Class

A blueprint defining the charactaristics and behaviors of an object of that class type. Class names should be written in CamelCase, starting with a *capital* letter.

```
class MyClass{
    ...
}
```

Each class has two types of variables: class variables and instance variables; class variables point to the same (static) variable across all instances of a class, and instance variables have distinct values that vary from instance to instance.

Class Constructor

Creates an instance of a class (i.e.: calling the Dog constructor creates an instance of Dog). A class can have one or more constructors that build different versions of the same type of object. A constructor with no parameters is called a default constructor; it creates an object with default initial values specified by the programmer. A constructor that takes one or more parameters (i.e.: values in parentheses) is called a *parameterized constructor*. Many languages allow you to have multiple constructors, provided that each constructor takes different types of parameters; these are called overloaded constructors.

```
class Dog{ // class name
    static String unnamed = "I need a name!"; // class variable
    int weight; // instance variable
    String name; // instance variable
    String coatColor; // instance variable

    Dog(){ // default constructor
        this.weight = 0;
        this.name = unnamed;
        this.coatColor = "none";
    }
    Dog(int weight, String color){ // parameterized constructor
        // initialize instance variables
        this.weight = weight; // assign parameter's value to instance
variable
        this.name = unnamed;
        this.coatColor = color;
```

```
    }
    Dog(String dogName, String color){ // overloaded parameterized
constructor
        // initialize instance variables
        this.weight = 0;
        this.name = dogName;
        this.coatColor = color;
    }
}
```

Method

A sort of named procedure associated with a class that performs a predefined action. In the sample code below, *returnType* will either be a data type or if no value need be returned. Like a constructor, a method can have or more parameters.

```
returnType methodName(parameterOne, ..., parameterN){
    ...
    return variableOfReturnType; // no return statement if void
}
```

Most classes will have methods called *getters* and *setters* that get (return) or set the values of its instance variables. Standard getter/setter syntax:

```
class MyClass{
    dataType instanceVariable;
    ...
    void setInstanceVariable(int value){
        this.instanceVariable = value;
    }
    dataType getInstanceVariable(){
        return instanceVariable;
    }
}
```

Structuring code this way is a means of managing how the instance variable is accessed and/or modified.

Parameter

A parenthetical variable in a function or constructor declaration (e.g.: in `int methodOne(int x)`, the parameter is `int x`).

Argument
The actual value of a parameter (e.g.: in `methodOne(5)`, the argument passed as variable `x` is `5`).

For Loop
This is an iterative loop that is widely used. The basic syntax is as follows:

```
for (initialization; termination; increment) {
   // ...
}
```

The *initialization* component is the starting point in your iteration, and your code for this section will typically be `int i = 0`. When we declare and initialize `int i` in the loop like this, we are creating a *temporary variable*that exists only inside this loop for the purposes of iterating through the loop; once we finish iterating and exit (or *break*) the loop, is deleted and can be declared elsewhere in our program.

The *termination* component is the condition which, once met, you would like to exit (or *break*) the loop and proceed to the next line in your code. This is the ending point for your loop, and is typically written as `i < endValue`, where is the variable from the initialization section and is some variable holding the stopping point for your iteration.

The *increment* component is executed each time the end of the code inside the loop's brackets is reached, and should generally be some modification on the initialization variable that brings it closer to the termination variable. This will typically be `i++`. The `++` operator is also called the post-increment operator, and it will increment a variable by after a line executes (for more detail and an example, see the *While* section).

To recap, this sample code:

```
int endOfRange = 4;
for(int i = 0; i < endOfRange; i++){
    System.out.println(i);
}
```

produces this output:

```
0
1
2
3
```

While Loop

This type of loop requires a single boolean condition and continues looping as long as that condition continues to be true. Each time the the end of the loop is reached, it loops back to the top and checks if the condition is still true. If it's true, the loop will run again; if it's false, then the program will skip over the loop and continue executing the rest of the code.

Much like in the *For* section, the code below prints the numbers through . Notice that we are using the *post-increment* operator on :

```
int min = 0;
int max = 4;
while(min < max){
    System.out.println(min++);
}
```

Once , the boolean condition () evaluates to false and the loop is broken. The line `System.out.println(min++);` is a compact way of writing:

```
System.out.println(min);
min = min + 1;
```

Do-While Loop

This is a variation on the *While* loop where the condition is checked at the end of the brackets. Because of this, the content between the brackets is guaranteed to always be executed at least once:

```
do{
    // this will execute once
    // it will execute again each time while(condition) is true
} while(condition);
```

Unlabeled Break

You may recall *break;* from our previous discussion of *Switch Statements*. It will break you out of a loop even if the loop's termination condition still holds true.

Strings and Characters

As we've mentioned previously, a String is a sequence of characters. In the same way that words inside double quotes signify a String, a single letter inside single quotes signifies a character. Each character has an ASCII value associated with it, which is essentially a numeric identifier. The code below creates a char variable with the value , and then prints its ASCII value.

```
char myChar = 'c'; // create char c
System.out.println("The ASCII value of " + myChar + " is: " + (int) myChar);
```

Output:

```
The ASCII value of c is: 99
```

Observe the `(int)` before the variable name in the code above. This is called *explicit casting*, which is a method of representing one thing as another. Putting a data type inside parentheses right before a variable is essentially saying: "The next thing after this should be represented as this data type". Casting only works for certain types of relationships, such as between primitives or objects that inherit from another class.

To break a String down into its component characters, you can use the String.toCharArray method. For example, this code:

```
String myString = "This is String example.";
char[] myCharArray = myString.toCharArray();
for(int i = 0; i < myString.length(); i++){
    // Print each sequential character on the same line
    System.out.print(myCharArray[i]);
}
// Print a newline
System.out.println();
```

produces this output:

```
This is String example.
```

Notice that we were able to simulate printing *myString* by instead printing each individual character in the character array, *myCharArray*, created from *myString*.

Data Structures

A way of organizing data that enables efficient storage, retrieval, and use.

Arrays

A type of data structure that stores elements of the same type (generally). It's important to note that you'll often see arrays referred to as in documentation, but the variable names you use when coding should be descriptive and begin with *lowercase* letters.

You can think of an array, , of size as a contiguous block of cells sequentially indexed from to which serve as containers for elements of the array's declared data type. To store an element, , in some index of array , use the syntax `A[i]` and treat it as you would any other variable (i.e., `A[i] = value;`). For example, the following code:

```
// the number of elements we want to hold
final int _arraySize = 4;

// our array declaration
String[] stringArray = new String[_arraySize];

for(int i = 0; i < _arraySize; i++) {
    // assign value to index i
    stringArray[i] = "This is stored in index " + i;

    // print value saved in index i
    System.out.println(stringArray[i]);
}
```

saves and then prints the values listed below in their respective indices of :

```
This is stored in index 0
This is stored in index 1
This is stored in index 2
This is stored in index 3
```

Most languages also have a *method*, *attribute*, or *member* that allows you to retrieve the size of an array. In Java, arrays have a attribute; in other words, you can get the length of some array, `arrayName`, by using the `arrayName.length` syntax.

Note: The *final* keyword used in the code above is a means of protecting the variable's value by locking it to its initialized value. Any attempt to reassign (overwrite) the value of a *final* variable will generate an error.

Note on Arrays in C++

If you want to create an array whose size is unknown at compile time (i.e., being read as input), you need to create a pointer to whatever data type you'll be declaring your array as (e.g., *char*, *int*, *double*, etc.). Then you must use the new operator to set aside the space you need for your array. The example below shows how to create an array of type *DataType* and unknown size *n* that is read from stdin.

```
// array size
int n;
cin >> n;

// create array of unknown size n
DataType* arrayName = new DataType[n];
```

Java Maps

Map is an interface that provides a blueprint for data structures that take pairs and map keys to their associated values (it's important to note that both the and the must be Objects and *not* primitives). The *implementation* is done by *implementing classes* such as HashMap or LinkedHashMap. Consider the following code:

```
// Declare a String to String map
Map<String, String> myMap;

// Initialize it as a new String to String HashMap
myMap = new HashMap<String, String>();
```

```
// Change myMap to be a new (completely different) String to String
LinkedHashMap instead
myMap = new LinkedHashMap<String, String>();
```

Here are a few Map methods you will find helpful for this challenge:

- *containsKey(Object key)*: Returns true if the map contains a mapping for ; returns false if there is no such mapping.
- *get(Object key)*: Returns the value to which the is mapped; returns *null* if there is no such mapping.
- *put(K key, V value)*: Adds the (*Key, Value*) mapping to the Map; if the is already in the map, the is overwritten.

Example (Java)

The code below:

```
// Create a Map of String Keys to String Values, implemented by the
HashMap class
Map<String,String> myMap = new HashMap<String,String>();

// Adds ("Hi","Bye") mapping to myMap
myMap.put("Hi", "Bye");

// Print the Value mapped to from "Hi"
System.out.println(myMap.get("Hi"));

// Replaces "Bye" mapping from "Hi" with "Bye!"
myMap.put("Hi", "Bye!");

// Print the Value mapped to from "Hi"
System.out.println(myMap.get("Hi"));
```

produces the following output:

```
Bye
Bye!
```

It is not necessary to declare *myMap* as type *Map*; you can certainly declare it as a *HashMap* (the instantiated type).

Recursion

This is an algorithmic concept that involves splitting a problem into two parts: a *base case* and a *recursive case*. The problem is divided into smaller subproblems which are then solved recursively until such time as they are small enough and meet some base case; once the base case is met, the solutions for each subproblem are combined and their result is the answer to the entire problem. If the base case is not met, the function's recursive case calls the function again with modified values. The code must be structured in such a way that the base case is reachable after some number of iterations, meaning that each subsequent modified value should bring you closer and closer to the base case; otherwise, you'll be stuck in the dreaded infinite loop!

Example

The code below produces the multiple of two numbers by combining addition and recursion:

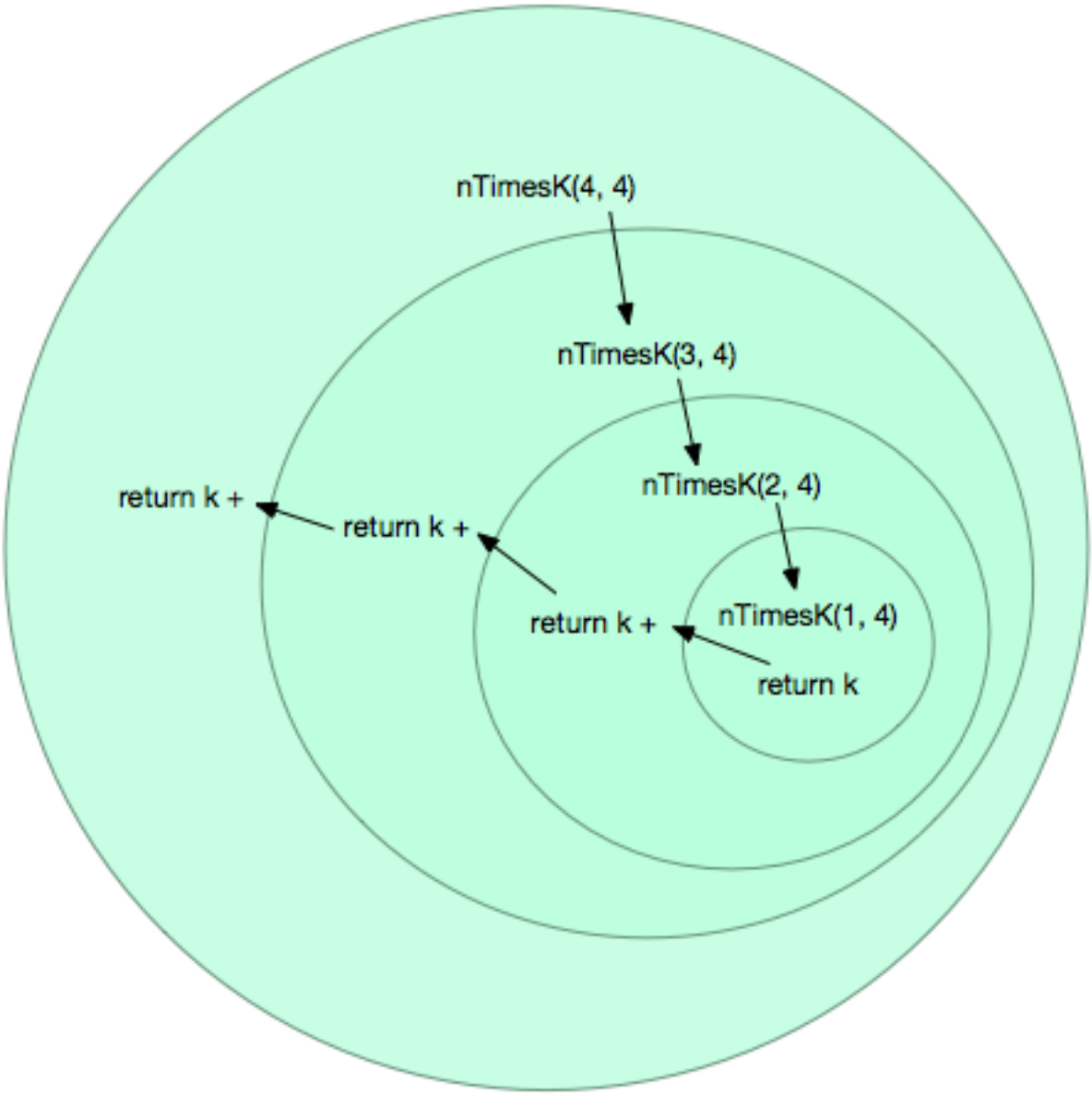
```
// Multiply 'n' by 'k' using addition:
private static int nTimesK(int n, int k) {
    System.out.println("n: " + n);
    // Recursive Case
    if(n > 1) {
        return k + nTimesK(n - 1, k);
    }
    // Base Case n = 1
```

```
    else {
        return k;
    }
}
public static void main(String[] args) {
    int result = nTimesK(4, 4);
    System.out.println("Result: " + result);
}
```

When executed, this code prints:

```
n: 4
n: 3
n: 2
n: 1
Result: 16
```

The diagram below depicts the execution of the code above. Each call to is represented by a bubble, and each new recursive call bubble is stacked inside and on top of the bubble that was responsible for calling it. The function recursively calls itself using reduced values until it reaches the base case (). Once it reaches the base case, it passes back the base case's return value () to the bubble that called it and continues passing back `k +` the previously returned value until the final result (i.e.: the multiplication by addition result of) is returned.



Once the code hits the base case in the bubble, it returns (which is) to the bubble.
Then the bubble returns , which is , to the bubble.
Then the bubble returns , which is , to the bubble.
Then the bubble returns , which is , to the first line in *main* as the result for , which assigns to the variable.

Radix (Base)

The number of digits that can be used to represent a number in a positional number system. The decimal number system (base-) has digits (); the binary (base-) number system has digits ().
We think in terms of base-, because the decimal number system is the only one many people need in everyday life. For situations where there is a need to specify a number's radix, number having radix should be written as .

Binary to Decimal Conversion

In the same way that , a binary number having digits in the form of can be converted to decimal by summing the result for each where , is the most significant bit, and is the least significant bit.
For example: is evaluated as

Decimal to Binary Conversion

To convert an integer from decimal to binary, repeatedly divide your base- number, , by . The dividend at each step should be the result of the integer division at each step . The remainder at each step of division is a single digit of the binary equivalent of ; if you then read each remainder in order from the last remainder to the first (demonstrated below), you have the entire binary number.
For example: . After performing the steps outlined in the above paragraph, the remainders form (the binary equivalent of) when read from the bottom up:

This can be expressed in pseudocode as:

```
while(n > 0):
    remainder = n%2;
    n = n/2;
    Insert remainder to front of a list or push onto a stack

Print list or stack
```

Many languages have built-in functions for converting numbers from decimal to binary. To convert an integer, , from decimal to a String of binary numbers in Java, you can use the *Integer.toBinaryString(n)* function.
Note: The algorithm discussed here is for converting integers; converting fractional numbers is a similar (but different) process.

Radix (Base)

The number of digits that can be used to represent a number in a positional number system. The decimal number system (base-) has digits (); the binary (base-) number system has digits ().
We think in terms of base-, because the decimal number system is the only one many people need in everyday life. For situations where there is a need to specify a number's radix, number having radix should be written as .

Binary to Decimal Conversion

In the same way that , a binary number having digits in the form of can be converted to decimal by summing the result for each where , is the most significant bit, and is the least significant bit.
For example: is evaluated as

Decimal to Binary Conversion

To convert an integer from decimal to binary, repeatedly divide your base- number, , by . The dividend at each step should be the result of the integer division at each step . The remainder at each step of division is a single digit of the binary equivalent of ; if you then read each remainder in order from the last remainder to the first (demonstrated below), you have the entire binary number.
For example: . After performing the steps outlined in the above paragraph, the remainders form (the binary equivalent of) when read from the bottom up:

This can be expressed in pseudocode as:

```
while(n > 0):
    remainder = n%2;
    n = n/2;
    Insert remainder to front of a list or push onto a stack

Print list or stack
```

Many languages have built-in functions for converting numbers from decimal to binary. To convert an integer, , from decimal to a String of binary numbers in Java, you can use the *Integer.toBinaryString(n)* function.

2D Arrays

Also known as *multidimensional* arrays, they are very similar to the regular *1D Array* data structure we've already discussed.

Consider the following code:

```
int rowSize = 2;
int colSize = 4;
int[][] myArray = new int[rowSize][colSize];
```

This creates a matrix where each element, , can be graphically represented as follows:

```
(0, 0) (0, 1) (0, 2) (0, 3)
(1, 0) (1, 1) (1, 2) (1, 3)
```

You may find it helpful to think of these elements in terms of real-world structures such as the cells in a spreadsheet table.

To fill the array's cells with values, you can use a *nested loop*. The outer loop represents the matrix's *rows* and uses as its variable, and the inner loop represents the matrix's *columns* and uses as its variable. The code below assigns the value of to each element in the *2D* array we declared previously:

```
int count = 0;

for(int i = 0; i < rowSize; i++) {

    for(int j = 0; j < colSize; j++, count++) {
        myArray[i][j] = count;
    }
}
```

If we print the contents of each row:

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

    // print the row of space-separated values
    for(int j = 0; j < colSize; j++) {
        System.out.print(myArray[i][j] + " ");
    }
    // end of row is reached, print newline
    System.out.println();
}
```

we'll see the following output:

```
0 1 2 3
4 5 6 7
```

Inheritance

This allows you to establish a hierarchy for your classes. A class that *inherits* from some other class (referred to as a *superclass*) is called a *subclass*. While a subclass inherits methods and behaviors from a superclass, it can also declare new fields and methods (as well as *override* superclass methods).

Subclass

A subclass is defined with the *extends* keyword. For example, the syntax `ClassB extends ClassA` establishes *ClassB* as a subclass of of *ClassA*. Java only supports *single inheritance*, meaning a subclass cannot extend more than one superclass.

Synonymous terms: derived class, extended class, child class.

Subclass Constructors
Because a constructor initializes an instance of a class, they are *never* inherited; however, the subclass must call a superclass constructor as it is an extension of a superclass object. This can be done in either of the two ways shown below.

Consider the following class:
```
class MySuperclass{
    // superclass instance variable:
    String myString;

    // superclass default (empty) constructor:
    MySuperclass(){}

    // superclass parameterized constructor:
    MySuperclass(String myString){
        // initialize instance variable
        this.myString = myString;
    }
}
```
1) The subclass makes an *explicit* call to the superclass' parameterized constructor (i.e.: it calls `super(...);`):
```
class MySubclass extends MySuperclass{
    // subclass constructor:
    MySubclass(String myString){
        // explicit call to superclass constructor:
        super(myString);
    }
}
```
2) The subclass makes an *implicit* call to the superclass' default constructor (i.e.: a behind-the-scenes call to `super();` happens automatically):
```
class MySubclass extends MySuperclass{
    MySubclass(String myString){
        // behind-the-scenes implicit call to superclass' default
constructor happens

        // subclass can now initialize superclass instance variable:
        this.myString = myString;
    }
}
```
In the second example above, observe that we are initializing a field (myString) that isn't even declared in that class; the reason why this works is because it's inherited from *MySuperclass* and therefore can be accessed with the this keyword.

Note: If a superclass does not have a default constructor, any subclasses extending it *must* make an explicit call to one of the superclass' parameterized constructors.

Overriding Methods
When overriding a method, it is best practice to precede the method with the `@Override` annotation. This signifies to both the reader and the compiler that this method is overriding an

inherited method, and will also help you check your work by generating a compiler error if no such method exists in the superclass. Method overriding is demonstrated in the example below.

Example

Let's say a not-for-profit organization has an *Employee* class, and each instance of the *Employee* class contains the name and salary for an employee. Then they decide that they need a similar-yet-different way to store information about volunteers, so they decide to write a *Volunteer* class that inherits from *Employee*. This is beneficial because any fields and methods added to *Employee* will also be accessible to *Volunteer*.

Take some time to review the code below. Observe that the *Volunteer* class calls the superclass' *getName* method, but overrides its *print* method.

```
import java.util.Locale;
import java.text.NumberFormat;

class Employee {
    // instance variables:
    protected String name;
    private int salary;

    /** Parameterized Constructor
    *@param name The volunteer's name. **/
    Employee(String name){
      // use name param to initialize instance variable:
        this.name = name;
    }

    /** @return The name instance variable. **/
    String getName(){
        return name;
    }

    /** @param salary The integer to set as the salary instance
variable. **/
    void setSalary(int salary){
        this.salary = salary;
    }

    /** @return The salary instance variable. **/
    int getSalary(){
        return salary;
    }

    /** Print information about an instance of Employee. **/
    void print(){
        if(this.salary == 0){
            System.err.println("Error: No salary set for " + this.name
                + "; please set salary and try again.\n");
        }
        else{ // Print employee information
            // Formatter for salary that will add commas between zeroes:
            NumberFormat salaryFormat =
NumberFormat.getNumberInstance(Locale.US);
```

```
            System.out.println("Employee Name: " + this.name
                + "\nSalary: " + salaryFormat.format(this.salary) +
"\n");
        }
    }
}

class Volunteer extends Employee{
     // instance variable:
    int hours;

    /** Parameterized Constructor
    *      @param name The volunteer's name. **/
    Volunteer(String name){
     // explicit call to superclass' parameterized constructor
        super(name);
    }

    /** @param Set the hours instance variable. **/
    void setHours(int hours){
        this.hours = hours;
    }

    /** @return The hours instance variable **/
    int getHours(){
        return hours;
    }

    @Override
    /** Overrides the superclass' print method and prints information
about an instance of Volunteer. **/
    void print(){
        System.out.println("Volunteer Name: " + this.getName()
            + "\nHours: " + this.getHours());
    }
}
```

The following code:

```
Employee employee = new Employee("Erica");
employee.print();
employee.setSalary(60000);
employee.print();

Volunteer volunteer = new Volunteer("Anna");
volunteer.setHours(20);
volunteer.print();
```

produces this output:

```
Error: No salary set for Erica; please set salary and try again.

Employee Name: Erica
Salary: 60,000

Volunteer Name: Anna
Hours: 20
```

Abstraction

This is an essential feature of object-oriented programming. In essense, it's the separation between *what* a class does and *how* it's accomplished.
One real world example of this concept is a snack machine, where you give the machine money, make a selection, and the machine dispenses the snack. The only thing that matters is *what* the machine does (i.e.: dispenses the selected snack); you can easily buy a snack from any number of snack machines without knowing *how* the machine's internals are designed (i.e.: the implementation details).

Abstract Class

This type of class can have *abstract methods* as well as *defined methods*, but it cannot be instantiated (meaning you cannot create a *new* instance of it). To use an abstract class, you must create and instantiate a *subclass* that *extends* the abstract class. Any *abstract methods* declared in an abstract class must be implemented by its subclasses (unless the subclass is also abstract).
The Java code below demonstrates an abstract *Canine* class and *2* of its canine breed subclasses, *KleeKai* and *SiberianHusky*:

```
/** Superclass **/
abstract class Canine{
    // instance variables
    String name;
    String color;
    String gender;
    int age;

    /** Parameterized Constructor
    *   @param name Dog's name
    *   @param color Dog's color
    *   @param age Dog's age
    *   @param mF Dog's gender ('M' for male, 'F' for female)
    **/
    Canine(String name, String color, int age, char mF){
        this.name = name;
        this.color = color;
        this.age = age;
        this.gender = (mF == 'M') ? "Male " : "Female ";
    }

    /** Abstract method declaration
    *   @return Implementations should return a string describing the
breed **/
    abstract String getBreed();

    /** Defined method **/
    void printInfo(){
     // print information about the dog:
        System.out.println(name + " is " + ((age%10 == 8)? "an " : "a ")
+ age + " year old "
           + gender + getBreed() + " with a " + color + " coat.");
        // note: the '(age%10 == 8)' conditional ensures grammatical
correctness if dog is 8 or 18; dogs do not live longer than this.
    }
}

//** Subclass of Canine **/
class KleeKai extends Canine{
```

```
     /** Parameterized Constuctor **/
    KleeKai(String name, String color, int age, char mF){
        super(name, color, age, mF);
    }

    /** Abstract method implementation
    *   @return "Klee Kai" **/
    String getBreed(){ // abstract method implementation
        return "Klee Kai";
    }
}

/** Subclass of Canine **/
class SiberianHusky extends Canine{
     /** Parameterized Constuctor **/
    SiberianHusky(String name, String color, int age, char mF){ //
Constructor
        super(name, color, age, mF);
    }

    /** Abstract method implementation
    *   @return "Siberian Husky" **/
    String getBreed(){ // abstract method implementation
        return "Siberian Husky";
    }
}
```

The *Canine* class has *1* abstract method, *abstract void getBreed()*, and *1* defined method, *void printInfo()*. Because an abstract class is not fully defined, attempting to instantiate it like so:

```
Canine myPuppy = new Canine("Lilah", "Grey/White", 5, 'F');
```

results in `error: Canine is abstract; cannot be instantiated`. This type of class is only meant to serve as a base or blueprint for connecting the subclasses that inherit (extend) it. While we can't instantiate *Canine*, we can instantiate its subclasses, *KleeKai* and *SiberianHusky*. This code:

```
Canine c = new KleeKai("Lilah", "Grey/White", 5, 'F');
Canine d = new SiberianHusky("Alaska", "Grey/Black/White", 16, 'F');
c.printInfo();
d.printInfo();
```

executes and produces this output:

```
Lilah is a 5 year old Female Klee Kai with a Grey/White coat.
Alaska is a 16 year old Female Siberian Husky with a Grey/Black/White
coat.
```

because `c` and `d` are polymorphic references objects of *Canine*'s subclasses.

Scope

This term refers to the region of the program to which an identifier applies. While it is not good practice, you can declare multiple variables within a program that use the same identifier *as long as* the identifiers have differing scopes; some exceptions to this are:

1. A constructor or method parameter will often have the same name as a class field it's intended to initialize or modify.
2. It is customary to use `i` as the condition variable in a for-loop (and, in cases of nested for-loops, to use `j` as the condition variable for the inner loop).
3.

Note: When dealing with a class variable (field), it's always best to explicitly refer to it using the *this* keyword. For example:

```
class MyClass{
    private int myInt;

    public MyClass(int myInt){
      this.myInt = myInt;
    }
}
```

Even though there is a `myInt` field in the class, the constructor has a completely different `myInt` parameter. The field (`this.myInt`) is then assigned the value of the parameter, so any argument passed as that parameter will initialize the `myInt` field in the class.

Still confused? Take some time to review the *Scope* class below and understand why variables with the same name will be of different types (and produce different outputs) at various points in the program:

```
public class Scope{
    boolean b = true; // b1 has scope of entire class
    int x = 88; // x1 has scope of entire class

    Scope(){
        double d = 9.0;
        example(d);
        classVariable();
    }

    void example(double x){ // parameter x2 has scope of this method
        System.out.println("----- example(double x):\n"
                +     "Initial value of Local Variable `x`: " + x +
"\n");

        x = 4.4; // reassign value of local variable x2

        System.out.println("New value of Local Variable `x`: " + x +
"\n");

        for(int b = 0; b < 4; b++){ // b2 has scope of this loop
            int i = b + 4; // begin scope of int i
            System.out.println(  "For Loop 1 in example(double x):\n"
                +     "Local Variable `b` (local to loop): " + b + "\n"
                +     "Local Variable `i` (local to loop): " + i + "\n"
                +     "Local Variable `x` (method parameter): " + x +
"\n");
        } // end the scope of int b2; end scope of int i
```

```
        for(int b = 0; b < 4; b++){
            x = b;
            System.out.println(  "For Loop 2 in example(double x):\n"
                +     "Local Variable `b` (local to loop): " + b + "\n"
                +     "Local Variable `x` (method parameter): " + x +
"\n");
        } // end of the scope of this version of int b

        System.out.println("Local Variable `x` after Loop 2: " + x +
"\n");

    } // end scope of double x2

    void classVariable(){
        System.out.println("----- classVariable():\n"
                +     "Instance Variable `b`: " + b + "\n"
                +     "Instance Variable `x`: " + x);
    }

    public static void main(String[] args){
        Scope s = new Scope();
    }
} // end of boolean b's scope; end of int x's scope
```

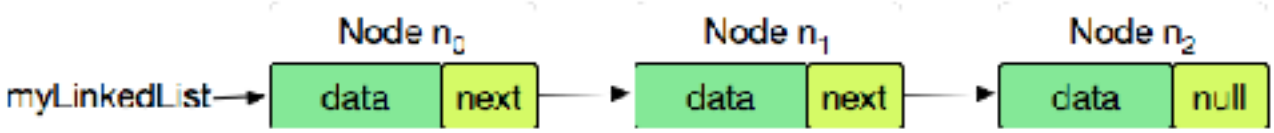
which produces this output:

```
----- example(double x):
Initial value of Local Variable `x`: 9.0

New value of Local Variable `x`: 4.4

For Loop 1 in example(double x):
Local Variable `b` (local to loop): 0
Local Variable `i` (local to loop): 4
Local Variable `x` (method parameter): 4.4

For Loop 1 in example(double x):
Local Variable `b` (local to loop): 1
Local Variable `i` (local to loop): 5
Local Variable `x` (method parameter): 4.4

For Loop 1 in example(double x):
Local Variable `b` (local to loop): 2
Local Variable `i` (local to loop): 6
Local Variable `x` (method parameter): 4.4

For Loop 1 in example(double x):
Local Variable `b` (local to loop): 3
Local Variable `i` (local to loop): 7
Local Variable `x` (method parameter): 4.4

For Loop 2 in example(double x):
Local Variable `b` (local to loop): 0
Local Variable `x` (method parameter): 0.0
```

```
For Loop 2 in example(double x):
Local Variable `b` (local to loop): 1
Local Variable `x` (method parameter): 1.0

For Loop 2 in example(double x):
Local Variable `b` (local to loop): 2
Local Variable `x` (method parameter): 2.0

For Loop 2 in example(double x):
Local Variable `b` (local to loop): 3
Local Variable `x` (method parameter): 3.0

Local Variable `x` after Loop 2: 3.0

----- classVariable():
Instance Variable `b`: true
Instance Variable `x`: 88
```

Linked List

A singly linked list is a data structure having a list of elements where each element has a reference pointing to the next element in the list. Its elements are generally referred to as *nodes*; each node has a *data* field containing a data value and a *next* field pointing to the next element in the list (or `null` if it is the last element in the list). The diagram below depicts a linked list of length :



The sample code below demonstrates how to create a *LinkedList* of Strings, and some of the operations that can be performed on it.

```
LinkedList<String> myLinkedList = new LinkedList<String>();

// Add a node with data="First" to back of the (empty) list
myLinkedList.add("First");

// Add a node with data="Second" to the back of the list
myLinkedList.add("Second");

// Insert a node with data="Third" at front of the list
myLinkedList.addFirst("Third");

// Insert a node with data="Fourth" at back of the list
myLinkedList.addLast("Fourth");

// Insert a node with data="Fifth" at index 2
myLinkedList.add(2, "Fifth");

// Print the list: [Third, First, Fifth, Second, Fourth]
System.out.println(myLinkedList);

// Print the value at list index 2:
System.out.println(myLinkedList.get(2));

// Empty the list
myLinkedList.clear();
```

```
// Print the newly emptied list: []
System.out.println(myLinkedList);

// Adds a node with data="Sixth" to back of the (empty) list
myLinkedList.add("Sixth");
System.out.println(myLinkedList); // print the list: [Sixth]
```
The above code produces the following output:
```
[Third, First, Fifth, Second, Fourth]
Fifth
[]
[Sixth]
```

String to Integer
Many languages offer some functionality for parsing an integer value from a string token. Here are a few languages and their respective integer parsing functions:

- Java: *Integer.parseInt(token)*
- C++: *stoi(token)*
- Python: *int(token)*
- Ruby: *Integer(token)*
-

Each of these functions will raise an error (i.e.: throw an exception) when the argument passed as the *token* parameter cannot be converted to an integer. Most languages have built-in exception handling methods such as *try-catch* or *begin-rescue* that enable you to code specific behaviors to handle execution issues.

Exceptions
If your code attempts to perform an action that cannot be completed, the flow of control is halted and an exception is *thrown*. This means that an *Exception* object is created as a response to this unusual condition. The control flow is then transferred (or handed off) to an *exception handler*. By anticipating and writing handlers for exceptional conditions in your program's logic, you can resolve the issue that raised the exception and your program can continue executing. A program "crash" is generally the result of an unhandled exception.

Exception handling is expensive, meaning it takes a lot of behind-the-scenes work for your program to stop everything and figure out how an exceptional scenario should be handled. Under normal circumstances, you can avoid the need to handle many kinds of exceptions by anticipating and coding for all possible scenarios.

Managing Exceptions: try
The *try* block is like a staging area for potentially error-raising code. If your program is unable to execute the code inside a *try* block, it throws an exception and tries to find an *exception handler* to salvage the situation. The syntax is as follows:
```
try{
    // write exception-throwing code here
}
```

Managing Exceptions: catch

A *catch* block should always immediately follow a *try* block, and looks like a sort of mini-function. It must take some type of exception (either Exception or one of its subclasses) as a parameter, and it looks like this:

```
catch(Exception e){
    // write exception handling logic here
}
```

Part of writing good code is knowing, circumventing, and anticipating exactly what type of exceptions your instructions might throw, but if your parameter is of type *Exception*, it will catch any exception that is a subclass of *Exception*. If the code in your *try* block has the potential to throw more than one type of exception, you can have multiple *catch* blocks to catch each type of anticipated exception.

Managing Exceptions: finally

The *finally* block immediately follows the *catch* block, and will always execute when the *try* block exits—regardless of whether or not an exception is thrown. The *finally* block is optional, and generally used for cleanup code.

Managing Exceptions: try with resources

This is useful when you are using a resource that must be opened/closed (anything that implements *java.lang.AutoCloseable* or *java.io.Closeable*), such as a Scanner or BufferedReader. While you will likely never need try-with-resources blocks for our challenges, it's still worth knowing about.

```
try(Scanner scan = new Scanner();){
    // use scanner to do something that potentially throws an exception
}
```

Example

The code below demonstrates how the *try*, *catch*, and *finally* blocks handle errors in bad code:

```
import java.util.*;

class Solution{
    LinkedList<String> list;
    int[] intArray = new int[4];

    // For testing Null Pointer Exception
    Solution(){
        this.list = null;
    }

    // For testing Index Out of Bounds
    Solution(String str){
        this.list = new LinkedList<String>();
        list.add(str);
    }

    void exceptionDemo(int i, String str){

        try{
            // throws ArrayIndexOutOfBoundsException if index >
intArray.length
            int myInt = intArray[i];
```

```
            // throws a NullPointerException if 'list' doesn't point to
an actual list object
            list.indexOf(str);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.err.println( "The following index is out-of-bounds: "
+ e.getMessage() );
        }
        catch(NullPointerException e){
            System.err.println( "Oh no! You tried to perform an
operation on an object whose value is null!" );
        }
        finally{
            System.out.println("This is printing regardless of whether
or not the program finishes executing.");
        }
        System.out.println("The program was able to continue execution!
\n");
    }

    public static void main(String[] args) {
        // creates a Solution object whose 'list' instance variable
points to null:
        Solution sNullList = new Solution();
        // attempt to access an element of the null list, throws
Exception
        sNullList.exceptionDemo(1, "x");

        // creates a Solution object whose 'list' instance variable
points to a list containing 1 element ("x"):
        Solution sNotNullList = new Solution("x");
        // attempt to access an invalid index of 'intArray' instance
variable, throws Exception
        sNotNullList.exceptionDemo(100, "x");
    }
}
```

Produces the following output:

```
Oh no! You tried to perform an operation on an object whose value is
null!
This is printing regardless of whether or not the program finishes
executing.
The program was able to continue execution!

The following index is out-of-bounds: 100
This is printing regardless of whether or not the program finishes
executing.
The program was able to continue execution!
```

If you were to comment out the first catch block:

```
catch(ArrayIndexOutOfBoundsException e){
    System.err.println( "The following index is out-of-bounds: " +
e.getMessage() );
}
```

Your output would be:

```
Oh no! You tried to perform an operation on an object whose value is
null!
```

```
This is printing regardless of whether or not the program finishes
executing.
The program was able to continue execution!

This is printing regardless of whether or not the program finishes
executing.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
    at Solution.exceptionDemo(Solution.java:22)
    at Solution.main(Solution.java:48)
```

If you were to comment out the second catch block:

```
catch(NullPointerException e){
    System.err.println( "Oh no! You tried to perform an operation on an
object whose value is null!" );
}
```

Your output would be:

```
This is printing regardless of whether or not the program finishes
executing.
Exception in thread "main" java.lang.NullPointerException
    at Solution.exceptionDemo(Solution.java:25)
    at Solution.main(Solution.java:43)
```

Propagating Exceptions

If an exception is not caught by the method that threw it, the program's control is transferred back (*propagated*) to the calling method (i.e.: whatever called the method that threw the exception). This can be good if you have designed your program to handle exceptions at a high level, but bad if you never write code to catch the exception in the calling methods that the exception is being propagated to. The *example* method below throws an exception of type *Exception*, which propagates back to the calling method (*main*), where a *catch* block catches it and prints a message:

```
class PropagatedException {

    void example() throws Exception{
        throw new Exception("This exception will always be thrown.");
    }

    public static void main(String[] args) {
        PropagatedException p = new PropagatedException();
        try{
            p.example();
        }
        catch(Exception e){
            System.err.println( e.getClass().getSimpleName() + ": " +
e.getMessage() );
        }
    }
}
```

The above code throws a `java.lang.Exception` and prints:

```
Exception: This exception will always be thrown.
```

Stacks

A stack is a data structure that uses a principle called *Last-In-First-Out* (*LIFO*), meaning that the last object added to the stack must be the first object removed from it.

At minimum, any stack, *s*, should be able to perform the following three operations:

- *Peek*: Return the object at the top of the stack (without removing it).

- *Push*: Add an object passed as an argument to the top of the stack.
- *Pop*: Remove the object at the top of the stack and return it.

The *java.util* package has a *Stack* class that implements these methods; check out the documentation (linked above) on the *peek()*, *push(object)*, and *pop()* methods.

Queues

A queue is a data structure that uses a principle called *First-In-First-Out* (*FIFO*), meaning that the first object added to the queue must be the first object removed from it. You can analogize this to a checkout line at a store where the line only moves forward when the person at the head of it has been helped, and each person in the line is directly behind the person whose arrival immediately preceded theirs.

At minimum, any queue, *q*, should be able to perform the following two operations:

- *Enqueue*: Add an object to the back of the line.
- *Dequeue*: Remove the object at the head of the line and return it; the element that was previously second in line is now at the head of the line.

The *java.util* package has a *Queue* interface that can be implemented by a number of classes, including *LinkedList*. Much like abstract classes, interfaces cannot be instantiated so we must declare a variable of type *Queue* and initialize it to reference a new *LinkedList* object. Check out the documentation (linked above) on the *add(object)* (enqueue) and *remove()* (dequeue) methods. You'll learn more about interfaces tomorrow!

Interface

Recall that *abstraction* is the separation between *what* something does and *how* it's accomplished. An *interface* is a collection of abstract methods and constants that form a common set of base rules/specifications for those classes that *implement* it. Much like an abstract class, an interface cannot be instantiated and *must* be implemented by a class.

Example

Consider a polygon. How do we interact with polygons? What properties are common among polygons? Take some time to review the simple *Polygon* interface below, as well as the classes that implement it.

```
/**
*   This is a collection of methods we expect and require a polygon to
have
**/
interface Polygon{
    /** @return The number of sides of the Polygon **/
    int getNumberOfSides();
    /** @return The perimeter of the Polygon **/
    double getPerimeter();
}

class Triangle implements Polygon {
    private static int numberOfSides = 3;
    private double side1;
    private double side2;
    private double side3;

    public Triangle(double side1, double side2, double side3){
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
```

```
    }

    public int getNumberOfSides(){
        return numberOfSides;
    }

    public double getPerimeter(){
        return side1 + side2 + side3;
    }
}

class Rectangle implements Polygon {
    private static int numberOfSides = 4;
    private double side1;
    private double side2;

    public Rectangle(double side1, double side2){
        this.side1 = side1;
        this.side2 = side2;
    }

    public int getNumberOfSides(){
        return numberOfSides;
    }

    public double getPerimeter(){
        return side1 + side1 + side2 + side2;
    }
}

/**
*    This inherits the properties and methods of its superclass,
Rectangle.
**/
class Square extends Rectangle implements Polygon {
    public Square(double side){
        super(side, side);
    }
}

class Solution{
    public static void print(Polygon p){
        System.out.println( "A " + p.getClass().getSimpleName() + " has
" + p.getNumberOfSides() + " sides." );
        System.out.println( "The perimeter of this shape is: " +
p.getPerimeter() + '\n');
    }

    public static void main(String[] args) {
        Polygon triangle = new Triangle(1, 2, 3);
        print(triangle);

        Polygon rectangle = new Rectangle(2, 3);
        print(rectangle);

        Polygon square = new Square(2);
```

```
        print(square);

    }
}
```

When run, the *Solution* class produces the following output:

```
A Triangle has 3 sides.
The perimeter of this shape is: 6.0

A Rectangle has 4 sides.
The perimeter of this shape is: 10.0

A Square has 4 sides.
The perimeter of this shape is: 8.0
```

By having *Rectangle*, *Square*, and *Triangle* implement *Polygon* (as opposed to having them be completely separate standalone classes), we have a guarantee that all three classes (and any future classes that implement *Polygon*) will follow the same basic rules. Knowing the rules for *what* any class implementing *Polygon* will do enables us independently write code that uses some type of *Polygon*, regardless of how it's implemented.

Another benefit is that if we decide to improve upon our *Polygon* interface by adding a `double getArea()` method, our code will not compile unless we add an implementation for *getArea* to each class that implements *Polygon*.

Sorting

Sorting is the act of ordering elements. The ability of a program to organize and retrieve data quickly and efficiently is incredibly important in software development. Learning how to effectively sort and retrieve the data you're working with enables you to write better, faster algorithms.

Bubble Sort

This is a very simple sorting algorithm. Because it's also very inefficient, Bubble Sort is not practical for real-world use and is generally only discussed in an academic context. The basic theory behind BubbleSort is that you take an array of integers and iterate through it; for each element at some index whose value is *greater than* the element at the index following it (i.e., index), you must swap the two values. The act of swapping these values causes the larger, unsorted values to float to the back (like a bubble) of the data structure until they land in the correct location.

Implementation

```
import java.util.*;

class Sorting {
    private static void printArray(String s, int[] x) {
        System.out.print(s + " Array: ");
        for(int i : x){
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void bubbleSort(int[] x) {
        printArray("Initial", x);

        int endPosition = x.length - 1;
        int swapPosition;
```

```
        while( endPosition > 0 ) {
            swapPosition = 0;

            for(int i = 0; i < endPosition; i++) {

                if( x[i] > x[i + 1] ){
                    // Swap elements 'i' and 'i + 1':
                    int tmp = x[i];
                    x[i] = x[i + 1];
                    x[i + 1] = tmp;

                    swapPosition = i;
                } // end if

                printArray("Current", x);
            } // end for

            endPosition = swapPosition;
        } // end while

        printArray("Sorted", x);
    } // end bubbleSort

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int[] unsorted = new int[n];
        for (int i = 0; i < n; i++) {
            unsorted[i] = scan.nextInt();
        }
        scan.close();

        bubbleSort(unsorted);
    }
}
```

Generics

Generic constructs are a very efficient way of reusing your code. In the same way you have parameters for a function, generics parameterize *type* and allow you to use apply the same interface, class, or method using different data types while still restricting operations to that data type (meaning that you still get strong type checking at compile time).

Sounds confusing? You've already used generics in more than one challenge! Consider the List and Map interfaces, as well as the classes that implement them. Their respective headings are:

```
public interface List<E> extends Collection<E>
public interface Map<K,V>
```

The letters enclosed between angle brackets (< and >) are *type parameters* and, like many things in programming, there is a convention behind them (remember, following conventions help us write clean, readable code!). The letters below are commonly-used generic type parameters:

- E - Element
- K - Key

- `V` - Value
- `N` - Number
- `T` - Type (e.g.: data type)
- `S`,`U`,`V`, etc. These are second, third, and fourth types for when `T` is already in use.
-

A *parameterized type* is basically a class or interface name that is immediately followed by a type parameter in angle brackets. Observe that *List* and *Map* are both parameterized types, and their respective parameters (`E`, `K`, and `V`) all follow the conventions shown above. This helps us make some assumptions about the type of objects these type parameters are standing in for.

Just like we pass arguments to functions and methods, we need to specify data types for our type parameters when we instantiate generic objects. For example:

```
List<String> stringList = new LinkedList<String>();
List<Integer> integerList = new ArrayList<Integer>();
Map<String, String> stringToStringMap = new HashMap<String, String>();
Map<String, Integer> stringToIntMap = new LinkedHashMap<String,
Integer>();
```

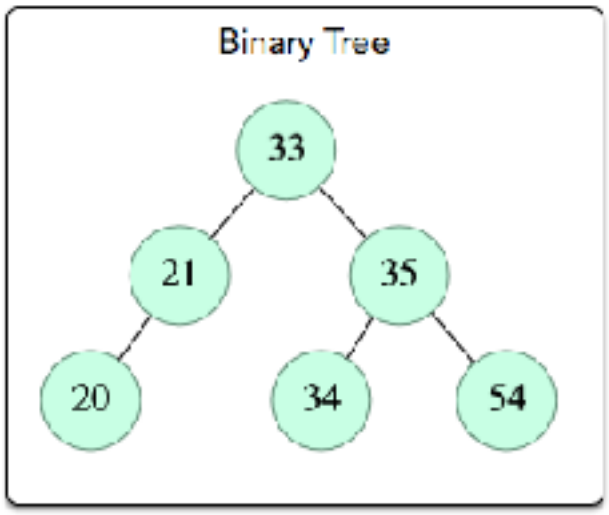
Once a data type is specified and an object is created, the specified type replaces every ocurrence of the generic type parameter in the instantiated class. The compiler also performs strict type checking to ensure you haven't tried to do anything not allowable for that data type (e.g.: trying to add an element to *integerList* that isn't of type *Integer*).

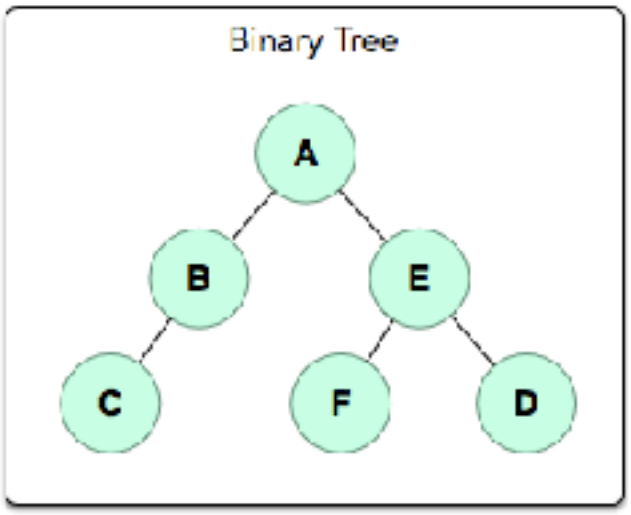
Check out Oracle's tutorials on Generic Types, Generic Methods, and Type Inference to learn more.

Binary Tree

Fundamentally, a binary tree is composed of nodes connected by edges (with further restrictions discussed below). Some binary tree, , is either empty or consists of a single *root* element with two distinct binary tree child elements known as the *left subtree* and the *right subtree* of . As the name *binary* suggests, a node in a binary tree has a *maximum* of children.

The following diagrams depict two different binary trees:





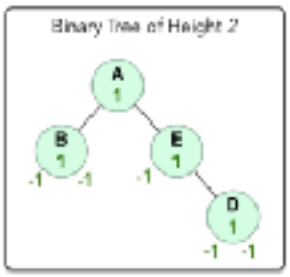
Here are the basic facts and terms to know about binary trees:

1. The convention for binary tree diagrams is that the *root* is at the top, and the subtrees branch down from it.
2. A node's *left* and *right* subtrees are referred to as *children*, and that node can be referred to as the *parent* of those subtrees.

3. A non-root node with no children is called a *leaf*.
4. Some node is an *ancestor* of some node if is located in a left or right subtree whose *root* node is . This means that the *root* node of binary tree is the ancestor of all other nodes in the tree.
5. If some node is an ancestor of some node , then the *path* from to is the sequence of nodes starting with , moving down the ancestral chain of children, and ending with .
6. The *depth* (or *level*) of some node is its distance (i.e., number of edges) from the tree's root node.
7. Simply put, the *height* of a tree is the number of edges between the *root* node and its furthest leaf. More technically put, it's (i.e., one more than the maximum of the heights of its left and right subtrees). Any node has a height of , and the height of an empty subtree is . Because the height of each node is the maximum height of its subtrees and an empty subtree's height is , the height of a single-element tree or leaf node is .

Let's apply some of the terms we learned above to the binary tree on the *right*:

1. The *root* node is .
2. The respective left and right children of are and . The left child of is . The respective left and right children of are and .
3. Nodes , , and are leaves (i.e., each node is a leaf).
4. The root is the ancestor of all other nodes, is an ancestor of , and is an ancestor of and .
5. The path between and is . The path between and is . The path between and is .
6. The depth of root node is . The depth of nodes and is . The depth of nodes , , and , is .
7. The height of the tree, , is . We calculate this recursively as . Because this is long and complicated when expanded, we'll break it down using an image of a slightly simpler version of whose height is still :



Traversal	Height (h)
A → B → left	1 + h(A.left) ⇒ 1 + (1 + h(B.left)) ⇒ 1 + (1 + (-1)) ⇒ 1
A → B → right	1 + h(A.left) ⇒ 1 + (1 + h(B.right)) ⇒ 1 + (1 + (-1)) ⇒ 1
A → E → D → left	1 + h(A.right) ⇒ 1 + (1 + h(E.right)) ⇒ 1 + (1 + (1 + h(D.left))) ⇒ 1 + (1 + (1 + -1)) ⇒ 2
A → E → D → right	1 + h(A.right) ⇒ 1 + (1 + h(E.right)) ⇒ 1 + (1 + (1 + h(D.right))) ⇒ 1 + (1 + (1 + -1)) ⇒ 2

In the diagram above, the height of is because that is the maximum height of 's left and right subtrees.

Binary Search Tree

A *Binary Search Tree* (BST), , is a binary tree that is either empty or satisfies the following three conditions:

1. Each element in the left subtree of is less than or equal to the root element of (i.e.,).
2. Each element in the right subtree of is greater than the root element of (i.e.,).
3. Both and are BSTs.

You can essentially think of it as a regular binary tree where for each node *parent* having a and , . In the diagram above, the binary tree of integers on the left side is a binary search tree.

Tree Traversal

A *traversal* of some binary tree, , is an algorithm that iterates through each node in exactly time.

InOrder Traversal

An *inorder* traversal of tree is a recursive algorithm that follows the left subtree; once there are no more left subtrees to process, we process the right subtree. The elements are processed in *left-root-right* order. The basic algorithm is as follows:

```
inOrder(t) {
    if(t is not empty) {
        inOrder( left subtree of t )
        process t's root element
        inOrder( right subtree of t )
    }
}
```

An inorder traversal of a binary search tree will process the tree's elements in *ascending* order.

PostOrder Traversal

A *postorder* traversal of tree is a recursive algorithm that follows the left and right subtrees before processing the root element. The elements are processed in *left-right-root* order. The basic algorithm is as follows:

```
postOrder(t) {
    if(t is not empty) {
        postOrder( left subtree of t )
        postOrder( right subtree of t )
        process t's root element
    }
}
```

PreOrder Traversal (DFS)

A *preorder* traversal of tree is a recursive algorithm that processes the root and then performs preorder traversals of the left and right subtrees. The elements are processed *root-left-right* order. The basic algorithm is as follows:

```
preOrder(t) {
    if(t is not empty) {
        process t's root element
        preOrder( left subtree of t )
        preOrder( right subtree of t )
    }
}
```

Because a preorder traversal goes as deeply to the left as possible, it's also known as a *depth-first-search* or *DFS*.

Level-Order Traversal (BFS)

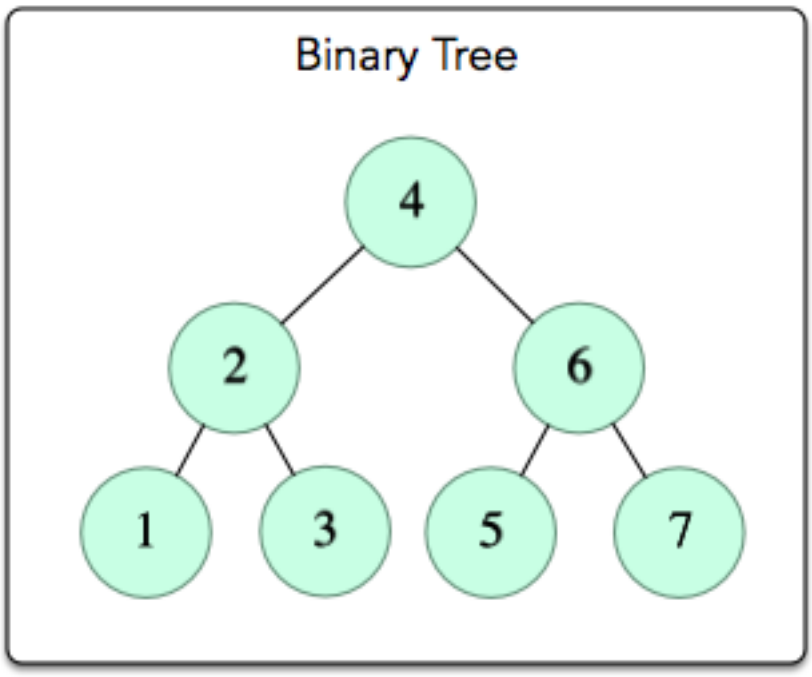
A *level-order* traversal of tree is a recursive algorithm that processes the root, followed by the children of the root (from left to right), followed by the grandchildren of the root (from left to right), etc. The basic algorithm shown below uses a queue of references to binary trees to keep track of the subtrees at each level:

```
levelOrder(BinaryTree t) {
    if(t is not empty) {
        // enqueue current root
        queue.enqueue(t)

        // while there are nodes to process
        while( queue is not empty ) {
            // dequeue next node
            BinaryTree tree = queue.dequeue();

            process tree's root;

            // enqueue child elements from next level in order
```

```
            if( tree has non-empty left subtree ) {
                queue.enqueue( left subtree of t )
            }
            if( tree has non-empty right subtree ) {
                queue.enqueue( right subtree of t )
            }
        }
    }
}
```

Because a level-order traversal goes level-by-level, it's also known as a *breadth-first-search* (*BFS*).

Example



The binary tree above has the following traversals:

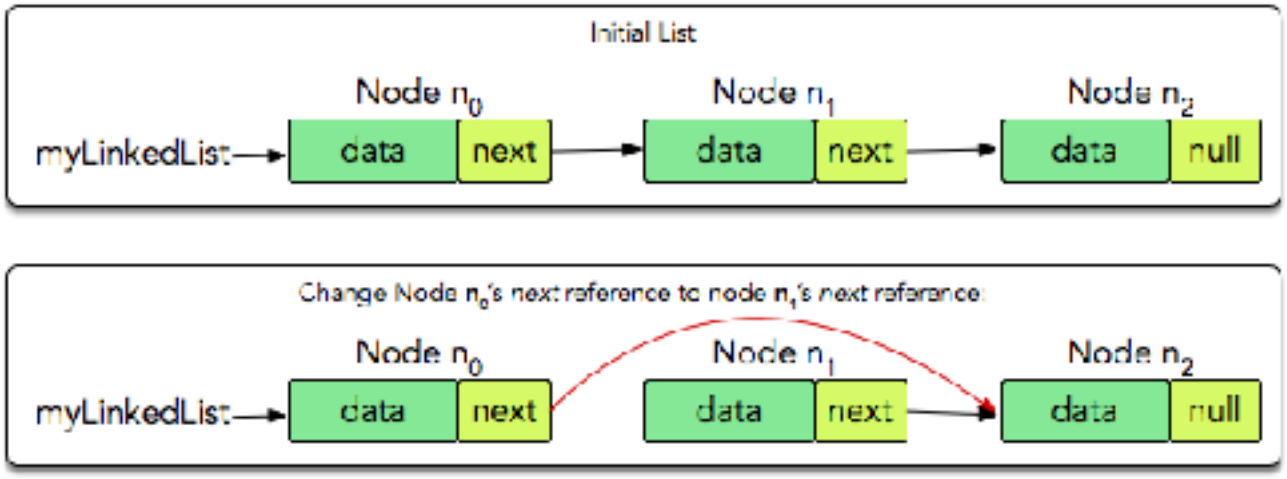
- InOrder: `1 2 3 4 5 6 7`
- PostOrder: `1 3 2 5 7 6 4`
- PreOrder: `4 2 1 3 6 5 7`
- Level-Order: `4 2 6 1 3 5 7`

Garbage Collection

This is a form of automated memory management that frees up memory locations that are no longer in use. When a program reserves a block of memory (e.g.: for storing variables), it marks it as 'in-use' and returns a reference to it which is managed by your variable. If a program is poorly written, the reference to that block of memory may be lost or broken; once the reference to that location is lost, the program can neither access the data stored there nor can it store new data there (because the memory was not *released* back to the program prior to breaking the reference). Java has something called *automatic garbage collection*; this means you don't have to worry about releasing (*deallocating*) blocks of memory you no longer need, because the Java garbage collector does it for you. If you're writing in a language that *does not* have garbage collection, you should make sure you are freeing up the memory that a reference refers to before deleting the reference itself.

Deleting Nodes from Linked Lists

Removing an element from a linked list of nodes is easier than it sounds! In a language with automatic garbage collection (like Java), you simply have to change the *next* reference from one node so that it points to another node. In the diagram below, we remove *n1* by changing *n0*'s *next* reference so that it points to *n2*. At this point, there is no longer a named reference to node *n1*, so Java's garbage collector will delete the orphaned node and free up the memory storing its data. In another language like C++, you should create a temporary reference to the node you are going to orphan, then change the reference from the previous node, and finally *delete* the node (which you have access to, thanks to the temporary reference you made).



Asymptotic Analysis: A *very* limited overview.

This is a means of discussing the general efficiency of an an algorithm. When discussing the time complexity of an algorithm, we use the positive integer to represent the size of the data set it processes. To evaluate the actual algorithm, we must *ignore machine-dependent constants* (i.e., think about the number of instructions being executed, *not* about how fast a certain machine can execute them) and look at its *growth rate* as approaches (i.e., how does it grow as a function of time — especially as gets large?).

Here are the big terms you need to know:

1
2 is Theta of if and only if there exists some positive constants , , and such that whenever . In short, is bounded both above and below by because after some point , and have the same *growth rate*.
3
4 is ("oh" or "big oh") of if and only if there exists some positive constants and such that whenever . In short, is *bounded above* by because after some point , will always grow at a larger asymptotic *growth rate* than .
5
6 is Omega ("big omega") of if and only if there exists some positive constants and such that whenever . In short, is *bounded below* by because after some point , will always grow at a larger asymptotic *growth rate* than .

The term time, or "constant time", is used to refer to fundamental operations that take a constant amount of time to execute (e.g., reading a single value, performing a comparison between two values, checking a condition, etc.).

At a very basic level, you need to think about how many instructions your algorithm *must* execute in the best and worst case scenarios when processing pieces of data. Then determine the function(s) that your algorithm is bounded above and below by, disregarding any leading constants (e.g., , , etc.) or lower-order terms (e.g., is a lower-order term than); basically, you don't

hang on to anything that doesn't directly impact the *growth rate* of and you only want to retain the term that has the greatest impact on growth rate (e.g., if , then is .
Resource: Algorithms Sequential & Parallel: A Unified Approach.
Example

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

    for(int j = 0; j < n/2; j++) {
        // 0(1) operation
        // 0(1) operation
        // 0(1) operation
    }
}
```

In the nested loop above, there are three constant time operations that will be performed times as a result of the nested loops. Because , our code is .
Primality Algorithms
A *prime number* is a natural number greater than that is only divisible by and itself (note that is *not* a prime). A *primality algorithm* is an algorithm for determining if some number, , is prime. The most basic primality algorithm is to simply iterate through each integer (where) and determine if it evenly divides ; if is an even divisor of , then is not prime. While this algorithm is , there are a number of optimizations you can perform that will reduce it to (even that can be slightly improved upon, though we will not discuss it here). What are you waiting for? Get started on today's challenge! And, when you're done, be sure to check the *Editorial* for a comparison of four primality algorithms.

Regular Expressions (RegEx)

This is a means of describing a set of strings using a subset of common characteristics. For example, the common characteristics of *aaa*, *aba*, and *aca* are that they all start and end with the letter *a*.

Regular expressions add a level of abstraction to strings that makes it easier to search and manipulate text. While they can seem very confusing at first, the syntax for regular expressions is pretty standard across different programming languages (so knowledge of RegEx is universally applicable).

Character Classes

This is not a class in the traditional sense, but rather a term that refers to a set of one or more characters that can be used to match a single character from some input string. Here are the basic forms:

- *Enclosed within square brackets.* Specify the what you'd like your expression to match within square brackets; for example, `[a-f]` will match any lowercase *a* through *f* character.
- *Predefined: `\` followed by a letter.* For example, `\d` for matching digits (*0-9*) or `\D` for matching non-digits. There are also additional predefined character classes that are followed by a set of curly braces, such as `\p{Punct}` which matches punctuation (i.e.: *!"#$%&'()*+,-./:;<=>?@[]^_`{|}~*).
-

Some key constructs to know are:

- **.** The period will match *any character*; it does *not* have to be a letter.
- **+** When appended to a character or character class, it means 'one or more instances of the previous character'.

- ***** When appended to a character or character class, it means 'zero or more instances of the previous character'.
- **^** if this is *before* a character class, it means you're matching the *first character*; however, if this is the first character inside a bracketed character class, it means negation/not. For example, **^[a].+** or **^a.+** matches any consecutive sequence of *2* or more characters starting with the letter *a*, and **^[^a].+** matches any consecutive sequence of *2* or more characters *not* starting with *a*.
- **$** When appended to a character or character class, it means 'ends with the previous character'. For example, **.+a$** will match a sequence of *2* or more characters ending in *a*.
-

Java's *Pattern* class documentation is a really handy reference for predefined character classes, but there are also many other RegEx resources available throughout the internet.

Coding With RegEx

When using regular expressions in your code, you need to remember that the `\` character is an escape character. This means that whenever `\` is occurs in a string (or as part of a char), it's a sort of flag indicating you may be trying to write a special instruction.

For example, when printing a string containing the *newline* character (`\n`); the backslash tells the compiler that you're writing a special instruction, and the `n` immediately following it indicates that your instruction is that it be a newline character. If your intent is to have the literal backslash character in your string, then you must write it as `\\`. When writing a RegEx string in code, you need your predefined character classes to be part of the string so you *must precede them with an additional backslash*. For example, a string intended to be used as a regex containing the `\d` character class *must* be written as `"\\d"`.

For Java, the important classes to know are Pattern and Matcher. A *Pattern* object is a compiled representation of a regular expresson. To create a *Pattern* object, you must invoke one of its static *compile* methods (usually *Pattern.compile(String regex)*). You must then create a *Matcher* object that matches your *Pattern* object (compiled RegEx) to the String you want to check. To do this, you must invoke *Pattern*'s matcher method on your *Pattern* object and assign it to a variable of type *Matcher*. Once created, a *Matcher* object can be used to perform matching operations.

```
// This will match a sequence of 1 or more uppercase and lowercase
English letters as well as spaces
String myRegExString = "[a-zA-Z\\s]+";

// This is the string we will check to see if our regex matches:
String myString = "The quick brown fox jumped over the lazy dog...";

// Create a Pattern object (compiled RegEx) and save it as 'p'
Pattern p = Pattern.compile(myRegExString);

// We need a Matcher to match our compiled RegEx to a String
Matcher m = p.matcher(myString);

// if our Matcher finds a match
if( m.find() ) {
    // Print the match
    System.out.println( m.group() );
}
```

The code above prints:

```
The quick brown fox jumped over the lazy dog
```

Notice that the ellipsis (...) at the end of the `myString` was not printed as part of the match; that is because `myRegExString` only matches lowercase and uppercase English letters and spaces—not punctuation. Thus, the ellipsis serves as a boundary for the end of our matched text.

For a string with more parts, you could do something like the following code which matches strings of one or more sequential alphanumeric characters:

```
String s = "Hello, Goodbye, Farewell";
Pattern p = Pattern.compile("\\p{Alpha}+");
Matcher m = p.matcher(s);

while( m.find() ){
    System.out.println(m.group());
}
```

This loops through the string, finds each match, and prints it:

```
Hello
Goodbye
Farewell
```

String Functions

Regular expressions can be pretty confusing, especially to new coders. Fortunately, there are still some workarounds using simpler regular expressions that you may find a little easier to use. If you know something about the structure of the strings you'll be working with (i.e.: that they all follow the same format), you can use a *split* method (e.g.: *String.split(String regex)* in Java, *strtok* in C++, *str.split()* in Python, *split* in Ruby, etc.). If you're able to section/cut your input string into components, then you can choose the pieces you want to use. For example, the code below splits two strings at a delimiter (a comma followed by a space) and puts whatever falls between the delimiter into an array:

```
String s1 = "Hello, Goodbye, Farewell";
String s2 = "Hola, Adios, Hasta Luego";

String myDelimiter = ", ";

String[] s1array = s1.split(myDelimiter);
String[] s2array = s2.split(myDelimiter);

System.out.println("s1array[0]: " + s1array[0]);
System.out.println("s1array[1]: " + s1array[1]);
System.out.println("s1array[2]: " + s1array[2]);
System.out.println("s2array[0]: " + s2array[0]);
System.out.println("s2array[1]: " + s2array[1]);
System.out.println("s2array[2]: " + s2array[2]);
```

The above code prints:

```
s1array[0]: Hello
s1array[1]: Goodbye
s1array[2]: Farewell
s2array[0]: Hola
s2array[1]: Adios
```

```
s2array[2]: Hasta Luego
```

Note: This only really makes sense to do when you know that the input strings follow a uniform format. You may also find the *replaceAll* method helpful when using this approach.

Logical Statements and Boolean Algebra

If you're not familiar with the term *boolean algebra*, you might be surprised to know that you likely already learned how to do this during the *logic* unit commonly taught during High School (secondary education). If you need a refresher, check out the Wikipedia article on Truth Tables. Instead of using (true) and (false), boolean algebra uses the binary numbers for *true* and for *false*.

Basic Operators

Here are some commonly used Java operators you should familiarize yourself with:

- **&** *Bitwise AND* (). This binary operation evaluates to (true) if both operands are true, otherwise (false). In other words:
  `1 & 1 = 1`
- `1 & 0 = 0`
- `0 & 1 = 0`
- `0 & 0 = 0`
-
- **|** *Bitwise Inclusive OR* (). This binary operation evaluates to if *either* operand is true, otherwise (false) if both operands are false. In other words:
  `1 | 1 = 1`
- `1 | 0 = 1`
- `0 | 1 = 1`
- `0 | 0 = 0`
-
- **^** *Bitwise Exclusive OR* or *XOR* (). This binary operation evaluates to (true) if and only if exactly one of the two operands is ; if both operands are or , it evaluates to (false). In other words:
  `1 ^ 1 = 0`
- `1 ^ 0 = 1`
- `0 ^ 1 = 1`
- `0 ^ 0 = 0`
-
- **~** The unary *Bitwise Complement* operator flips every bit; for example, the bitwise-inverted -bit binary number becomes , and the bitwise-inverted signed decimal integer becomes .

Example

The code below converts a word and an integer to binary strings:

```
class BinaryString {

    BinaryString(String string){
        for( byte b : string.getBytes() ){
            System.out.print(Integer.toBinaryString(b) + " ");
        }
        System.out.println();
    }

    BinaryString(Integer integer){
        System.out.println(Integer.toBinaryString(integer));
    }
```

```
    public static void main(String[] args) {
        new BinaryString("HackerRank");
        new BinaryString(8675309);
    }
}
```

When run, it prints the following output:

```
1001000 1100001 1100011 1101011 1100101 1110010 1010010 1100001 1101110
1101011 // Binary for "H a c k e r R a n k"
100001000101111111101101 // Binary for the integer 8675309
```

Next, let's modify the above class to find and print the *OR* of each character in the string with :

```
class BinaryString {

    BinaryString(String string, Integer integer){
        String binaryInteger = Integer.toBinaryString(integer);

        for( byte b : string.getBytes() ){
            // Perform a bitwise operation using byte and integer
operands, save result as tmp:
            int tmp = b | integer;
            System.out.println( Integer.toBinaryString(b) + " OR " +
Integer.toBinaryString(integer)
                + " = " + Integer.toBinaryString(tmp) + " = " + tmp );
        }
    }

    public static void main(String[] args) {
        new BinaryString("HackerRank", 8675309);
    }
}
```

The above code produces the following output:

```
1001000 OR 100001000101111111101101 = 100001000101111111101101 = 8675309
1100001 OR 100001000101111111101101 = 100001000101111111101101 = 8675309
1100011 OR 100001000101111111101101 = 100001000101111111101111 = 8675311
1101011 OR 100001000101111111101101 = 100001000101111111101111 = 8675311
1100101 OR 100001000101111111101101 = 100001000101111111101101 = 8675309
1110010 OR 100001000101111111101101 = 100001000101111111111111 = 8675327
1010010 OR 100001000101111111101101 = 100001000101111111111111 = 8675327
1100001 OR 100001000101111111101101 = 100001000101111111101101 = 8675309
1101110 OR 100001000101111111101101 = 100001000101111111101111 = 8675311
1101011 OR 100001000101111111101101 = 100001000101111111101111 = 8675311
```

Notice that the first bits () are always the same. This is because bit position is counted starting with the least-significant (rightmost) bit and then it moves left so, in the example above, the only values with the *potential* to change are the lower (rightmost) bits (as that is the number of bits in the smaller operand). For each bit position in the lower bits, an *OR* operation is performed. If we were to again modify the above code to print the *exclusive OR* (instead of the inclusive OR), we would get this output:

```
1001000 XOR 100001000101111111101101 = 100001000101111110100101 =
8675237
1100001 XOR 100001000101111111101101 = 100001000101111110001100 =
8675212
1100011 XOR 100001000101111111101101 = 100001000101111110001110 =
8675214
1101011 XOR 100001000101111111101101 = 100001000101111110000110 =
8675206
```

```
1100101 XOR 10000100010111111111101101 = 10000100010111111110001000 =
8675208
1110010 XOR 10000100010111111111101101 = 10000100010111111110011111 =
8675231
1010010 XOR 10000100010111111111101101 = 10000100010111111110111111 =
8675263
1100001 XOR 10000100010111111111101101 = 10000100010111111110001100 =
8675212
1101110 XOR 10000100010111111111101101 = 10000100010111111110000011 =
8675203
1101011 XOR 10000100010111111111101101 = 10000100010111111110000110 =
8675206
```

If you're still having some trouble understanding how bitwise operations work, spend some time comparing the different outputs and experimenting with the code that produced them.