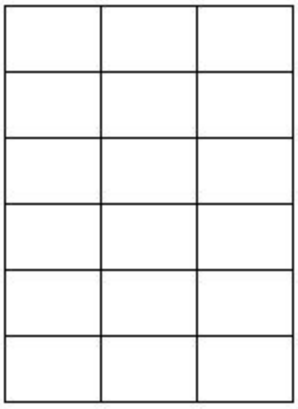
Lastly, note that when you create a 2-D array, every row in the array is going to be the same size. There is no way to create a 2-D array of arrays of a primitive data type where each sub-array has a different size. You can accomplish this by creating a class that contains an of variable size, and then create an array of that class, but let's not get ahead of ourselves just yet!

Iterating over values in a 2-D Array

What if I had a 2-D array, and I wanted to do something to each value in the array? Well, any time that we want to execute the same block of code many times in a row, we should use a loop! Since we have 2 dimensions of the array to process, we will need 2 dimensions to the loop that we are executing:

```
int[][] thoseNums = new int[6][3]; //an array with 6 rows, each row contains an array of 3 ints
```

This array might look like:



So what if we wanted to write a loop that puts the int 7 at every position? Consider first what we would be repeating:

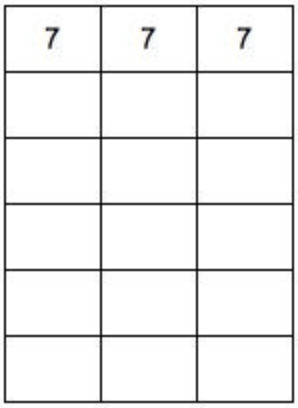
We would need to find a way to put a 7 in the first three columns of an array. Then, we would need to find a way to repeat that process 6 times, once for each row:

```
for(int c = 0; c < 3; c++){
      thoseNums[0][c] = 7;
}
```

This loop would go to the first row in the 2-D array, and then iterate over each of the columns. Notice that we used the iterator `c` here, as a reminder that we are dealing with the columns. If we wanted to be even smarter, we might replace the hard-coded 3 with the length of each row:

```
for(int c = 0; c < thoseNums[0].length; c++){
      thoseNums[0][c] = 7;
}
```

After running this loop, we would have the structure on the right. Now we want to repeat this process for each row:




```
for(int r = 0; r < thoseNums.length; r++){
      //do the loop above
}
```

By inserting the columns loop into the rows loop, we get:

```
for(int r = 0; r < thoseNums.length; r++){
      for(int c = 0; c < thoseNums[0].length; c++){
            thoseNums[r][c] = 7;
      }
}
```

Notice in the bold **`r`** above. Here, instead of only changing the values in the index 0 row, we are changing the row each time we go through the loop. The first iteration will call the internal loop on row 0, the second will call the internal loop on row 1, and so on to fill the entire array with 7's!

You could accomplish similar goals with for-each loops:

```
int[][] thoseNums = new int[7][5]; //creates a 7-row, 5-column array

for(int[] row : thoseNums){
      //do something
}
```

Here, we are using the structure of a for-each loop. First, we specify the data type of the array we are working with, in this case, we are trying to grab individual arrays that are inside of our 2-D array.

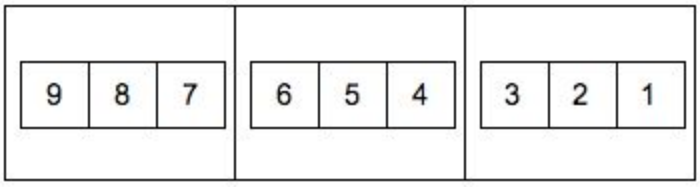
```
for(int[] row : thoseNums){
      for(int c = 0; c < row.length; c++){
            row[c] = 7;
      }
}
```

Notice that, in order to change the value at each position, we need a counter variable that can represent the index positions for each value. Because of this, we couldn't have put another for-each loop inside, because we would have no way to assign a new value!

****A technicality**** While it is useful to think of a 2-D array as a grid of values, it is equally valid to think of them as a 1-D array of arrays:

9	8	7
6	5	4
3	2	1

is basically equal to



So if this array was called `thoseOtherNums`, accessing `thoseOtherNums[1][0]` would return 6! This is the first column of the 2nd row, or the first value of the second sub-array!

2-D Arrays & Methods

Since a 2-D array of objects is a data type, we can write methods that take in 2-D arrays as parameters, and/or return 2-D arrays as output.

```
public String[][] numsToAs(int[][] inputArr){
      //do something
}
```

This might be a method that takes in a 2-D array as a parameter (`inputArr`) and returns a 2-D array as output (of the type `String[][]`).

Let's try to make this method take the input array and return a 2-D array of strings where each value is some number of "A"s. Imagine we have `int[][] nums = {{1,2,3},{1,1,1},{1,2,1}};`
We want our method to return: `{{"A","AA","AAA"},{"A","A","A"},{"A","AA","A"}}`

```
//
public String[][] numsToAs(int[][] inputArr){
      int rows = inputArr.length;
      int cols = inputArr[0].length;
      String[][] answer = new String[rows][cols];

      for(int r = 0; r < rows; r++){
            for(int c = 0; c < cols; c++){
                  String str = "";
                  for(int i = 0; i < inputArr[r][c]; i++){
                        str += "A";
                  }
                  answer[r][c] = str;
            }
      }
      return answer;
}
```

Woah, what's going on here? Start with the creation of the local variables `inputRows`, `inputCols` and the 2-D array of Strings, `answer`. Using these variables, we can create the same shape for our output array as we received in our input array. Then, we use some nested loops: The first iterates over each sub-array, the second iterates over each individual value in the sub-array. The third is where the magic happens - we are taking each integer, and constructing a string of that many "A"s. Once we have the correct string, we put it into the `answer` array at the appropriate row and column.

Changing the Size of 2-D Arrays

Since a 2-D array is actually an array of 1-D arrays, the same rules apply when it comes to changing the size of an 2-D array. If you try to assign a new value to an index position that does not exist in the array, you will get an IndexOutOfBounds error:

```
boolean[][] facts = {{true,false},{false,true},{false,false}};
facts[1][0] = true; //this is valid

facts[3] = {true,true};  //IndexOutOfBounds error!
```

In order to add a new sub-array, you would need to first construct a new, larger array, and then copy the values from `facts` into the new array, and finally re-assign `facts` to be equal to the new, larger array:

```
boolean[][] temp = new boolean[facts.length + 1][facts[0].length];
for(int r = 0; r < facts.length; r++){
     temp[r] = facts[r];
}
```

At this point, temp should look like:

```
{{true,false},{false,true},{false,false},{false,false}}

//why is the last sub-array {false,false}? Remember that when you use the new
//keyword to create an array, it fills that empty array with default values based
//on the data type for that array.  The default for booleans is false.
```

Now we can set:

```
facts = temp;
```

Following these steps (make a new, larger array; copy values from the original to the new array; set the original array equal to the new larger array), you can increase the number of rows in a 2-D array.

But what if we wanted to change the number of columns in the 2-D array, not the rows?

Well, you would need to follow a similar operation, except you would be instantiating a new array with a larger column value, and your copying loop would need to iterate over each individual value and copy them into the empty positions in the new array. For example:

```
String[][] words = {{"red","yellow"},{"one","two"},{"rock","paper"}};

String[][] temp = new String[words.length][words[0].length + 1];
//we have instantiated an empty array of size [3][3]
```

```
for(int r = 0; r < words.length; r++){
      for(int c = 0; c < words[0].length; c++){
            temp[r][c] = words[r][c];
      }
}

words = temp;
```

This loop will copy individual values from the sub-arrays (instead of the full sub-array, as seen in the first example). What will be left in the empty position for each now-larger sub-array? `""`, which is the default empty value for `Strings`.

```
System.out.println(Arrays.toString(words));
//prints [["red","yellow",""]["one","two",""]["rock","paper",""]]
```

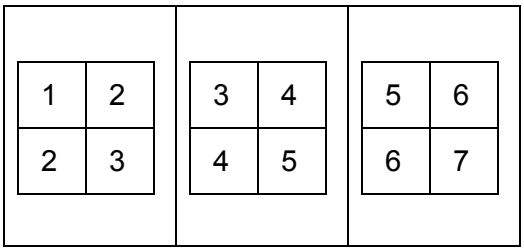
The AP test is unlikely to ask you to change the size of a 2-D array, but you might need to do so at some point in one of your programs, or in one of our labs!

3-D Arrays?

You might be wondering, 'can I create 3-D arrays, or even larger structures?' Well yes, yes you can! The AP test has never had a question about any structures larger than 2-D, but here is an example:

```
int[][][] lotsOfNums = {{{1,2},{2,3}},{{3,4},{4,5}},{{5,6},{6,7}}};
//this is an array of 2-D arrays, or, an array of arrays of integer arrays
```

The shape of this array might look like:



While this is interesting to think about, it is likely easier for you to create a class that holds a collection of this data, and store it as a 1-D array of this class type.

Unit 3 Cycle 4: ArrayLists

Over the last two cycles, we have been studying arrays of objects, including both one and two-dimensional data structures. These structures allowed us to store, access, and manipulate many pieces of data without creating many variables. While these structures are incredibly important to programming in general (and to be clear - they *will* play a large role in the AP test), sometimes it is frustrating to use them. For example, we might have the following code from our last reading:

```
String[] roster = {"David", "Dajour", "Tiemoko", "Jessica", "Anthony"};
```

As we saw in the last cycles, if we needed to add a new student to the roster, it would take a lot of work:

```
String[] temp = new String[roster.length + 1];
for(int i = 0; i < roster.length; i++){
      temp[i] = roster[i];
}
roster[5] = "William";
```

And we would need to do the same thing *each time* we want to add a new member to the array. Annoying, right? Well, it turns out that there is another tool that we could use to accomplish similar work: the **`ArrayList`** class.

ArrayLists

An **`ArrayList`** is a class that can be imported into your program that has the data structure of a normal array, but includes many useful methods similar to `String` methods. This class is not a part of the standard Java subset, which means that it is not automatically included in your programs when you write a new program file, so you will need to add the following to the top of your file:

```
import java.util.ArrayList;
//or
import java.util.*;  //the * means "all", so this imports all classes in the
                        java.util package
```

This import statement tells your program to import the functionality of `ArrayLists` so that you can use them in your code. You might be wondering: "if this class is part of the Java utilities package, why isn't it automatically included?" Well, consider how many other hundreds of classes are also available in the Java utilities package. Automatically including them all would increase the size of your project, and you likely would not use many or any of them. Instead, Java allows you to import the utility classes that you need in your program, and only those utilities.

*****Note: for the AP exam, the test uses the "AP Subset" of Java, which *includes* `ArrayLists`, so you do not need to call the import statement above when writing code on the test. You DO need to import the class, however, when writing any code that includes `ArrayLists` in Eclipse!*****

The ArrayList Constructor

Like any other class, the imported ArrayList class has a constructor that allows you to create instances of that class. The following syntax should be used when creating an ArrayList:

```
ArrayList<dataType> varName = new ArrayList<dataType>(size);
```

Note that you will replace *dataType* with a data type, such as `String`, *varName* with the name of the instance that you are creating, and *size* with a size for your array (the size is an optional parameter - if you put nothing in this space, the default size of 10 will be used).

```
ArrayList<String> myWords = new ArrayList<String>(3);
ArrayList<Integer> myNums = new ArrayList<Integer>();
```

Here we have created an ArrayList of Strings named `myWords` that has a size of 3. We have also created an ArrayList of `Integer`s that has a size of 10. Notice that we used the ***wrapper class*** *`Integer` instead of the primitive type `int`. This is because you can only create ArrayLists of objects, and primitive types such as `int` are not objects. That's OK - `Integer` objects can be used the same way as `int` objects, and we can even convert values between them.*

So you might be thinking, "this looks more complicated than a normal array, why would I even use this?" Well, look what we can do:

```
myWords.add("red"); //adds a String to the ArrayList
myWords.add("green").add("yellow");  //you can chain together methods
myWords.add("blue");
```

But wait, wasn't the size of the `myWords ArrayList` only 3? This is the best part of `ArrayLists` - if you tried to add a new value beyond the length of an Array, you would get an out-of-bounds error, but ArrayLists allow you to add new values without an error. The ArrayList simply increases it's size when you try to add a new value beyond its current size! **Notice that we need to use the .add() method, not an index position!**

```
System.out.println(myWords); //would print ["red","green","yellow","blue"]
```

ArrayList size vs capacity

The size of an ArrayList isn't the same as its capacity. This can be confusing.

```
myNums.add(3).add(4).add(5); //adding some values to the myNums ArrayList
myNums.size(); //this would return 3
```

Why is the size of `myNums` 3, when the default constructor above assumed that we will hold up to 10 values? Well, the *size* of an ArrayList is the number of values that are currently in the list, whereas the *capacity* is the current potential size of the list. Note that currently, both the size and capacity of `myWords` would be 4. Generally speaking, we will be using the size of ArrayLists in our programs, rather than the capacity.

ArrayList Methods

The following methods represent some, but not all, of the methods that are defined in the `ArrayList` class. You can use any of them on the AP exam, and in your own programs in Eclipse (assuming you have imported the `java.util.Arraylist` class).

return type & method name	description
boolean add(Element e)	adds the element e to the end of the ArrayList. Returns true after completing this task
void add(int i, Element e)	adds the element e at the index position i in the ArrayList. Other items in the ArrayList after this index are *'pushed'* back one position
void clear()	removes all elements from the ArrayList
Object clone()	returns a copy of the ArrayList - you might use this method if you want to create a new ArrayList with the same values, without setting both objects equal to the same memory location (ie `newList = oldList.copy();`)
boolean contains(Element e)	returns true if the list contains the input parameter, false otherwise
Element get(int i)	returns the element that is found at index position i
int indexOf(Element e)	returns the index position of the first instance of element e, returns -1 if the ArrayList does not contain the element
int lastIndexOf(Element e)	returns the index position of the last instance of element e, returns -1 if the ArrayList does not contain the element
boolean isEmpty()	returns true if all elements of the ArrayList are *null* (note - a 0 is not a null value!)
Element remove(int i)	removes the element at index position i from the ArrayList, and returns that element. All elements beyond the removed element are pulled back one index position
boolean remove(Element e)	removes the first instance of element e in the ArrayList, and returns true if this task is completed (false if there is no element e in the list)
Element set(int i, Element e)	sets the index position i to the element e, and returns that element
Object[] toArray()	returns the list as a normal array of objects

Remember, these are *instance methods*, which means that you need an instance of this class to call these methods. For example:

```
ArrayList<Double> otherNumse = new ArrayList<Double>();
otherNums.add(0.2).add(3.4).add(9.0); //add three values to the ArrayList
otherNums.add(1, 5.5); //add a 5.5 at position 1: [0.2, 5.5, 3.4, 9.0]
otherNums.indexOf(3.4); //returns 2, the current index for the value 3.4
otherNums.add(otherNums.remove(0.2)); //adds the result of removing 0.2
//results in [5.5, 3.4, 9.0, 0.2]. Note that calling .remove() returns a value!
```

Converting ArrayLists to Arrays

While this method was in the chart above, this concept deserves a closer look. Imagine that on a test, you were asked to write a method that takes in an array of Strings, and adds the String "fish" to the start of this array a random number of times between 1 and 5. Working with standard arrays might be difficult, since we would be changing the size of the input array. We might want to use an ArrayList instead, but the question *requires* you to return an `Array`, not an `ArrayList`:

```
public String[] addRandomFish(String[] inputArr){
      //do something
}
```

You can still use an `ArrayLis`t, but you would need to convert your final list back to a regular array:

```
public String[] addRandomFish(String[] inputArr){
      ArrayList<String> answer = new ArrayList<String>();
      int rand = (int)(Math.random()*5 + 1);
      for(int i = 0; i < rand; i++){
            answer.add("fish");
      }
      for(String s : inputArr){
            answer.add(s);
      }

      //We now have a correct collection of Strings, but in the wrong format.
      //The method returns an array, so we need to convert answer to an array:
      return answer.toArray();
}
```

In the example above, if we had simply returned `answer`, our program would have had an error, because we would have been returning the wrong type of data!

2-D ArrayLists?

Yes, you can create an ArrayList that contains sub-ArrayLists. How would you do this? Well, consider the data type that you are working with:

```
ArrayList<ArrayList<String>> bigArray = new ArrayList<new
ArrayList<String>()>();
//or
ArrayList<String[]> otherBigArray = new ArrayList<String[]>();
```

This is ugly and confusing, even if the second option is a little better. If you are given a question that deals with 2-D arrays, stick with the general Array class instead of trying to use an `ArrayList`!

ArrayLists & the AP Test

Generally speaking, the AP test will give you regular arrays to work with. This does not mean, however, that you can't use ArrayLists - you just need to make sure that your work returns the right type of data, as seen above. Occasionally, multiple choice questions will specifically work with ArrayLists, but more often you will use them on the Free Response section of the test when writing your own methods and classes. AP Test graders do not value one data structure over another (Arrays vs. ArrayLists), so you should use the one that seems the most appropriate for the task at hand, and the one that you are most comfortable with.

ArrayLists Pro's and Con's

Considering the information above, if you have a choice in terms of what data structure to use in one of your programs, remember the following traits for the ArrayList data structure:

Pro's:

- ArrayLists have variable size/capacity
- ArrayLists have many helpful methods such as:
    - .add(), .add(int i, Element e)
    - .get(int i)
    - .indexOf(Element e)
    - .remove(Element e), .remove(int i)
    - .contains()

Con's:

- ArrayLists are more complicated to create
    - `Type[] name = [...];` vs. `ArrayList<Type> name = new ArrayList<Type>();`
- You can't create an ArrayList and set it's values in the same line, like with Arrays
    - `String[] words = ["word1","word2"];` *//no equivalent to this with ArrayLists!*
- To use ArrayLists, you have to import `java.util.ArrayList`

Unit 4 Cycle 1: Classes & Inheritance

In Unit 3 we learned about a number of ways to store, access, and edit groups of information. Whether using arrays, two-dimensional arrays, or ArrayLists, we were able to take large collections of the same data type and organize them to make them more useful in our programs. We will keep using these structures, but what if you wanted to store a collection of *different* types of data? Consider the following:

```
String[] names = {"Alice", "Ada", "Grace"};
int[] ages = {10, 20, 30};
boolean[] leftHanded = {true, false, false};
```

This might seem like an OK way to collect some facts about a few people, but accessing the data for a single person is going to be difficult. If we wanted to see everything about 'Grace', we would need to access `names[2]`, `ages[2]` and `leftHanded[2]`. Luckily, we already learned a way to encapsulate these different types of data - Classes:

```
public class Person{
      //fields
      private String name;
      private int age;
      private boolean leftHanded, isDoctor;

      //constructor
      public Person(String n, int a, boolean l, boolean d){
            name = n;
            age = a;
            leftHanded = l;
            isDoctor = d;
      }

      //methods
      public String getName(){  return name; }
      public int getAge(){  return age; }
      public boolean isDoctor(){ return isDoctor; }
      public String about(){ return name + " is " + age + " years old!"; }
}
```

Now we can create some Person objects using the constructor in another file:

```
Person alice = new Person("Alice", 10, true, false);
Person ada = new Person("Ada", 20, false, false);
Person grace = new Person("Grace", 30, false, true);
```

All of the objects above belong to the Person class, and this class allows us to wrap up different types of data and methods into a single structure. We could even collect these objects into an array:

```
Person[] people = {alice, ada, grace};
```

Notice that `Person` is a data type, so an array of these objects is of the type `Person[]`.

Inheritance Basics

So we wrote a definition for the `Person` class, which is nice to have. Take a closer look at `grace` - her value for the `isDoctor` field is `true`. What if our program was going to include multiple `People` who were also doctors? Beyond that, what if we want our `People` who are doctors to have extra *fields* (such as '`specialty`' or '`patients`') and extra *methods* (such `addPatient()` and `removePatient()`)?

We *could* do a lot of extra work by defining a new class called `Doctor` that has all of the same fields and methods as the `Person` class, an updated constructor for this class, and all of the new methods that we want `Doctor`s to have. But wouldn't it be nice if we could simply say that Doctor objects are just Person objects with a few extra bits? Luckily, we can using ***inheritance***.

When creating a class, we can say that one class `extends` another class:

```
public class Person{
      //see earlier definition
}

public class Doctor extends Person{
      private String specialty; //fields unique to Doctors (not Person objects)
      private ArrayList<Person> patients;

      public Doctor(String n, int a, boolean l, String s){
            super(n,a,l,true); //uses the constructor of the superclass, Person
            specialty = s;
            patients = new ArrayList<Person>(); //empty ArrayList
      }

      //methods unique to the Doctor class (Person can't do this!)
      public void addPatient(Person p){ patients.add(p); }
      public String getSpecialty(){ return specialty; }
      public Person[] getPatients(){ return patients.toArray();  }
}
```

Here we are saying that any object belonging to the `Doctor` class is an extension of the `Person` class. Notice the bold term **`super`**`()` in the constructor for the `Doctor` class. Here we are saying that when you construct a `Doctor` object, the first thing you do is complete the constructor for the `Person` class, and then you set the fields that are specific to `Doctors`. This is why our `Doctor` constructor still takes in similar inputs to the `Person` class (`String n`, `int a`, `boolean l`...) - `Doctor`s are `Person`s too, so we need to complete the superclass constructor and give it the appropriate inputs. Why didn't we use `boolean d` in the `Doctor` constructor? Well, for a `Person` object that field represented whether the `Person` was a doctor, so we can assume for the `Doctor` class that that input for the `Person` constructor will be `true`.

```
Doctor moreau = new Doctor("Moreau", 50, false, "Geneticist");
//                         n       a     l          s
```

When we create this Doctor, the first three inputs are sent to the constructor from the Person superclass using **`super`**`(n,a,l,true)`. The remaining input, `s`, is set to this `Doctor`'s specialty field.

There are many important implications for this extension:

1) The `Person` class is considered the *superclass* in this relationship
2) The `Doctor` class is considered the *subclass* in this relationship
3) Any fields or methods belonging to the `Person` are *inherited* by `Doctor` objects
4) The constructor for the `Doctor` class must include the constructor for its super-class, `Person`
5) Any object that is a `Doctor` is also a `Person`, which means that `Doctor` objects have multiple data types!

Consider our the `Person` objects that we created above:

```
Person alice = new Person("Alice", 10, true, false);
Person ada = new Person("Ada", 20, false, false);

//create a Doctor object using the constructor
Doctor phil = new Doctor("Phil", 60, true, "Gastronomy");

//create a Doctor by invoking the superclass as a data type
Person oz = new Doctor("Oz", 55, false, "Herbology");

//use methods from the Doctor class
phil.addPatient(alice);
phil.addPatient(ada);
phil.getPatients(); //returns an array containing alice and ada

//use methods from the Person class - these methods are inherited!
phil.getName(); //returns "Phil"
oz.getAge(); //returns 55

//notice that instructions getName() and getAge() methods are not a part of the
//Doctor class. Doctor extends Person, so all Doctor objects have the methods
//defined in the Person class as well!
```

Data Types

The whole reason that we wanted to create the `Doctor` class in the first place was that we wanted to reuse the data structures and methods of `Person`, with a few added pieces. But we also said that a `Doctor` *is* also a `Person`, right? That means that we can do the following:

```
Person[] peeps = {alice, ada, grace, phil, oz};
```

We can add `phil` to an array of `Person` objects, because `phil` has more than one data type - `phil` is both a `Person` object, and a `Doctor` object! This would allow us to do the following without an error:

```
for(Person p : peeps){
     System.out.println(p.getAge());
}
```

This works because every object in `peeps` is a `Person`, and every person has a `.getAge()` method!

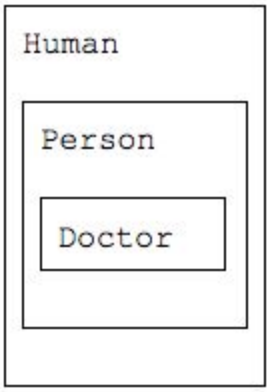
Class Hierarchies

We said that every `Doctor` extends the `Person` class, so `Doctor` has a second data type. You might be wondering if that means that an object can have even more data types - the answer is complicated. When you define a class, you can only **`extend`** one other class. This means that once we have written the `Doctor` heading as:

```
public class Doctor extends Person{
…
}
```

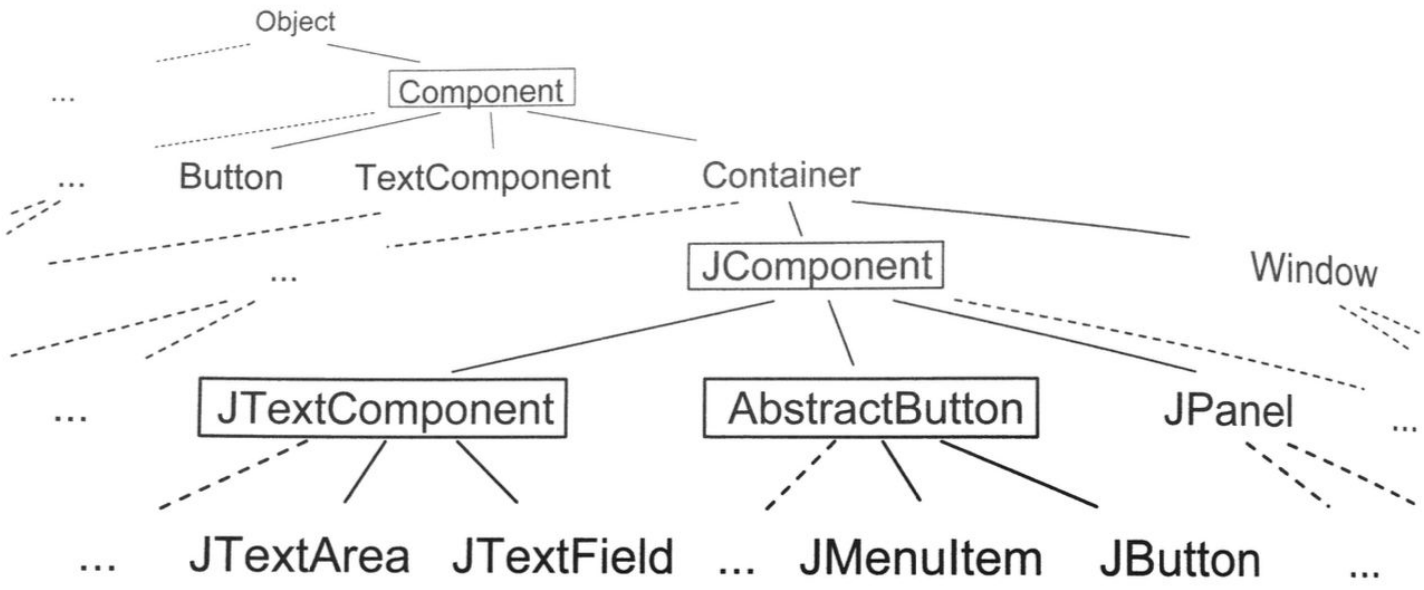
The `Doctor` class cannot extend any other classes. But what about the `Person` class? What if we added the following to the heading of `Person`:

```
public class Person extends Human{
…
}
```



As long as our `Person` constructor correctly fills the `Human` constructor, this will work. This would mean that all `Person` objects also have the data type of `Human`. And since all `Doctor` objects are `Person` objects, all `Doctor` objects *also* have the data type of `Human`! (Ignore the Doctor Who error for now - no, he is not a human…)

In fact, this multilevel inheritance goes much farther up the chain. In Java, **all classes eventually inherit from the `Object` class**.



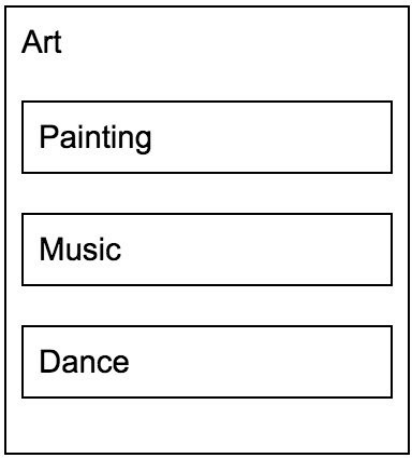
While the items in this hierarchy have not been covered in this course so far, they illustrate the fact that all of the categories of classes that are provided inside of Java eventually extend from the `Object` class!

Abstract Classes

So far in our study of Java, all of the classes we have worked with have included a *constructor* - a special method that defines the process of creating an *instance* of that class. The `Person` and `Doctor` classes, from earlier in this reading, are examples of that - by calling the constructor for the `Person` class we can instantiate (create a new instance of) the `Person` class:

```
Person fred = new Person("Fred", 27, true, false);
```

But do we actually need a constructor for every class? What if we were making a program that had a superclass that we never planned to instantiate? For example, maybe we were making a program that deals with art. All of the `Art` will have similar fields (such as `'artist'`, `'title'` and `'year'`), and similar methods (such as `getInfo()`). But you never plan to create a generic `Art` object, so you don't want to write a constructor for that superclass. Instead, you will write a constructor for the subclasses `Painting`, `Music` and `Dance`.



If a class has no constructor, we call it an **abstract class**, which means it can contain **abstract methods**. Consider the following:

```
public abstract class Art{
     public abstract String getInfo(); //we are saying that any Art object can
}                                      //have this method.  Abstract methods have
                                       //no body { }
//**any class that has an abstract method needs to be an abstract class!
```

```
public class Painting extends Art{
     private String artist, title, paintType;
     private int year;

     public Painting(String p, String a, String t, int y){
          paintType = p;
          artist = a;
          title = t;
          year = y;
     }

     public String getInfo(){
          return title + " by " + artist + ". " + paintType + ", "+ year;
     }

     //other methods of the Painting class
}
```

It doesn't make sense to create a generic `Art` object for our program, but we want to make sure that any piece of art has a method called `getInfo()`. In addition, we can now store `Painting`, `Music`, and `Dance` objects in a single array, because they all share the `Art` data type.

Overriding Methods

So far we have defined a *subclass* of `Person` to include special fields and methods. This allows us to include the fields and methods of `Person`, but what if we want one of those methods to work differently? For example, our `.getName()` method for the `Person` class returns their first name, so it will do the same thing for `Doctor` objects. But what if we wanted the `Doctor .getName()` function to return "Dr. " in front of their name? We can ***override*** a method to accomplish this.

```
public class Doctor extends Person{
      //fields, constructor and methods hidden to save space

      public String about(){  //overriding the about() method
            return "Dr. " + name + " specializes in " + specialty + "!";
      }

      public String getName(){  //overriding the getName() method
            return "Dr. " + super.getName();
      }
}
```

Here we are ***overriding*** the `about()` and `getName()` methods for `Doctor` objects. For the `about()` method, a brand new set of instructions replaces the method from the `Person` class. If you check the `Person` class definition provided earlier, a `Person`'s `about()` method returns something like *"Ada is 20 years old!"* For the `Doctor` class, the new overridden `about()` method would return something like *"Dr. Oz specializes in Herbology!"*

You might also notice the bold term **`super`** in the overridden version of `getName()`. We used this keyword earlier in the constructor for the `Doctor` class, because the constructor for a `Doctor` requires you to use the constructor for its superclass, `Person`. Here we are using the keyword **`super`** to access one of the methods of the superclass, `Person`: the new version of `getName()` *calls* the `getName()` method from the `Person` class. When we call a `Doctor`'s `getName()` method, it takes the string "Dr. " and adds it to the result of calling the `Person` class' `getName()` method.

When you use the keyword `super`, you are accessing information (either data or methods) from the class that you have extended!

```
Person stuart = new Person("Stuart", 17, true, false);
Person zhivago = new Doctor("Zhivago", 35, false, "Surgery");
Doctor who = new Doctor("Who", 1000, true, "Time Travel");

stuart.about();  //returns 'Stuart is 17 years old!"
zhivago.about(); //returns 'Dr. Zhivago specializes in Surgery!"
who.about();    //returns 'Dr. Who specializes in Time Travel"

zhivago.getName(); //returns 'Dr. Zhivago'
who.getName();    //returns 'Dr. Who'
```

Polymorphism

All of this talk about extending classes and inheritance gives us the ability to do some important, if complicated, work with our classes. By creating a system of interacting, interdependent classes, we will be able to represent more nuanced and complex data structures, more varied and useful methods, all without adding a frustrating number of class names and data types. Consider the example below:

```
public abstract class Shape{
      public static abstract draw();
}
```

```
public class Triangle extends Shape{
      private int[] sideLengths;

      public Triangle(int[] l){  sideLengths = l; }

      public static draw(){
            //draw a triangle using the data in the fields for this object
      }
}
```

```
public class Circle extends Shape{
      private int radius;

      public Circle(int r){  radius = r; }

      public static draw(){
            //draw a circle using the data in the field for this object
      }
}
```

While triangles and circles are very different - they have different data, and the way that we draw them is very different - the definitions above allow us to do some powerful *abstraction*. Instead of defining differently-named methods for drawing each different shape, we use inheritance to guarantee that any object that is a `Shape` will have a method to `draw()`. This would allow us to do the following:

```
Triangle t1 = new T`riangle({3,4,5});
Circle c1 = new Circle(5);
Triangle t2 = new Triangle({4,5,6});

Shape[] shapes = {t1, c1, t2};

for(Shape s: shapes){  s.draw();  }
```

This concept, of multiple related classes that call different versions of the *same* method is an example of ***polymorphism***. This feature is an important component of building programs that are based on *classes* and *objects*.

Unit 4 Cycle 2: Interfaces

In our last cycle, we learned about **superclasses** and **subclasses**. By making one class extend another, we were able to reuse fields and methods, and we were able to create objects with different features but a shared data type. This is an important part of Object-Oriented Programming (OOP), but it isn't always a perfect solution to the problems that we are trying to solve. Consider the following situation:

```
public class Mammal{
      //fields, constructor and methods
}
```

```
public class DomesticAnimal{
      //fields, constructor and methods
}
```

```
public class Cow extends Mammal ~~extends DomesticAnimal~~{    //uh-oh...
      //fields, constructor and methods
}
```

Cows are mammals, but they are also domesticated. While it is true that all cows are mammals, and that cows are domesticated, it isn't true that all domesticated animals are mammals (ducks, for example, can be domesticated, but they aren't mammals). Nevertheless, we want the Cow class to inherit data and methods from *both* the `Mammal` and `DomesticAnimal` classes. Unfortunately, as we learned in the last cycle, we can't say `public class Cow` ~~`extends Mammal extends DomesticAnimal`~~`{...}` because a class can only extend one other class.

So how can we get `Cow` objects to inherit from both the `Mammal` and `DomesticAnimal` classes? Interfaces!

Interfaces

An interface is a document that is *similar* to a Java class:

```
public interface DomesticAnimal{
      private static final String food;  //'final' makes this a constant, which
                                         //is a variable that can't change
      public String getFood();
      public void feed();
}
```

This code might be saved in Eclipse as *DomesticAnimal.java*. As you can see above, an interface can have variables (such as `food` above) and methods (such as `getFood()` and `feed()`). One thing that an interface **cannot** have is a constructor. You will never create an instance of `DomesticAnimal` - you might instantiate a `Cow` object, but there is no constructor for a generic `DomesticAnimal`. Notice in this interface the methods `getFood()` and `feed()` do not have any statements between `{}` - the interface doesn't say how these methods are supposed to work. Instead, any class that uses this interface *must* include this method and provide a definition for how that method works.

```
public interface DomesticAnimal{
      private static final String food;  //'final' makes this a constant, which
                                         //is a variable that can't change
      public String getFood();
      public void feed();
}
```

```
public class Cow implements DomesticAnimal{
      private int weight;

      public Cow(int w){
            weight = w;
            food = "hay";
      }

      public String getFood(){ return food; }
      public void feed(){ System.out.println("Feeding cow…"); }
      public void speak(){ System.out.println("Moo"); }
}
```

Similar to the way that one class can *extend* another, a class can ***implement*** the features of an interface. In the example above, the `Cow` class is inheriting the data and methods of the `DomesticAnimal` interface. Notice that the `Cow` class provides definitions for the interface methods `getFood()` and `feed()` - if these had not been provided, there would have been an error. In addition, remember that other classes that implement `DomesticAnimal` might have different definitions for these methods (this is an example of *polymorphism*).

Lastly, just as with inheritance through classes, when a class implements an interface it also inherits a new data type. This means that any `Cow` object is also a `DomesticAnimal` object, which could allow us to store different objects in an array of the same type! So why is this preferable to creating another class? Well, consider the following:

```
public interface Ungulate{ //ungulate is a word for animals that have hooves
      public boolean hasHooves = true;
      public void walk();
}
```

```
public class Cow implements DomesticAnimal, Ungulate{
      //field, constructor and methods from above

      public void walk(){   System.out.println("clop clop"); }
}
```

One class can implement *multiple* interfaces. Now any `Cow` object inherits from `DomesticAnimal` ***and*** `Ungulate`, and shares both of these data types! Separate implemented interfaces with commas!

Code Sample

Let's see some of the usefulness of interfaces in action by putting it all together:

```
public interface DomesticAnimal{
      private static final String food;  //'final' makes this a constant, which
                                         //is a variable that can't change
      public String getFood();
      public void feed();
}
```

```
public interface Ungulate{ //ungulate is a word for animals that have hooves
      public boolean hasHooves = true;
      public void walk();
}
```

```
public class Cow implements DomesticAnimal, Ungulate{
      private int weight;
      public Cow(int w){
            weight = w;
            food = "hay";
      }
      public String getFood(){ return food; } //DomesticAnimal
      public void feed(){ System.out.println("Feeding cow…"); } //DomesticAnimal
      public void speak(){ System.out.println("Moo"); }
      public void walk(){   System.out.println("clop clop"); } //Ungulate
}
```

```
public class Duck implements DomesticAnimal{
      private String breed;
      public Duck(String b){
            breed = b;
            food = "duck food";
      }
      public String getFood(){ return food; } //DomesticAnimal
      public void feed(){ System.out.println("Feeding duck…"); } //DomesticAnimal
}
```

Now we have two classes that implement `DomesticAnimal`: `Cow` and `Duck`. Our Cow objects have the benefit of inheriting from both the `DomesticAnimal` class and the `Ungulate` class, and our `Duck` objects only inherit from the `DomesticAnimal` class. Even though `Ducks` and `Cows` are significantly different, that share the `DomesticAnimal` data type. This would allow us to store instances of these classes in the same array or ArrayList, or to use `DomesticAnimal` as a data type for inputs or return types in a method.

Practice

```
//create two instances of the Cow class and the Duck class
Cow c1 = new Cow(300);
DomesticAnimal c2 = new Cow(400);

Duck donald = new Duck("mallard");
DomesticAnimal duckFace = new Duck("also mallard");


//using these instances, call the .walk() method for the Cow objects, and the
//.feed() method for the Duck objects.  In a comment, write the expected output

c1.walk()
donald.getFood();
donald.walk();//ERROR!




//create an array that stores all 4 of your instances

DomesticAnimal[] myAnimals = {d1, donald, c1, c2};
ArrayList<DomesticAnimal> myAnims = new ArrayList<DomesticAnimal>();
myAnims.add(d1).add(donald).add(c1).add(c2);


//write a loop that goes through each item in the array and prints the result
//of it's getFood() method

for(DomesticAnimal animal : myAnimals){
      animal.getFood();

}


//write a method that takes in a DomesticAnimal as a parameter and prints a
//combination of the .getFood() and .feed() methods for this parameter

public boolean isDelicious(DomesticAnimal animal){
      //return animal.getFood().equals("corn");
      if(animal.getFood().equals("corn")){
            return true;
      } else {
            return false;
      }
}
```

Unit 5 Cycle 1: Recursive Methods

In the last unit, we studied **class inheritance** and **interfaces**. Using these structures, we were able to add more complex functionality to the basic classes and objects that we had been using previously. Similarly, in this unit we will take a concept that we are already familiar with - **methods** - and add a new layer of complexity: **recursion**. Recursive methods are complicated, and they are consistently the topic that students struggle with the most in the MC portion of the AP exam. The payoff for this complexity is twofold: first, recursive methods can be more efficient at solving some problems, and two, some problems cannot easily be solved without a recursive approach. Don't worry - while recursive functions are challenging, they are not necessarily more complex than a topic such as nested loops or 2D arrays! Consider the example below:

```
public int magic(int num){
      int x = 3;
      for(int i = 0; i < num; i++){
            x = ((2*x) - 1);
      }
      return x;
}
```

The method above is simply a method like any other that we have seen before. It is going to return an `int` when it is done running, it takes in an input parameter, `num`, it has a variable that it tracks, `x`, and it changes the value of `x` inside of a loop. What would be the result of calling the following?

```
magic(0);  //
magic(1);  //
magic(2);  //
magic(3);  //
magic(4);  //
```

You might consider the output of these calls to `magic()` as a *number pattern*. How would we figure out the next number in the pattern? We could call `magic(5)`! The resulting values in the sequence would be *3,5,9,17,33,65*…

Iterative Methods

The definition for `magic()` above is considered an **iterative** method - it **iterates**, step-by-step (often using a loop) to calculate information and then return a value. Between each iteration, the method needs to store some value before moving on to the next step. In this case, in order to determine the 6th number in the sequence, the method needed to find the 1st number, remember it, and process it to create the 2nd number, remember it, and process it to create the 3rd number - and so on until we have reached the value we are looking for.

All of the programs that we have written in this course so far have been *iterative*, because they follow a step-by-step process, even if some of those steps get repeated in a loop.

Recursive Methods

A **recursive** method is different. Instead of following a sequence of steps from beginning to end, a **recursive** method calls itself in a new way in order to process information. This means that somewhere in the definition for a **recursive** function, the function itself is called. Consider the following:

```
public int recursiveMagic(int num){
      if (num == 0){
            return 3;
      } else {
            return (2*(recursiveMagic(num-1) - 1);
      }
}
```

Let's think through what is happening here. First, this method is going to return an `int`, and it takes in an `int` as a parameter, just like our first method. What would happen if we called `recursiveMagic(0)`? Well, if `num` is 0, then the first condition is true, and the method will simply `return 3`. But what if we call `recursiveMagic(1)`?

```
                                  1
public int recursiveMagic(int num){
      if (num == 0){  //1 != 0, so false
            return 3;
      } else {
            return (2*(recursiveMagic(num-1) - 1);
      }     //return (2 * recursiveMagic(0) - 1)
}
```

So what exactly is happening in this `return`ed value? Well, the method is supposed to return 2 times the result of calling `recursiveMagic(0)` and subtracting 1. We already know that `recursiveMagic(0)` returns a 3, so the final result is `(2 * 3 - 1)`, or 5. What if we called `recursiveMagic(2)`? The result would be `(2 * recursiveMagic(1) - 1)`, which is the same as saying `(2 * (2 * recursiveMagic(0) - 1) - 1)`.

In this pattern, you might notice two features to this method: first, the method needs to include **a call to itself**. Any method definition that calls itself is going to be recursive. Second, the method has a **base case**, or a condition that causes the self-calling to stop. In our example above, the condition `if(num == 0){ return 3; }` is the base case. Without a base case, the method might call itself infinitely:

```
public int badRecursion(int num){
            return (2 + badRecursion(num - 1));
}
```

This method will cause an infinite loop! Calling `badRecursion(1)` would return `(2 + badRecursion(0))`. But what happens when we call `badRecursion(0)`? It returns `(2 + badRecursion(-1))`, which returns `(2 + badRecursion(-2))` and so on infinitely. A recursive method needs to be told when to stop calling itself, and this is often accomplished with an `if()` statement.

Strengths & Weaknesses: Recursive Methods

So, why might we choose to use a recursive method rather than an iterative one? Both strategies can result in the same outputs: our `magic()` and `recursiveMagic()` methods both can accurately identify the nth term in our number sequence. The difference can be seen in the way that both methods use system resources: in the iterative method, each time through the loop the method changes the value of a variable. This operation only takes up one position in the system's memory, but it takes time to calculate, save, and repeat this process. Calling `magic(5)` or `magic(100)` might still seem pretty fast, but calling `magic(1000000)` will take some time - the method will need to change the value of the same variable 1000000 times!

What about the recursive solution? Well, instead of updating the same memory location many times, the recursive method calculates each recursive call simultaneously, only assigning a value to memory once the calculations have been added together. This can be faster, but it also consumes another system resource: RAM. The computer needs to temporarily hold the value of each recursive call all at once in order to combine them for the final answer. As a result, if a recursive function might strain the memory resources of a program, potentially causing a crash. Generally speaking, a recursive function can be more efficient (i.e. *faster*) than an iterative function, but a poorly written one can overtax a system and cause a program to crash.

Example Comparison

<table>
<tr><th colspan="2">Iterative Solution</th></tr>
<tr><td>

```
public int iterativeFib(int n){
      if(n == 0){ return 0;}
      else{
            int x = 0;
            int y = 1;
            for(int i = 0; i < n; i++){
                  int temp = x + y;
                  x = y;
                  y = z;
            }
            return y;
      }
}
```

</td><td><code>iterativeFib(12)</code> would be the 12th fibonacci number:
1 1 2 3 5 8 13 21 34 55 89 144

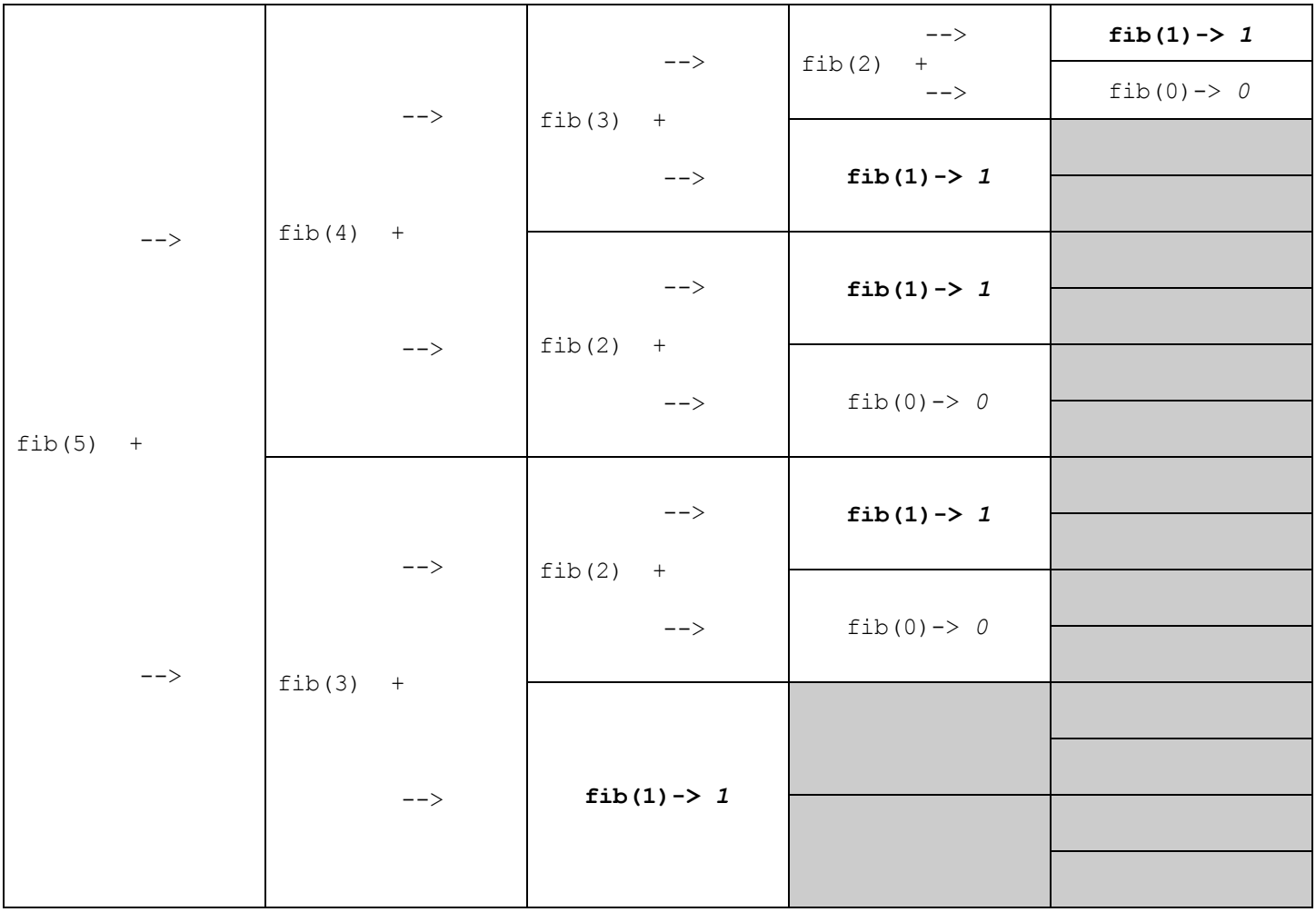
This method call would require the loop to re-assign each variable 10 times.</td></tr>
<tr><th colspan="2">Recursive Solution</th></tr>
<tr><td>

```
public int recursiveFib(int n){
      if(n == 0 || n == 1){ return n;}
      else{
            return (recursiveFib(n-1) +
                  recursiveFib(n-2));
      }
}
```

</td><td><code>recursiveFib(12)</code> would be the 12th fibonacci number:
1 1 2 3 5 8 13 21 34 55 89 144

Since this method calls itself twice in each layer, the total number of recursive calls is exponential, and large values for n could cause problems for your program.</td></tr>
</table>

In the example on the previous page, consider the note about exponential recursive calls:



When we call `fib(5)`, our program returns the sum of `fib(4)` and `fib(3)`. Each of these returns the sum of smaller `fib()` calls, until those calls are `fib(1)` or `fib(0)`, which return a value. Add all of these values together, and that is what `fib(5)` returns. Notice that this method ends up calling itself **14 times**, and that's only asking for the 5th number in the sequence. You can see why calling this method on a larger value, such as 100, might cause your program to crash - the number of calls to the function will increase rapidly, and each sub-call requires your program to return the sum of each sub-call before calculating the total. As a general rule, a recursive function that calls itself more than once becomes inefficient as the levels of recursion increase.

Practice

Consider the number sequence below. Write both an iterative and recursive method that would calculate the nth number in this sequence:

4, 13, 40, 121, 364, 1093

```
//returns the nth number of the sequence above
public int iSequence(int n){









}
```

4, 13, 40, 121, 364, 1093

```
//returns the nth number of the sequence above
public int rSequence(int n){









}
```