A Guided Introduction to Thinking *With* and *About* Object Oriented Programming

1 Introduction to Abstraction

The purpose of learning computer science, especially at the high school level, is not to memorize syntax[1]. Rather, we are working to develop approaches to solving problems. Some of the best tools that we have to solve problems are programming languages.

Almost all programming languages help solve problems. Some languages are better suited to specific problems than other languages. There are many elements of a programming language that help solve problems. These elements can include built-in commands, data types, as well as the structure of the language itself.

When beginning a programming language, it is common to struggle over syntactical features such as commas, semi-colons, or indentation. Although, those elements must be correct for programs to work - they are *minor details* that simply support the true purpose of a programming language - **solving problems**. The structure of the programming language itself determines how we solve problems - namely, how we *de-construct* and think about problems.

When discussing programming languages, it is common to refer to *low level* and *high level* languages. By level, we are not referring to complexity but rather to the level of ***abstraction***. Whereby ***abstraction*** refers to a general class - a broader collection of characteristcics or qualities.

For example, when discussing the color *red*, red does not refer to one specific color of one specific object, but rather refers to a color characteristic shared across objects that English speakers refer to as red.

With programming languages, *abstraction* can be seen in the amount of specificity commonly required and utilized across a program. For example, does each instance of *red* require a specific hex combination for each pixel on a screen to be manually specified or can the programmer simply refer to *red*?

A *low level* programming language such as **C** might require the programmer to be very specific and provide low level specifics such as actual memory bits of the computer. By contrast, a *high level* programming language such as **Processing**, **Python**, or **Java** may very well hide such details from the programmer - allowing the programmer to focus on more *abstract* i.e. 'big picture' ideas.

[1] The specific minutiae of how a language is written e.g. using commas or semicolons.

2 Object Oriented Programming

What is Object Oriented Programming (OOP)? Quite simply stated, OOP is programming structured around *objects* - conglomerations of qualities and characteristics - rather than always detailing specific interactions with the computer's memory. This *object oriented* approach is intended to simplify work for the programmer and improve programming efficiency by closer approximating real world complexity and interactions.

2.1 Generalization

A programmer can work to defining interactions across entire *classes* of objects rather than having to specify interactions for each specific instance. ***But what does that mean????***

Imagine - if every time someone bought a new pair of shoes they had to learn a specific system for tying the laces on that particular pair of shoes. That would be highly inefficient. Rather, we as a society have developed a few methods of tying shoes. Learning to tie one type of shoe readily transfers to tying other types of shoe laces. The laborious process of learning to tie a specific type of shoe lace does not have to be repeated each time. Similarly, imagine that a shoemaker would not want to invent and create a new system for tying shoes each time they made a pair of shoes. Rather, a system would be created for tying shoes and that system could be *inherited* by each pair of shoes that that shoemaker or other shoemakers could use.

2.2 Classes

Imagine further - someone has developed a good underlying design (or *template*) for shoes. In OOP, this can be refered to as a ***class***. In essence, a *class* is like a recipe that can be used as the basic for making many similar, but differentiated, *objects*. The shoemaker could create a *class* (template, recipe, etc.) for a generic shoe design. That class could then be used to mass produce however many shoes of that type (with whatever specified variations) that the shoemaker wants. In terms of OOP, the shoe template would be the *class* used to create individual shoe *objects*, whereby each pair of shoes would be considered an *instance* of **Class** Shoe.

As programmers, and problem solvers (e.g. think about Algebra), the goal is not to spend all our time focused on very specific problems. Rather, we attempt to represent and solve as many problems as possible at once. Sometimes, it may indeed be necessary to specify the behaviors for very particular interactions in very particular contexts. Cf., we may use particular greetings in particular situations (e.g. a specific handshake for a specific friend)- but before we develop specific greetings we develop a generalized greeting such as *Hi!* that can be broadly applicable.

But what does that mean???? In short, as programmers and people, it is not time efficient to develop specific greetings for each individual person

and take that approach to developing greetings for each possible random encounter. Someone could spend the better part of a month just planning out what they might say during a quick trip to the grocery store. Much more time effective is to plan a *generalized* i.e. *abstracted* approach and simply plan on saying "hello" to everyone and plan exceptions from there. For example, plan on saying "hello" to everyone except for people whose name is known - then say "hello `$personsname`." This *abstracted* approach is the essence of OOP.

In Java, or other similarly *object oriented* programming languages, a class might simply be defined by something such as:

```
Class Person{ }
```

However, such a code by itself is useless - because this `Class Person` does not actually attribute any qualities to a person nor does it provide a way to create more Person objects i.e. people.

2.3 Data types

One of the biggest, most important concepts in programming, and perhaps in the modern world, is that there are different types of data. We use different types pf data to answer different types of questions. For example, if someone asks "What type of cookies are those?" - the response "12" is not helpful or useful. Similarly, if someone asks "How many cookies do you have?" and is answered with "chocolate chip" - the response is neither helpful nor useful. A specific question requires a specific type of response.

In order to solve any problem, it is first important to understand what the problem is, the characteristics of the problem, and the likely characteristics of the solution. For example, if we are deciding on the types of cookies for a class party - some of the information we might need are numbers and type of cookies.*Number* and *Type* would be the data types we would use to discuss cookies. More cookie specific data types might include *sugar-type* or *leavening-agent*[2]Leavening agents such as yeast or baking soda make breadrise.). As can be seen with the cookie example - certain data types make more sense based on the environment. In a cookie specific environment *sugar-type* and *leavening-agent* are very useful. However, in the design of cars they would basically be useless.

Data types in programming are those types of data that have been found to be most useful in programming i.e. computer related tasks. Therefore, each programming language has its own built in set of a data types. There is significant overlap - because there is significant overlap in the function of computer languages - also, there is significant overlap because all computer languages must be designed to work on computers i.e. the actual specifics and architecture of processors determines the shape that a programming language takes - e.g. certain data types reflect the amount of space available in the computer's working memory.

[2](

2.3.1 Data Types in Java

In Java, the built-in i.e. hard-coded data types are referred to as *primitives*. The *primitive* data types in Java include:

- **Whole numbers**
  - **byte - a whole number (*integer*) between -128 and 127 (8 bits[a])**
  - **short - an *integer* between -32768 and 32767 (16 bits)**
  - **int - an *integer* between -2,147,483,648 and 2,147,483,647 (32 bits)**
  - **long - an *integer* between 9,223,372,036,854,775,808 and 9,223,372,036,854,755,807 (64 bits)**
- **Floating points (decimal numbers)**
  - **float - a decimal (*floating point*) (32 bits)**
  - **double - a decimal (*floating point*) (64 bits)**
- **Non numerical**
  - **char - a Unicode character e.g. one single letter, one single number, one single type of punctuation, etc.**
  - **boolean - true or false**

[a] A bit is a single binary value of 0 or 1.

See also the official Java documentation.

2.3.2 Strings

Another very common data type, that is **not** a *primitive* is the `String`. A `String` is simply an *array* (i.e. list) of characters. For example, `String name=''John Doe''` contains all the characters and spaces for J-o-h-n-space-D-o-e. Because *Strings* are *objects*, they are able to perform actions (*methods*) not readily available to other primitives.

2.4 Using Data Types

In short, *classes* are recipes that create *objects* which do things (with *methods*) and contain *data*. There is perhaps no absolutely perfect way to organize data. However, some approaches to working with data are more useful than others. At the end of the day, programming, and indeed much of what we do in our lives, revolves around the organization and restructuring of data in new and useful ways.

Java is an *object oriented* programming language - which means that the language is structured to work with data as aspects of *objects*.

Earlier, the example of the empty `Class Person` was given, which was basically an empty recipe with no way of being made and no associated data. In order to do anything meaningful however, we will want to create *objects* that have data and can do things i.e. have *methods*. [3]

2.5 First Worked Example - An *object*-first approach

In Java, basically everything occurs in relationship to some *Class*. Indeed, every Java program **must** have a *main class*.

2.5.1 The main class in Java

```
public class GenericProgramName {
  public static void main(String[] args) {
      // The magic starts here.
  }
}
```

The most important aspects to note (for now) in this example are:

- the curly braces *{}*
- the actual *main method* i.e.
  `public static void main(String[] args) {}`
- the actual *class* name for the overarching program i.e.
  GenericProgramName

In Java, units of code are enclosed in curly braces *{}*. The code between the first "*{*" and last "*}*" belongs together. Of course, this can, and often is, subdivided into further little bits. It may help to think of curly braces *{}* as boxes holding code. As with boxes, larger boxes can contain varied combinations of boxes within boxes - so too is true of curly braces cf.

{{{{}{}}}}

Every Java program needs a *main* method to run. The majority of the program's code may lie outside the *main* method - however, code execution (what is run) begins here. [4]

[3] Note: many introductions to Java would include the typical "Hello World" example as the first program. However, to build a practiced, conceptual understanding of Java as an *object oriented* language - and to reinforce and better practice *object oriented* thinking - this text is written to emphasize an *objects* first approach.

[4] For more discussion of the *main* method in Java, see this Stack Overflow answer.

The `public class` declaration specifies the name of the overarching program *class* - consequently, the name of the file should match this *class declaration* i.e. `public class GenericProgramName` should be inside a file named **GenericProgramName.java**.

2.5.2 Most basic class with object in Java

The most basic program whereby we could create an object would look like this:

```
public class GenericProgramName {
    public static void main(String[] args) {
        Person billy = new Person();
    }
}

class Person {
    public Person() {
    }
}
```

There is the *public class* declaration with the name of the program (matching the file name). This contains the *main* method. In the *main* method all that is accomplished is an empty Person *object* named *billy* is created. Lastly, after the actual `public class GenericProgramName{...}` the `class Person` is created. However, `class Person` is basically empty - there are no data types associated with *Person* objects and no *methods* except for an empty **class constructor**: `public Person(){}`.

2.5.3 Using data types with a class in Java

When creating a *class* **Person**, it may be useful to provide some attributes. `class Person` is created here - there are no mandatory data types. It is at the discretion of the programmer to decide which data types may be most useful for `class Person`.[5]

Defining *Person* objects, the programmer may want to use common person related qualities such as: age, last name, first name, and whether the person is still a student. Consequently, the `class Person` will be created with the corresponding data types.

```
class Person {
    int age;
```

[5]The objects being created, as well as *object oriented* programming in general, are not structured solely to help the computer run best - but rather, the design is to help people utilize the computer most efficiently. The objects, their data, their data labels, their methods - are all approaches used by the programmer to make a program do what she decides. For the computer, everything boils down to 1's and 0's - everything else is to help the programmer use the computer (i.e. program the computer) most effectively.

```
    String lastname;
    String firstname;
    boolean isStudent;
    public Person() {
    }
}
```

However, `class Person` still has the problem that age and name are not defined for Person objects. There are many possible solutions to this problem. Each solution has different advantages and disadvantages. As a programmer, we work not to find **THE** solution but rather to find *the better* or *most practical* solution to a given problem. As far as assigning common qualities to a Person object, we may want to start at the beginning - when an object is created i.e. when it is *instantiated*.

Objects, i.e. instances of a class, are created with the *class constructor*. Previously, the object *billy* was created with the line `Person billy = new Person();` where `new Person()` made a call to the *class constructor*.

2.5.4 Passing information to *constructors*

We will then create a *class constructor* to accept values for first name, last name, age, and whether someone is still a student. Values are passed *inside parentheses (...)*. The class constructor is the mechanism for creating new objects i.e. new *instances* of a class. Consequently, we must specify which data should be passed to a class constructor. The variable names used inside the class constructor are *only recognized **inside** the class constructor*. Consider:

```
  public Person(String fn) {
    this.firstname = fn;
    }
}
```

This class constructor allows us to create a person and specify their first name upon creation i.e. `Person billy = new Person("billy");` creates a new Person object with the value *billy* for the firstname variable. Note the use of `this.firstname` - the **this** is used to ensure that the value specified is assigned to the *instance* being created. Here, the *this* is not completely necessary but is important for developing good coding habits. Further, the "**.**" specifies that the firstname is the value for "firstname" belonging to *this*. Elsewhere, the syntax `billy.lastname` could be used to directly refer to the object *billy*'s lastname value.[6]

In order to pass more values at the creation of each Person *instance*, we will need to specify the values to be passed. Continuing with in the same vein, we can create an additional constructor - one that looks almost identical but

[6] However, directly calling `billy.lastname` is not the preferred practice in Java. Rather, *getters* and *setters* should be used. [These are discussed later.]

accepts more information. In Java, having duplicate constructors (or methods) with the same name that only differ by the data they accept, is called ***overloading***. Our new constructor that accepts first name, last name, and age is:

```
public Person(String fn, String ln, int a){
        this.firstname=fn;
        this.lastname=ln;
        this.age = a;
    }
```

To use this constructor we would pass first name, last name, and age when creating a person object:

```
Person suzy = new Person("Suzy", "Jenkins", 12);
```

Notice: as with "billy" the names "Suzy" and "Jenkins" are in quotes. The quotation marks inform the *compiler* that we are sending a *String* and not a **variable**.

The program thus far might appear so:

```
public class GenericProgramName {
    public static void main(String[] args) {
        Person billy = new Person("billy");
        Person suzy = new Person("Suzy", "Jenkins", 12);
    }
}
class Person {
    int age;
    String lastname;
    String firstname;
    boolean isStudent;

    public Person() {
      //constructor one - no information is passed
    }

    public Person(String fn){
      this.firstname=fn;
      //constructor two - only first name is set
    }

     public Person(String fn, String ln, int a){
        this.firstname=fn;
        this.lastname=ln;
        this.age = a;
        //constructor three - first name, last name, and age are set
```

```
    }
}
```

2.5.5 Doing stuff: Adding methods in Java

We have now created two *objects* - "billy" and "suzy" both *instances* of `class Person`. However, our objects cannot do anything as of yet. For our objects to do stuff, we will want to create *methods* that specify what our Person objects can do.

To create a method, we will need to set access level/modifiers, list the return type, give the method a name, specify any data to be passed (parameters), if any, and actually define what the method does (the method body).

First, we will create the method *sayHi ()*. We will not specify an access modifier. The access level then defaults to *private-package* access. [The differences will be explained later.] Next, we specify the return type which will be *void* because we will not be passing any data back to the program. For the program name, we will choose sayHi - the actual action. We will not be passing any parameters - because we do not need additional information to say "hello." For our actual command in the method body we will use the Java command to print a newline `System.out.println("Hello");`.

In short, the code is structured:
void sayHi () { System.out.println("Hello"); }.

The resulting new method should be inside the Person class - because it is a method that belongs to `class Person`:

```
class Person {
  ...
  //The code has been trimmed to highlight the relevant bits.

  public Person(){
    ...
  }

  void sayHi(){
    System.out.println("Hello");
  }
}
```

The command itself - `System.out.println("Hello");` - (read backwards) consists of the *println()* command which prints a new line containing whatever text is between the parentheses i.e. (*"Hello"*). The println() command itself is a type of *output* i.e. information coming out of the computer instead of going in. These *out* options for **output** belong to the very large number of built-in *System* commands available in Java.

The command itself is like a backwards address going from largest to smallest component - similar to if one were to write their address from largest to smallest unit as *USA.Texas.San Antonio.Navarro.someaddress()*.

Now any Person object can use the method. For any method that belongs to a class, the object name is typed (here "suzy") followed by a dot "**.**" and then the name of the method with any parameters inside the parentheses.[7] The method is then called inside the **main()** method - after the object has been created.[8] The resulting code:

```
public static void main(String[] args) {
  ...
  suzy.sayHi();
}
```

Note this "suzy" not "Suzy" because *suzy* is the variable name used when creating the object i.e. Person suzy = new Person;.

2.5.6 *Need to know*: Using *getters* and *setters*

In Java, objects may have various data associated with them. Hypothetically, the data could be directly accessed by listing the object a dot "**.**" and the variable name i.e. *suzy.age*. Such direct access is not the preferred approach in Java as it can lead to problematic security and other issues in the long term. Consequently, the preferred practice in Java is to use *getter* and *setter* methods whereby **private** data is accessed through **public** methods - guaranteeing that variables are only accessed on a *need to know* basis.

2.5.7 Getting *getters*

We will first create getters to access the values associated with each object. Currently, the possible values for a Person object are age (*an integer*), last name (*a String*), first name (*a String*), and whether an object is still a student (*a boolean* i.e. true or false). To access these data, we have the methods `return` the value. **Returning** a value means that the method gives the value to the program itself. **NOTE:** *nothing is printed by default - printing is a separate command provided for the user **NOT** the program itself.* As with other methods, *getters* specify access, which should be **public**, the return type, which is that variables type, a method name that clarifies what the method does, and a method body, which here quite simply **returns** the variable value

[7] We are not passing any information to the method so our parentheses are empty.
[8] An object that has not yet been created cannot be called.

for the particular object calling it. The methods, as with other class methods, go inside the class itself - typically after the constructor(s). The code itself:[9]

```
public int getAge() {
  return age;
}

public String getLastname() {
  return lastname;
}

public String getFirstname() {
  return firstname;
}

public boolean isIsStudent() {
  return isStudent;
}
```

Hypothetically, we could now run the code `suzy.getAge()` inside our *main()* method. It would run but we would not see anything - because users to not see what values are returned to the program - unless a *method* such as *print* is used. Consequently, we could use **System.out.println()** in combination with **suzy.getAge()** to see the results of the call, i.e., `System.out.println(suzy.getAge());`.

However, this only prints the number by itself. We might want to do something a little more involved - combining mutliple commands.

```
System.out.println("Hello, my name is " +
suzy.getFirstname()  + ". I am " +
suzy.getAge() + " years old.");
```

This command, placed in the *main()* method creates a sentence combining text and output from two of the *getters*.

2.5.8 Setting *setters*

We retrieved information using *getters*. What if we want to change information? *Setters* are perhaps even more important to a programs overall security and utility. Namely, ***setters*** determine whether and *how* information can be changed.

As with other methods, *setters* need an access type, return type, method name, and actual method body.

First, let us focus on changing the object's age value creating a method *setAge ()*. As with *getters*, we will continue to use the **public**

[9] Getters, setters, and constructors can be generated in Netbeans by pressing *alt+insert*.

access modifier (for now). Next, we specify the return type which will be *void* because we will not be passing any data back to the program - we are setting a value for an object, not expecting communication across methods. For the program name, we will choose setAge - a clear description of what we are doing. We will be passing **age** as a parameter as we will want to specify the object's age. This value will be an *int* because we have set the variable to be a whole number. The actual command in the method body, using **this**, a dot **.** and the variable name *age*, we will specify that the particular object's age will be changed to the value supplied for *age*. The equal sign = signifies that one value (`this.age`) is being set to another value (the *age* value passed to the method).

Our resulting method would be:

```
public void setAge (int age) { this.age=age; }.
```

Creating setters, which are commonly placed before *getters* in the *class* definition, for all variables defined so far for `class Person` would produce:

```
public void setAge(int age) {
  this.age = age;
}

public void setLastname(String lastname) {
  this.lastname = lastname;
}

public void setFirstname(String firstname) {
  this.firstname = firstname;
}

public void setIsStudent(boolean isStudent) {
  this.isStudent = isStudent;
}
```

2.6 Questions for review

In writing - answer the following.

1. In your own words, what is *object oriented* programming?
2. Why do we use object oriented programming?
3. How and why do we use a *class*?
4. Write your home address (or a similar imaginary address) as a characteristic of a Java object i.e. start with the largest unit and get progressively smaller: **USA**.*Texas*.*SanAntonio*.etc....
5. Create a class **Animal**.
   The class should have:
   (a) **Five** different *variables* with at least 3 different **data types** e.g. boolean, int, String, float, double, etc.
   (b) Two *different* **class constructors**
   (c) Three *getters*
   (d) Three *setters*
   (e) Three other action methods that use `System.out.*`
6. *Instantiate* three different animals with different characteristics
7. Write and explain three code examples of those animal objects using the methods you created. *You should pass data to at least one of the methods.*
8. Explain, in your own words, Why do we use *getters*? Also - Give a code example.
9. Explain, in your own words, Why do we use **setters**? Also - Give a code example.
10. Explain, in your own words, Why do we create objects?
11. How do we know that we have created a new *object*?
12. What is the code for the *main* method in Java?
13. When and why do we use the main method in Java?

3 Control flow

Objects are *very* important to effective programming. It is beneficial to learn to thing of programming in terms of objects first - in order to work more effectively with an *object oriented* language such as Java. However, many of the tools that make programming powerful precede wide spread use of objects in programming.[10] It is important to understand and affect flow in a programming language, if we are to use computers effectively.

3.1 Order matters

First, it is important to consider that actions are executed in order. Imagine if we are creating and eating a virtual peanut butter sandwich, the following order should succeed:

- open the bread
- take out to slices of bread
- spread peanut butter on the bread
- place peanut butter faces together
- eat sandwich

However, if we were to try:

- spread peanut butter on the bread
- open the bread
- take out to slices of bread
- place peanut butter faces together
- eat sandwich

The program (or *algorithm*) would fail on the first step - because no bread is available.

Consider the following two examples using objects:

Example 1

```
Person suzy = new Person("Suzy", "Jenkins", 12);
suzy.sayHi();}
```

[10] Newer is not always better.

Example 2

```
suzy.sayHi();
Person suzy = new Person("Suzy", "Jenkins", 12);
```

Example 1 succeeds because the *suzy* **object** is created before the method `sayHi()` is called. The second example fails because `suzy.sayHi()` tries to call the *suzy* object's `sayHi()` method before there is a suzy object.

In short, commands are executed in the order written.

Such a *sequential order* is the default order of completion, i.e. one thing then the next.

The other types of order are *selection* and *repetition*.

3.2 Selection structures

3.2.1 if statements

The most common and versatile of selection statements in Java is the **if** statement. The basic structure of the *if* statement is *if* **(some condition)***{What happens if the condition is true;}* For example, the following statement is **true** so it will run once and exit.

```
if(true){
  System.out.println(``This will print once and end.'');
}
```

By contrast, the following code begins has the integer i set to 1, so it executes because i is less than 10, then quits.

```
int i = 1;
  if(i<10){
    System.out.println(``This will print once and end.'');
  }
```

Common **equality and relational operators** for Java are:

- $<$ less than - e.g. `if (a < b){ action();}` executes if a is less than b
- $>$ greater than - e.g. `if (a > b){ action();}` executes if a is greater than b
- $==$ equal to - e.g. `if (a == b){ action();}` executes if a is equivalent to b

- != not equal to - e.g. `if (a != b){ action();}` executes if a is not equal to b
- <= less than or equal to - e.g. `if (a <= b){ action();}` executes if a is less than *or equal to* b
- >= greater than or equal to - e.g. `if (a >= b){ action();}` executes if a is greater than *or equal to* b

3.2.2 if else statements

In Java, **if** is commonly combined with **else**. Perhaps most straight forward is specifying one condition and having a default action if that condition is not true. For example, the following code will print "pass" if the **integer** *grade* is *greater than* or *equal to* 70.

```
if (grade >= 70){
  System.out.println(``pass'');
} else {
  System.out.println(``fail'');
}
```

Indeed, we can connect several *if*, *else if* statements to construct more complex logic. The careful structuring of increasingly complex *if*, *else if* statements can create more nuanced programs.

Consider, for example, a program that will assign a letter grade based on a numeric score:

```
if (grade >= 90) {
  System.out.println(``A'');
} else if (grade >= 80) {
  System.out.println(``B'');
} else if (grade >= 70) {
  System.out.println(``C'');
} else {
  System.out.println(``fail'');
}
```

Notice especially, that the final **else** serves to define what to do if all the preceding criteria are not met. A grade will be greater than or equal to 90, 80, 70 or it is failing.

A common exercise asked of job applicants is to design *if* to determine whether a number is divisible by 3, 5, or both. Only one response should be printed. If a number is divisible by 3 and 5 the specified response (and only specified response) is printed - most failed programs print all three responses or the divisible by 3 or divisible by 5 response.

3.3 Repetition AKA *looping*

One of the great advantages of computers is that they are very good at doing things - repeatedly and very fast. In Java, we use *for*, *while*, and *do while* loops. The most archetypal of loops in Java is the ***while*** loop. Its use and syntax is very straight forward. Basically, as long as some condition is met, the specified action continues. The basic structure of a **while** loop is while (some condition is met){ do this action(); }.

For example, the following `while` loop will print "it's so true" continously because the condition always evaluates to **true**.

```
while(true){
  System.out.println(``it's so true'');
}
```

How then might a *while* loop exit? In the example below, the *while* loop counts up to 10 and then exits because the integer ***i*** is no longer less than 10. Note the *++* - it means that 1 is added to the variable ***i*** each time the loop repeats (i.e. each time it *iterates*).

```
int i =0;
while(i < 10){
  i++;
}
```

3.4 *for* statements

It turns out that it is very useful to have computers *iterate*[11] through various lists of commands. In many programming languages, including Java, the *for* loop is used to count *up* or emphdown and then exit at a specified value. The basic syntax of a for loop is:

for (initial value; condition; increment){ action ;}

For example, the following *for* loop starts at 0, will continue as long as i is less than 10, and the value of i goes up by 1 each *iteration*. During each iteration the value of i will be printed[12].

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

A very common use for *for* loops are as counters - either counting up or down. Here we see a *for* loop that counts down from 100 and stops at 1. **i >0** is used to stop the counter before 0 and ***i***– is used to count down rather than up.

```
for (int i=100; i>0; i--){
  System.out.println(i);
}
```

[11] iterate is much like *count through* or do something in a *step-by-step* manner.
[12] Note the lack of quotation marks around *i*.

3.4.1 Doing things, e.g. math, with *for* loops

Using Java's basic mathematical operators, we can begin *iterating* through many different calculations.

In Java, the basic mathematical operators are:

- **addition - 2+2 == 4**
- **subtraction - 2-2 == 0**
- **multiplication - 6*3 == 18**
- **division - 6/3 == 2**
- ***modulo* or remainder - 248%2 == 0**

These mathematical operators can leverage the values created by *for* loops. For example, the following *loop* will *iterate* through the values of i up to 100 - showing the result of the number i multiplied by 5 i.e. 1 x 5, 2 x 5, 3 x 5, 4 x 5, etc.

```
for (int i = 0; i <= 100; i++){
  System.out.println(i * 5);
}
```

Though this is not the most exciting program but illustrates the underlying utility of a *for* loop. Moreover, *for* loops can become even more useful combined with other *selection* and *repetition* such as *if* statements or additional *for* loops.

In the following example, we can print every even number between 0 and 1000 as even or odd by using the *modulo* % operator. Namely, a number divided by 2 with no remainder is even; otherwise, it is odd.

```
for (int i = 0; i < 1000; i++){
  if(i%2==0){
    System.out.println(i + `` is even'');
  } else {
    System.out.println(i + ``is odd'');
  }
}
```

Similarly, using *nested* ***for*** loops - *for* loops inside other *for* loops can be useful. In the example below, each *for* loop counts up to 12 - thereby creating a 12 x 12 multiplication table. [13]

```
for (int i = 0; i <= 12; i++) {
  for (int j = 0; j <= 12; j++) {
```

[13] *for* loops can use any variable. They commonly use i to refer to *index* variable (from math) and nested *for* loops use the subsequent letters in the alphabet i.e., j, k, l, m, and n.

```
      System.out.println(i + "x" + j + " = " + (i * j));
    }
  }
```

The following table shows the values of i and j for each iteration.

Iteration	i	j	**Iteration**	i	j
it. 1	0	0	*it. 25*	1	11
it. 2	0	1	*it. 26*	1	12
it. 3	0	2	*it. 27*	2	0
it. 4	0	3	*it. 28*	2	1
it. 5	0	4	*it. 29*	2	2
it. 6	0	5	*it. 30*	2	3
it. 7	0	6	*it. 31*	2	4
it. 8	0	7	*it. 32*	2	5
it. 9	0	8	*it. 33*	2	6
it. 10	0	9	*it. 34*	2	7
it. 11	0	10	*it. 35*	2	8
it. 12	0	11	*it. 36*	2	9
it. 13	0	12	*it. 37*	2	10
it. 14	1	0	i*it. 38*	2	11
it. 15	1	1	i*it.39*	2	12
it. 16	1	2	i*it.40*	3	0
it. 17	1	3	i*it. 41*	3	1
it. 18	1	4	i*it. 42*	3	2
it. 19	1	5	i*it. 43*	3	3
it. 20	1	6	i*it. 44*	3	4
it. 21	1	7	i*it. 45*	3	5
it. 22	1	8	i*it. 46*	3	6
it. 23	1	9	i*it. 47*	3	7
it. 24	1	10	*it. 48*	3	8
...	...	...	...	...	...

3.5 Iteration practice

Of course, we as people use computers and for loops to compute faster and more extensively than we as humans can. **However**, it is important that we develop basic proficiency and understanding of how for loops work. Examine the following code and and complete the accompanying table.

Practice 1.

```
for(int i=0;i<5;){
  for(int j=0; j<5; j++){
    i++;
}}
```

Iteration	i	j	**Iteration**	i	j
iteration 1			iteration 5		
iteration 2			iteration 6		
iteration 3			iteration 7		
iteration 4			iteration 8		

Practice 2.

```
for (int i=0; i<3; ){
  for (int j=0; j<3;j++){
    if(j%2==0){i++;
    }
  }
}
```

Iteration	i	j	**Iteration**	i	j
it. 0			*it. 5*		
it. 1			*it. 6*		
it. 2			*it. 7*		
it. 3			*it. 8*		
it. 4			*it. 9*		

Practice 3.

```
for(int i=0; i<5;i++){
  for(int j=0; j<5; j++){
    for(int k=0; k<5;k++){
```

```
        int a= i+j+k;
      }
    }
  }
```

It.#	i	j	k	a	**It.**#	i	j	k	a
it 0					*it 10*				
it 1					*it 11*				
it 2					*it 12*				
it 3					*it 13*				
it 4					*it 14*				
it 5					*it 15*				
it 6					*it 16*				
it 7					*it 17*				
it 8					*it 18*				
it 9					*it 19*				

Practice 4.

```
  int a=0;
  for (int i=0; i<4; i++){
    for (int j=0; j<4; j++){
      if (j%2 ==0){
      a++;
    }
  }
  }
System.out.println(a);
```

It.#	i	j	a	**It.**#	i	j	a
it 0				*it 5*			
it 1				*it 6*			
it 2				*it 7*			
it 3				*it 8*			
it 4				*it 9*			

Practice 5.
Consider the following code and answer the following questions ***before*** filling in the table:

- What will the final output of k be?
- Why?
- Looking at the for loops, how would you justify your assertion?

```
int a = 1;
int k = 0;
for(int i = 0; i < 10; i++) {
  for(int j = 0; j < 3; j++){
    a++;
    if((a % 2) == 0){
    k++;
  }
}
}
  System.out.println(k);
```

It.#	i	j	k	a	**It.#**	i	j	k	a
it 0					*it 16*				
it 1					*it 17*				
it 2					*it 18*				
it 3					*it 19*				
it 4					*it 20*				
it 5					*it 21*				
it 6					*it 22*				
it 7					*it 23*				
it 8					*it 24*				
it 9					*it 25*				
it 10					*it 26*				
it 11					*it 27*				
it 12					*it 28*				
it 13					*it 29*				
it 14					*it 30*				
it 15					***Final***				

Practice 6.

Imagine - you have been given 200 cinder blocks to wall off a flower bed - each measuring 1 ft.

The flower bed must have a minimum width and length of at least 1 ft. [14]

Design an algorithm to:

- Print all possible combinations of width, length, and area
- Print the greatest possible area at the completion of the function

Write your algorithm here. Label and explain the major components.

[14] Assume a rectangular flower bed. Perimeter is the sum of all sides. Area is width multiplied by length.

Practice 7.

Train a leaves New York going 100mph. Train b leaves Florida going 200mph. The distance from Florida to New York is 1000 miles.

- Design an algorithm using *nested* ***for*** loops to determine where the two trains intersect.
- Show the distance that each train is travelling per hour in a table.
- Explain in your own words how your algorithm calculates when the trains pass each other.