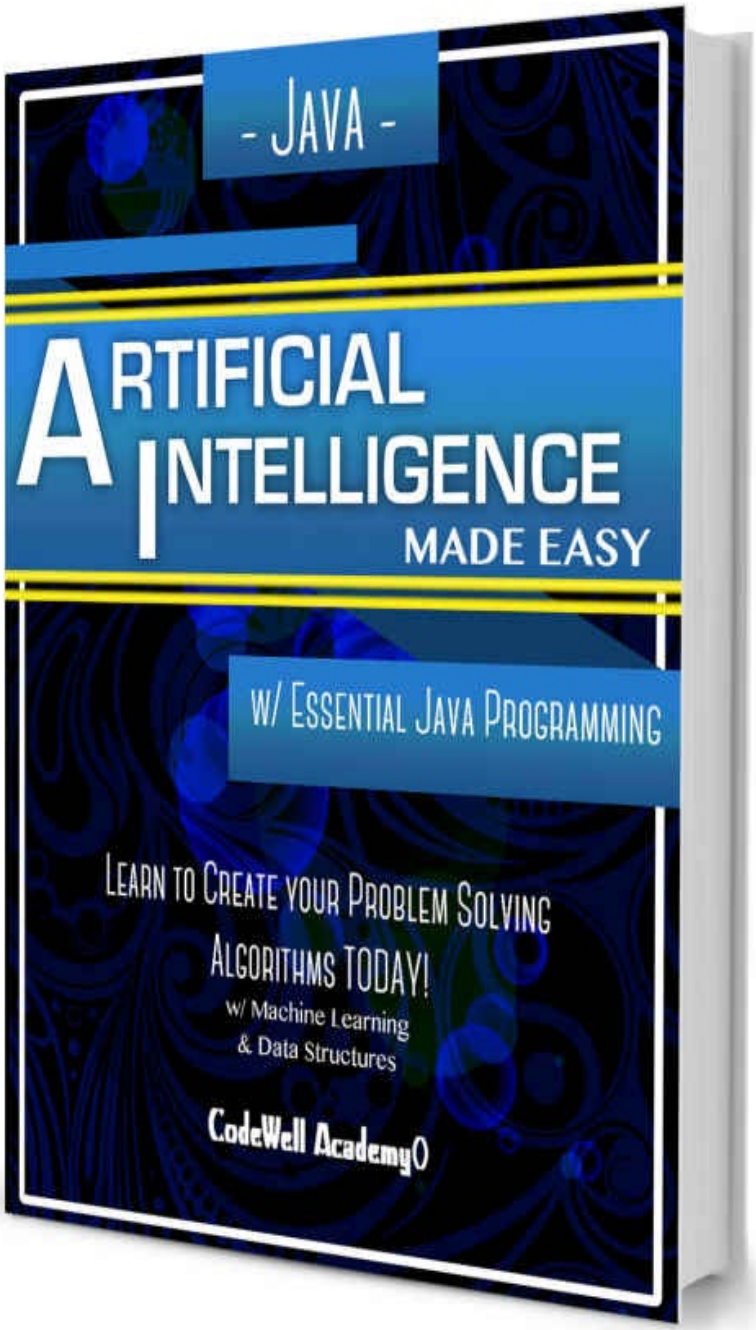
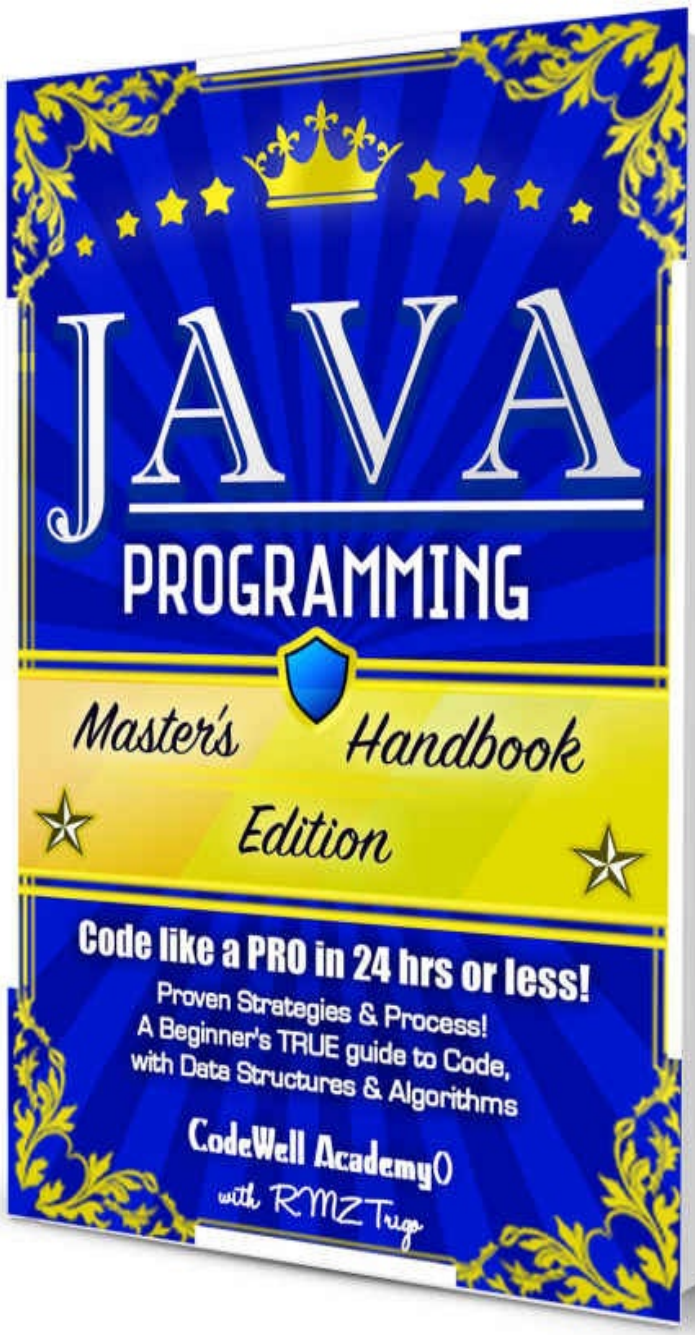

- JAVA -
ARTIFICIAL INTELLIGENCE
MADE EASY
W/ ESSENTIAL JAVA PROGRAMMING
LEARN TO CREATE YOUR PROBLEM SOLVING ALGORITHMS TODAY!
w/ Machine Learning & Data Structures
CodeWell Academy0


JAVA
PROGRAMMING
Master's Handbook
Edition
Code like a PRO in 24 hrs or less!
Proven Strategies & Process!
A Beginner's TRUE guide to Code, with Data Structures & Algorithms
CodeWell Academy0
with RMZ Trigo

CodeWell Academy() and R.M.Z.
present:

Java Programming
Box Set

Master’s Handbook

&

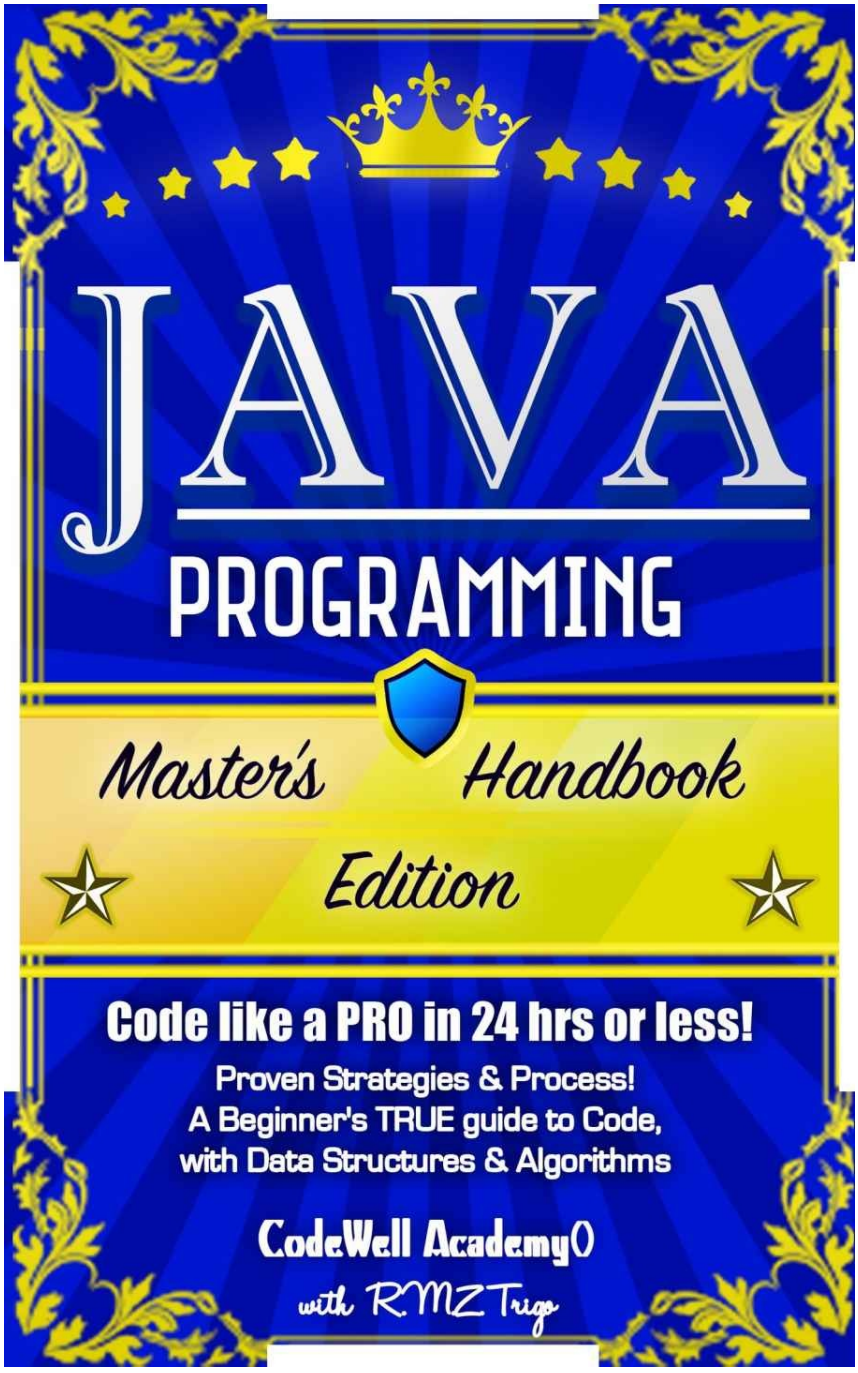
Artificial Intelligence Made Easy

1st Edition

w/ Code, Data Science, Automation,
problem solving, Data Structures & Algorithms

CodeWell Box Set Series

Table of Contents


JAVA
PROGRAMMING
Master's Handbook
Edition
Code like a PRO in 24 hrs or less!
Proven Strategies & Process!
A Beginner's TRUE guide to Code,
with Data Structures & Algorithms
CodeWell Academy0
with R.MZ Trigo

CodeWell Academy() and R.M.Z.
present:

Programming Java,

Master’s Handbook Edition

3rd Edition

Code like a PRO in 24 hrs or less!

Proven Strategies & Process!
A Beginner’s TRUE guide to Code,
with Data Structures & Algorithms

Master's Handbook Series

Introduction

========== ======

Welcome to the Path of Mastery

We thank you for purchasing & downloading our work, the Master’s Handbook. By doing so, we can tell you have a curiosity to learn Programming in a deeper, more comprehensive way.

We notice that you don’t just want to learn a few tricks here and there, but you want the confidence to take on any programming challenge with ease.

Hence, you’ve come to the right place…

The Master’s Circle

You aren’t alone.

Behind this book are programmers hailing from some of the most Best Computer Science Programs taught by some of the most Advanced Universities in the World Today.

Foundations

There are two major things that you need to do in order to be a good programmer. One is to get a good amount of practice. The other is to get a really good education.

But how can you tell what source of education is good or not?

You see, we can tell you that 90% of programming learning sources out there will show you WHAT the code is and HOW it works, for any amount of Programming languages. But that’s not necessarily a bad thing - and plenty of those sources are really good too.

However, they might not teach you WHY or WHEN you would use any particular code. NOR would they show you WHAT ELSE you might need with that code.

If you needed to program something, you don’t want to be someone that knows a bunch of code but does not know how to use them, right?

As you read on, you’ll quickly learn not just the HOW and WHAT the code is, but the WHERE, WHEN, WHY to use it, WHAT ELSE you’ll need with it- and more importantly, HOW to really use it.

The Master Structure

We start by observing the world and defining the code to represent things (data) or actions (functions). As you progress through the book, you’ll find more advanced concepts and ways to combine them all together.

You’re also accommodated with the Main Programming Language this book comes with, as well as general PseudoCode to help understand coding concepts. Often times, you’ll find that our PseudoCode bridges

you from learning this Book’s Main Language to learning your Next Language!

Whether you haven’t coded a single line before, or you’ve already build serious projects, you WILL find great value in this book. Often times, you’ll run into a coding challenge in your programming journey. This book will help you identify how to progress through it!

========== ======

Editor’s Note

========== ======

Reality

If you ever wonder why Computer Scientists make so much money (including junior programmers, developers, software engineers, IT folks, and just about any job involving programming), there are plenty of good reasons.

One, is that in today’s world, just about EVERY industry out there requires a level of technological sophistication. So you can imagine the level of demand for hiring a qualified programmer.

But the truth about Computer Science is that it can be very intellectually challenging most of the time. Thus, only a certain number of people will be good enough to get a real career in programming.

So there you have it. High Demand + Few Good Programmers = GOOD Salary.

However, you may not need a university degree to be a good programmer…

In-Depth

The true point of this book, along with others in this series, is to go deeper than the lines of code you see. You'll learn how to use every bit of data and code to any situation you encounter - and at one point, intuitively.

A real programmer's job is to create tools that improve life in one way or another. Almost all of the time, it will involve having to come up with ways to represent things in life as data - as well as getting a computer to process the data properly and turn into something useful.

So if you want to become a better programmer, read on…

JAVA Introduction

========== ======

JAVA: Universal Remote Control

Java is one of the most widely used, popular programming languages in the world today. And it’s not difficult to see why. Endless apps across many platforms - including Windows, Linux, Android & Windows phones, and even servers - are powered by Java code. In most schools and colleges that teach programming, Java is often the first language taught. With many Java programmers available, massive app projects can be crafted more easily.

However, the Java language does have its challenges. Many programmers may find Java’s syntax strict and inflexible. Also, programmers may find that Java is quite verbose. For the exact same program functionality, Java often requires more code written than other languages. Furthermore, programmers need to be careful with Java app memory usage - as Java-based apps often use more memory to run.

Nevertheless, learning to code in Java can be very rewarding - in terms of skill and salary.

JAVA Advantages:

- universal; popular
- massive array of java libraries & API’s
- code runs on many platforms

JAVA Disadvantages:

- inflexible, complex
- verbose
- higher memory usage

JAVA Workshops:

These workshops are yours to complete in whichever way you like (However, the code MUST work!).

They’re designed to put the most recent concepts into real-life practice, yet giving you the flexibility and critical thinking along the way.

And of course, flexibility and deep critical thinking are key programmer traits!

Find them throughout the book!

========== ======

JAVA-00: Quick Important notes about Java code

A Java Class File

Classes will be explained more in detail later in JAVA-02. But for now, start your Java code with these next few lines of code:

```
// ————————————————————

import java.util.*;

import java.lang.*;

import java.io.*;


class ___{

public static void main(String args[]) {

// CODE HERE

}

}

// ————————————————————
```

By default, online compilers will have a similar code structure to the above. For now, just make sure you ONLY add code within the brackets by ‘public static void main’ (where the comments tell you to)

Also, make sure all three import lines above are in your code. Plenty of Java functionality is from these Java libraries.

NOTE:

if you use an online compiler we mentioned, fill in the blank line beside class into one of the below, depending on which site you've used:

Rextester (rextester.com): fill with 'Rextester'

CodeChef (www.codechef.com/ide): fill with 'Codechef'

Codepad: (doesn't support Java)

Ideone (https://ideone.com/): fill with 'Ideone'

Prelude: Atomic Data Types

First off, we'll briefly start with primitive data types. It's important to know what they are, because you'll be identifying real-life information with them later.

Booleans

Booleans, often called bools, are either TRUE or FALSE. This is the simplest data type, but often one of the most important. A LOT of functionality depends on Booleans, as you will find out later

A Boolean will always be a two-state situation. For example, the lights in your living room are either on (TRUE) or off (FALSE).

Integers

These are all the standard whole numbers, both positive and negative. Higher and lower (negative) integers depend on the number of bits to represent them (i.e. 8-bit integers, 16-bit, etc.). Mathematic and boolean operators often use Integers.

For example:

10 + 50 == 60,

-4 - 12, != -10,

and so on.

Characters

These are all letters and symbols that can be represented by ASCII characters. Think of one character as a single symbol or letter, such as upper or lower case letters, symbols (!,@,#,$ and so on), and even numbers. However, take note: you cannot do any arithmetic with number characters. (i.e. if you have characters 12 and 9, you can’t add them, subtract, and so on.)

Floats

Formally called floating-point numbers, these represent decimals - including the decimal point and decimal numbers beyond.

Examples:

2.3

0.75

Prelude: Data Sequences & Combinations

Strings

These are merely a collection of Characters in sequence. Think of these as words or phrases.

In most programming languages, Strings are represented by a sequence of characters between quotation marks: *“Hi there”*, or *“Hello.”*, for example.

How these would look like as a sequence of characters is as follows:

“Hi There” is represented as characters H,i, space, T, h,e , r, and e

“Hello.” is represented as characters H, e, l, l, o, and the period.

Lists

These are a sequence of individual elements put together as a list. Often times, all the elements within that list are the same data type

Examples would be:

a List of Integers: *[3, 1, 4, 9, 2]*

a List of Strings: *[“Apple”, “Banana”, “Caramel”]*

a List of Booleans: *[true, false, true, true]*

Enumerations

These are fixed sets of data values. The data within these sets are the same data type. You would have to choose between one of the data elements within that set.

For example, traffic lights are either red, orange, or green.

As an enumeration, traffic lights would be: *[“Red, “Green”, “Yellow”]*

What differs Enumerations from lists is that Enumerations are supposed to have a FIXED set of values. You wont be able to add or delete the elements unless you edit the code directly.

Itemizations

Itemizations combine different data types together to form a finite set. Depending on the data type of a single element, you'll have to process that data in a certain way (you'll learn about this later in function templates).

Enumerations have the same data type, but Itemizations have different data types in the set. You also won't be able to add or delete any of the elements as well.

For example, a Space Rocket launch would be a set of integers 10 to 1, then the booleans true or false, to signifiy whether or not it has launched yet.

As an itemization, a Space Rocket Launch would be: [false, 10, 9, … 1, true]

Prelude: Your Coding Environment

Simple Online IDE

For now, it’s all about understanding all the Programming Concepts, from the simple to the downright advanced.

To test out these concepts, you’ll only need a simple online Compiler to run your code and make sure it works the way you planned it to.

Here’s a few ones online. They’re FREE and they don’t require any membership to test out your code:

www.codechef.com/ide

codepad.org

rextester.com

https://ideone.com/

Full Development Kits

You may also set up your computer for app development, if you wish.

First, identify what Programming Language(s) and Framework(s) you wish to use. Then choose and set up the ideal Integrated Development Environment (IDE) for your language-framework combinations.

Popular IDE’s include IntelliJ, Eclipse, Netbeans, CodeLite, XCode (for Mac Users) and more. But remember: make sure your IDE supports the programming languages of your choice.

NOTE: Comments

For newcomers just learning how to program, it’s important to describe what your lines of code are and why they exist in your code.

For programming teams, comments are essential. Even seasoned programmers who work in teams need to explain what their code does and why they have it. Also, programming teams find that they save time analyzing and figuring out other people’s code. They will also have less errors along the way - knowing that they’re coding what they mean to code, or editing code to be the way they originally want it to be.

PART I: Your Code Structure and Foundations

Chapter 1: Defining & Designing your Data

Anything in this universe can be represented by data. Your name, age, gender, what car you drive, what city you live in, country, planet, galaxy, and so on.

To design great apps, games, and any other digital tool you can think of, you'll have to identify what type of data you're dealing with.

Identify your Data Type

There are two questions to ask yourself when you're designing data to represent something.

First, identify whether or not you can define that 'thing' as an Integer, Number, Boolean (yes/no type things), or String. For example, your name is a String (a sequence of letters), your Age is a Number, and you live in a City or Town. Your city of town has a name - another String.

Second, identify whether or not that thing is a part of a whole; included in a larger thing. For example, you may have a friend named Jamie. She's included in a list of your friends.

Representing your information as a Data Type

In most programming languages, you can define your data in a line of code. Here we define your data as a Global Variable - meaning this editable line of data is available throughout your program.

In most languages, you'll very likely declare the Type of data you represent, as well as the Name of your data.

Let's start with your name, using pseudocode for now:

String NAME;

In the above line of code, you defined your data as a String of characters, labelled as a NAME.

An Explanation for your Data

A good practice in defining your data is to make comments above the line of code that explains why you have your data the way it is. One of the lines can be in the format of "____ is a (Data Type)".

In most programming languages, comments usually start with two slashes (//). For the pseudocode we use here, we'll do the same

For the above line, here's the example:

// My Name is a String

String NAME;

Here are some more examples:

// My Age is a Number

Number AGE;

// I live in a City or Town, with a Name

// My city's name is a String

String CITY;

// I am either hungry or not

// My hunger status is a Boolean

Boolean HUNGRY?;

// I either have a pet or I don't

// Whether or not I have a pet is a Boolean

Boolean HASPET?;

JAVA-01: Defining & Designing your Data

JAVA Comments

In Java, comments are just the same as other common languages like C, C++, PHP, and even our own pseudocode.

Single-line comments start with two slashes (//) and end when the line breaks to the next line of code.

Multiple comment lines start with a slash and a star (/*) and end with the reverse: a star and a slash (*/). Often, multiple comment lines also start with stars (*) on each line.

Here’s an example:

/ This is a*

** multi-line*

** comment */*

Essential Java Syntax

Out of most programming languages out there, Java has one of the more strict code syntax.

Similar to other code such as C, C++, and PHP, lines of code in Java end with a semicolon (;). This is very important; Java programmers commonly have errors in their code just because their lines don't end with a semicolon.

Atomic Data in Java

Atomic data in Java is defined VERY strictly.

Booleans in Java are strictly lower case. If your booleans are all upper-case (TRUE), or first-letter uppercase (True), most Java compilers will report these as errors. So for booleans, stick with all lower-case letters. So either (true) or (false).

Strings in Java can only start and end with double parentheses (“). Do not use single parentheses (‘) to start and end strings. For example, “String” is a string, but ’String’ is not.

Instead, single characters in Java start and end with single parentheses (‘). For example, ‘a’ is a single character.

Floats in Java always end with the letter f. For example, some Floating Point Integers in Java can be 1.2f and 0.75f.

On the other hand, Java uses the data type Double to also represent decimal numbers. For example, 1.2 and 0.75 can be Doubles.

Java Data Definitions

Let’s define some data in Java.

Luckily, the pseudocode we’ve used earlier follows a very similar syntax to what we use here with Java. You will have to define your data type once you code the data you want to store. However, Java uses distinct, case-sensitive words to define your data variables.

For Strings, define with this data type: String (note the capital S!)

For Integers, define with this data type: int

For Booleans, define with this data type: boolean

For Floats (decimal numbers), define with this data type: float

For Doubles (decimal numbers), define with this data type: double

Overall, your data definitions in Java will look like our pseudocode:

(data type) (variable name);

So, let's practice. Let's start with defining the data for your friend's name. Let's call her Haley. First, let's add a comment line to note what data type we want to deal with:

// Haley's name is a String

Now we define the data for her name. Remember Java's the end-of-line semicolon:

// Haley's name is a String

String NAME;

The Java compiler knows that NAME is going to be a String, since you've defined that variable as that data type.

PRACTICE:

Let's start by declaring things that follow into each of: Strings, Integers, and Booleans.

First, go to an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or www.codechef.com/ide. (if you do, make sure to set your Programming Language to Java)

You should see something similar to this:

```
// ————————————————————
import java.util.*;
import java.lang.*;
import java.io.*;

class (whatever class name)
{
public static void main(String args[])
{
// INSERT CODE HERE
}
}
// ————————————————————
```

Now, copy-paste the below code to where you would INSERT your code above. Afterwards, fill in the blanks.

```
// ######
// Katie's name is what Java data type?
____ NAME = "Katie";
// Katie's age is what Java data type?
___ AGE = 21;
// Katie is EITHER married OR not.
// Katie's marriage status is what Java data type?
___ ISMARRIED = true;

System.out.println(NAME);
System.out.println(AGE);
System.out.println("Is " + NAME + " married? " + ISMARRIED);
// ######
```

If you try to run your code - and it runs correctly - you should see the following:

Katie

21

Is Katie Married? true

Chapter 2: Compound/Composite Data

From the last chapter's example, you can start to wonder that there just has to be a way to group all that data: your name, age, location, whether or not you have pets, etc.

Also, notice that a person can neither be a String, Number, Integer, nor Boolean. A person just holds too much data to be defined as either one of the above.

So what do we do?

What a Composite Data Structure is

From the previous example, you can think of all the data you've defined as small parts of a whole. But what is this "whole"?

Enter Composite Data.

A Composite Data structure includes many parts of data within it.

Those parts of the Composite could be whatever you wish to declare. Strings, Integers, Booleans, Lists, and even Other Composite Data.

Identify & Defining a Composite Data Structure

When you were asked earlier to define what type of data are you dealing with, what if you designed & defined data for an object that you couldn't identify as atomic data? What if it had plenty of Characteristics? What if there was more depth in that object?

The key thing to remember in identifying composite data is depth. There are more parts to that 'thing' you were trying to define as data. If there's more to anything than just a name, number, or true/false switch, then it's probably going to be a composite data structure.

Representing information as Composite Data

Let's take YOU as an example. You are a Person. As a person, you're not JUST a name or number; you are comprised of a lot of data. An endless amount of data, rather.

The Elements that Comprise your Data

As an example of Composite Data, let’s define you.

For now, let’s start with the basics.

Remember: you are a Person. Using simple pseudocode, let’s define that:

// I am a PERSON

CompositeStructure PERSON;

An Explanation for your Composite Data’s Parts

Similar to what you did earlier for defining data, it’s also best that you identify what your composite data structure is comprised of.

For practice, use comments to describe what your data structure has.

Following the example above, You have a Name and an Age as well. Let’s include that:

// I am a PERSON

// A person has:

// - a name (string)

// - an age (number)

CompositeStructure Person {

String NAME;

Number AGE;

{

You live in a City. Oh but wait, a City isn't just a name is it? It's comprised of plenty of data as well!

// This is a CITY

// A City has:

// - a name (string)

// - a Latitude and Longitude (2 numbers)

// - a Population count (an integer, above 0)

CompositeStructure City {

String NAME;

Number LATITUDE;

Number LONGITUDE;

Integer POPULATION;

{

Let's not forget about YOU now. You live in a City, remember?

// I am a PERSON

// A person has:

// - a name (string)

// - an age (number)

// - a City they live in (Composite data City)

CompositeStructure PERSON {

String NAME;

Number AGE;

City LOCATION;

}

Notice what happened here. A compound data structure within another compound data structure!

JAVA-02: Compound/Composite Data

In designing Java classes, the syntax is again very similar to the pseudocode we've used earlier. However, we define our composite data structure as a Data Class.

Think of classes as "blueprints" for representing & creating your objects as data. If you were to create a data object, you would simply define that data object by stating that it's using the same "blueprints" you've defined earlier - your Data Class

Java: an Object-Oriented Programming Language Only

As you can see, data structures classes form the foundation of Java code.

A java class file will always have its EXACT case-sensitive filename as one of its classes. At the very minimum, a java class file will have at least one class. For example, a java file named ClassOne.java will have a class named ClassOne.

ClassOne.java:

```
class ClassOne {
// code here
}
```

Here’s how a data structure with two attributes would look like, including the Explanation Comments:

```
// Our class has:
// - an attribute variable (String)
// - another attribute variable (Integer)
class className {
String attribute1;
int attribute2;
}
```

Initializing a Java Object

To create an Object based on your Java class, it would be just like setting a Java variable's initial data value. However, you'll be setting that variable as a new Object. This object will have the same data structure as the class you set it as.

For Java classes, there are a few special lines of code you'll need to create called a Constructor.

A defined constructor within a class will have the following structure:

```
public ExactClassName(datatype input) {
// insert any code here
}
```

Often times, the class constructor will be right below all of the attributes of that class.

Overall, initializing a Java object would look like the following notation:

ClassName OBJECTNAME = new ClassName();

The word 'new', along with the constructor for that class, initializes your object.

You've already defined that your object will have the data structure of whatever class you've assigned it to.

Accessing Class Attributes

This follows a similar structure to our PseudoCode.

To access a Class Attribute, you first have a variable object that has been assigned that Class structure. Then follow it up with a period (.), then the class attribute you want to access.

Let's say we have a class with two attributes in it:

```
class Class {
String attribute1;
int attribute2;
}
```

Next, let’s recall setting object1 into a class:

Class object1 = new Class();

Then accessing object1’s attributes would look like the following notation:

object1.attribute1

object1.attribute2

Can you guess what data type you end up with when you access these two class attributes? If you look at your data definition for your Class, you’ve defined the attributes for that class with specific data types. Therefore:

object1.attribute1 (this returns a String)

object1.attribute2 (this returns an Integer)

EXAMPLE:

Think about a book. A physical book that probably sits near your shelf. Let’s start defining the Composite Data Structure for it.

First, let’s use comments to describe what our our data will look like.

// A Book Has:

Now think of the little attributes a book has. Is the cover a Hardcover, Paperback, or something else? How many pages does it have? Is it a Fiction book or not? What's the title? Who's the Author? How much did that Book cost?

// A Book Has:

// - a Cover Type (String)

// - a Page Count (Integer)

// - a Title (String)

// - an Author Name (String)

// - a price (Number)

In Java, numbers with decimals can have either data type float or double. For now, let's go with double.

Now, let's design the Book Class in Java.

```
class Book {
String coverType;
int pageCount;
String title;
String authorName;
double price;
}
```

For fun, let’s also create a Harry Potter book object.

```
Book harryPotter = new Book();
```

JAVA Workshop #1

Go to an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or www.codechef.com/ide. (if you do, make sure to set your Programming Language to Java).

If you use one of the online IDE's, you should see something similar to this:

```
// ————————————————————

import java.util.*;

import java.lang.*;

import java.io.*;

// INSERT MORE IMPORTS HERE

// INSERT MORE CLASSES HERE


class (whatever class name)

{

public static void main(String args[])

{

// INSERT CODE HERE

}

}

// ————————————————————
```

Classy Kids

Design a data class definition for a car. Think carefully. What attributes would that car have?

First, use comments to describe what attributes you want the car to have. Then figure out what Data Types those attributes are. This is totally up to you. Make it up as you go along!

Then, code your data class to include those attributes.

Then, create a data object, which can be a car you really like (or hate).
Afterwards, test and run your code in an IDE of your choice.

You can also use a free IDE such as http://rextester.com/runcode

Chapter 3: Data Initialization

Let’s recall the Composite Data Structure of a Person:

// A person has:

// - a name (string)

// - an age (number)

// - a City they live in (Composite data City)

CompositeStructure PERSON {

String NAME;

Number AGE;

City LOCATION;

}

We’ll also create two Atomic Data pieces as the Time: two integers.

Integer HOUR;

Integer MINUTE;

You have have noticed one thing: You’ve defined what types of data you’re dealing with - but we don’t know any people yet. Nor do we know what time it is!

Now we INITIALIZE our data. Now that we’ve defined what types of data we have, we then set our data for the first time.

Initializing Atomic Data

First, we'll set the time.

Let's say it's 8:30 PM. We'll set up our time as is.

To set data, most programming languages use the Equals (=) operator. Here, we'll do the same.

HOUR = 20;

MINUTE = 30;

DON'T make this mistake…

But what if we tried to set HOUR and MINUTE to another data type?

HOUR = aaaa;

MINUTE = Composite{Integer; String; Boolean} ;

Notice earlier that you've set both HOUR and MINUTE as Integers. Here, we're trying to set up those data as different data types. In some programming languages, it's not going to work. And in most cases, your code might not work because of this.

Here, you've set up your data as Integers - therefore, you need to initialize & change them as Integers.

Lesson learned: if you set up your data as a certain data type, unless you really know what you're doing, DO NOT try to set up that data as another data type!

Initializing Composite Data Structures

Now, let's define an actual Person using our data structure.

There are FOUR KEY steps to do this:

STEP ONE:

Identify & describe a data object you're trying to create.

For practice, use comments to describe what that object is & what it's like.

We'll use a friend of yours called Jamie, for example. We use comments to describe her:

```
// Jamie Denise is a person
// She has:
// - a name: Jamie Denise
// - an age: 19
// - a City she lives in: New York
```

STEP TWO:

Declare what type of composite data your object is.

In this example, we declare Jamie as a person:

```
// Jamie Denise is a person
Person Jamie;
```

STEP THREE:

You INITIALIZE your object's data structure, so that your object actually IS represented by the Composite Data in your program. In most languages, you declare that you have a new 'case' or instance of this object. Think of this step as "registering" your new object into your data program.

In this example, we INITIALIZE Jamie as a data object that HAS the Person Composite Data Structure

// Jamie Denise is a person

Person Jamie = new Person;

Again, just like Atomic Data, we use the Equals (=) operator to set data.

STEP FOUR:

Identify your object. Then, for each data part that your Composite Data is made of, set those initial values.

Remember Jamie's Attributes?

// Jamie Denise is a person

// She has:

// - a name: Jamie Denise

// - an age: 19

// - a City she lives in: New York

Now let’s initialize each attribute onto our Data Object Jamie. You first need to identify the data object you’re trying to reach. In this case, it’s Jamie.

Next (and this is important!), identify which attribute you’re planning to reach. Here’s it’s best to reference the Data Structure you’ve defined earlier:

CompositeStructure PERSON {

String NAME;

Number AGE;

City LOCATION;

}

Let’s set all three of Jamie’s Attributes:

// Jamie Denise is a person

// She has:

// - a name: Jamie Denise

// - an age: 19

// - a City she lives in: New York

Person Jamie = new Person;

Jamie-NAME = “Jamie”; // <— a “String”: Remember?

Jamie-AGE = Nineteen;

CITY = NewYork;

Okay, we’re done.
Hold on. This code is wrong. Why?

DON’T make these mistakes…

This line: *Jamie-AGE = Nineteen;* won’t work. Why?

Just a friendly reminder. Make sure the data type you’re trying to set MATCHES the data type you’ve defined. In most programming languages, this is one of the most common mistakes programmers make. Nineteen is definitely not a Number data type, nor is it a String (where’s the “Quotation marks?”). However, 19 works.

Jamie-AGE = 19;

Also, This line: *CITY = NewYork;* won’t work. Why?

What’s CITY? Did we mean Jamie’s current CITY?

Remember to first identify the DATA OBJECT you’re accessing. AND THEN that object’s attributes.

Well, let’s try that.

Jamie-CITY = NewYork;

This line: *Jamie-CITY = NewYork;* won’t work either. Why?

Because Jamie is a data object that follows the Person Composite Data Structure you’ve defined. And note how that Structure does NOT have any attributes named CITY in it.

Again, Remember to first identify the DATA OBJECT you’re accessing. AND THEN access that object’s correct attributes.

The Person Structure includes a separate City data structure, but it certainly isn’t called CITY.

CompositeStructure PERSON {

String NAME;

Number AGE;

City LOCATION;

}

Oh, so it should be *Jamie-LOCATION= NewYork;*

But you’re missing one more thing. Where in your program is NewYork defined?

Well, that can be arranged. Let’s recall the City data Structure and define the NewYork data object as well:

// This is a CITY

// A City has:

// - a name (string)

// - a Latitude and Longitude (2 numbers)

// - a Population count (an integer, above 0)

CompositeStructure City {

String NAME;

Number LATITUDE;

Number LONGITUDE;

Integer POPULATION;

{

// NewYork is a CITY

// NewYork has:

// - a name: “New York”

// - a Latitude and Longitude: 40.7127 and 74.0059

// - a Population count: 8406000

City NewYork = new City;

NewYork-NAME = “New York”;

NewYork-LATITUDE = 40.7127;

NewYork-LONGITUDE = 74.0059;

NewYork-POPULATION = 8406000;

and now, we fully complete Jamie’s data entry:

// Jamie Denise is a person

// She has:

// - a name: Jamie Denise

// - an age: 19

// - a City she lives in: New York

Person Jamie = new Person;

Jamie-NAME = “Jamie”;

Jamie-AGE = 19;

Jamie-LOCATION= NewYork

Chapter 4: Data Changes & Mutable States

In the previous chapters, you've defined some facts as data structures and even represented people and cities as data.

However, nothing ever stays the same in data.

Data changes over time - and it's important to keep track of how data values change and what they currently are.

Modifying your Defined Data Over Time

In reality, modifying the data values you've set in place is nearly similar to initializing them in the first place. In most programming languages, the same principles between initializing and updating data apply: identify the data you want to access, use the Equals Operator (=), and set the new data to another value, but usually the SAME data type you've originally set. So change data defined as Strings to other Strings, Integers to Integers, and so on.

For example, let’s take a look at Jamie and New York from the past chapter:

// NewYork is a CITY

// NewYork has:

// - a name: “New York”

// - a Latitude and Longitude: 40.7127 and 74.0059

// - a Population count: 8406000

City NewYork = new City;

NewYork-NAME = “New York”;

NewYork-LATITUDE = 40.7127;

NewYork-LONGITUDE = 74.0059;

NewYork-POPULATION = 8406000;

// Jamie Denise is a person

// She has:

// - a name: Jamie Denise

// - an age: 19

// - a City she lives in: New York

Person Jamie = new Person;

Jamie-NAME = “Jamie Denise”;

Jamie-AGE = 19;

Jamie-LOCATION= NewYork

So let’s say 10 years have passed since we defined Jamie’s data object onto our program. Since then, Jamie got married and changed her last

name. She also moved to Los Angeles. So how would her new Data Object look like?

You're essentially setting up all your changed data values to their new values. If you wanted to know what these values are, they would give you their current values.

// Jamie Walker is a person

// She has:

// - a name: Jamie Walker (changed from Jamie Denise)

// - an age: 29 (was 19)

// - a City she lives in: LosAngeles (was NewYork)

Jamie-NAME = "Jamie Walker";

Jamie-AGE = 29;

Jamie-LOCATION= LosAngeles;

and yes, make sure even LosAngeles is defined.

// LosAngeles is a CITY

// LosAngeles has:

// - a name: "Los Angeles"

// - a Latitude and Longitude: 34.0500 and 118.2500

// - a Population count: 3884000

City LosAngeles = new City;

LosAngeles-NAME = "New York";

LosAngeles-LATITUDE = 34.0500;

LosAngeles-LONGITUDE = 118.2500;

LosAngeles-POPULATION = 3884000;

Keeping Track of your Defined Data

let’s recall the Clock from the previous chapter:

Integer HOUR = 20;

Integer MINUTE = 30;

At the moment, it’s definitely not 8:30 PM anymore; let’s say it’s 10 AM now.

How would the clock change? Easy:

HOUR = 10;

MINUTE = 0;

Then three and a half hours pass. How would the clock change? Again, easy.

HOUR = 13;

MINUTE = 30;

If you wanted the time afterwards, what would it be?

Not 8:30 PM, not 10 AM either. But 1:30 PM.

Here, you've been essentially setting up both your integers named HOUR and MINUTE to new values. If you wanted to know what these values are, they would give you their current values.

However, i**t's Important that you keep track of your changes. You MUST understand what the changes to your data have been - and you MUST determine whether or not those changes are what you want.**

We stress this because one of the many traits a programmer needs to have (and be good at) is managing what happens to your data.

If you don't believe us, wait until you have your tech interviews for a programming position you're applying for…

JAVA-03: Data Changes & Mutable State

Changing your Variables' Data Types in Java

You can't. Once you've set your variable's data type, DO NOT assign a different data type to it. Otherwise, you'll get an error when you try to compile or run your code.

On the other hand, there are ways in Java that let you turn one data type into another. But even then, variables will still expect to have the same data types you've defined them to have.

Changing Data in Java

Just like our pseudocode, you can set and reset most of your variables using the Equals (=) sign.

Keeping Track of your Data

It's important. Very important. You'll see why in this example…

EXAMPLE:

(get your IDE ready…)

Let’s say there’s a square playground in a park within your neighbourhood. The playground is 20 feet long and wide.

You’re babysitting your best friend’s 5-year old son. Let’s call him James. Kids in this playground has X-Y coordinates that tell you how far away they are from the Top-Left Corner of the playground (and there happens to be a bench that you’re sitting in)

```
// James is a 5-year old kid.
// Kids have:
// - an X (sideways) Coordinate (Integer, between 0 and 20)
// - a Y (up-down) Coordinate (Integer, between 0 and 20)
class Kid{
int x;
int y;
}
```

James is running around the playground. Since you’re babysitting, James is NOT allowed to set foot outside the playground.

You can’t see James (because you’re probably reading this instead) but you know what his X-Y coordinates are. Therefore, either X or Y coordinate cannot go below 0 or past 20.

First, let’s initialize James as a Kid, then give him a location. He’s at (12,3)

```
// James is a 5-year old kid with coordinates (12,3)
Kid James = new Kid();
James.x = 12;
James.y = 3;
```

And clearly, James is going to move around the playground randomly. In the past few seconds, James moved around like so:

- 5 feet to left
- 10 feet to right
- 2 feet to up
- 3 feet to right
- 4 feet down

- 12 feet left
- 5 feet up
- 2 feet down
- 5 feet right
- 3 feet up
- 5 feet right
- 2 feet down
- 4 feet right

Question: Did James leave the playground? (did either X or Y go below 0 or above 20?)

Let’s put James’ movements on Java code. Before that, recall James’ initial X-Y coordinates. We’ll also comment James’ current coordinates after his every move. Remember: moving left & up goes CLOSER to 0.

```
// James is a 5-year old kid with coordinates (12,3)
Kid James = new Kid();
James.x = 12;
James.y = 3;

// 5 feet to left
James.x -= 5; // James is at (7,3)
// 10 feet to right
James.x += 10; // James is at (17,3)
// 2 feet to up
James.y -= 2; // James is at (17,1)
// 3 feet to right
James.x += 3; // James is at (20,1)
// 4 feet down
James.y += 4; // James is at (20,5)
// 12 feet left
James.x -= 12; // James is at (8,1)
// 5 feet up
James.y -= 5; // James is at (8,-4): James is outside the Playground!!!
// 2 feet down
James.y += 2; // James is at (8,-2): James is still outside the Playground!!!
// 5 feet right
James.x += 5; // James is at (13,-2): James is still outside the Playground!!!
```

// 3 feet up

James.y -= 3; // James is at (13,-5): James is still outside the Playground!!!

// 5 feet right

James.x += 5; // James is at (18,-5): James is still outside the Playground!!!

// 2 feet down

James.y += 2; // James is at (18,-3): James is still outside the Playground!!!

// 4 feet right

James.x += 4; // James is at (22,-3): James is still outside the Playground!!!

So James has been outside the playground for quite a long time. He just ran past the closest edge and past your bench. If you were babysitting James, you'd be in big trouble…

In cases where you set clear boundaries for your data, yet you have some code that sets your data to go past those boundaries (and nothing is done about it), it might create problems for your code later on.

Now you understand how crucial it is to keep track of your data. There are many aspects of coding that depend on it, such as Debugging and making sure your code works the way you intend it to.

On another example, if you designed a video game and your characters go out of bounds, your video game wouldn’t be as good, won’t it?

JAVA Workshop #2

Go to an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or www.codechef.com/ide. (if you do, make sure to set your Programming Language to Java).

If you use one of the online IDE's, you should see something similar to this:

```
// ————————————————————

import java.util.*;

import java.lang.*;

import java.io.*;

// INSERT MORE IMPORTS HERE

// INSERT MORE CLASSES HERE


class (whatever class name)

{

public static void main(String args[])

{

// INSERT CODE HERE

}

}

// ————————————————————
```

World Cup of Football (or Soccer)

Let's say there's a Football (Soccer) game between All-Star National Teams of Two Countries.

Add this import on the top of your Java file:

```
import java.util.Random;
```

Then, insert this class into your Java Code:

```
// a Team has:
// - a Score (Integer, minimum 0)
class Team{
int score;
}
```

Afterwards, look in your code for a line named 'public static void main' (there should only be one line in your code with this)

Within the curly brackets of that 'public static void main' line, insert this following code:

```
// ————————
// The game Begins:
Team Country1 = new Team();
Country1.score = 0;
Team Country2 = new Team();
Country2.score = 0;

// It is now halftime,
// Each Team scores a random amount of points between 2 and 8
// FIRST HALF:
Country1.score += 2 + (int)(Math.random()*8);
Country2.score += 2 + (int)(Math.random()*8);
// ————————
```

Which team is winning? How would you know? And how would you show it?
How would you edit the above code to PRINT the scores for each team?
Also, how would you edit the code to add the SECOND half of the game?
Test and run your code in your IDE of choice.

Chapter 5a: Defining & Designing your Functions

Now we'll move on to the parts of programming where the magic happens.

Functions.

Where data structures are used to represent "things" in this universe, you define functions as the "actions" or verbs in this universe.

And you can make your functions do whatever you want/need it to do, as long as you know what you're doing. You can calculate math, write sentences for you, change or update data, sort out lists with tens of thousands of items, make websites for you, whatever you like. In reality, the possibilities can be endless.

But first let's understand the core parts of function design: its inputs, its output, its signature, its effects, and its functionality.

A Function’s Inputs

You can set your function to accept whatever data you need it to.

These are called a function’s arguments or parameters. Your function will use this incoming data to perform what you intend it to.

Or, on the other hand, you can also have a function NOT require any inputs. Your function will then perform what you’ve programmed it to, but it won’t need any incoming data.

For practice, let’s use comments to declare what inputs we want our function to have. Let’s say we want a name (a string) as an input

// INPUT: - a name (String)

A Function’s Output

Your function can also return a data value - based on whatever you want to set it to. You can then program your function to output that same data type.

Or, you can also have a function NOT return anything. You can then program your function to do what you intend it to do, but it won't return any data after it executes.

In most programming languages, functions only return ONE thing - whether it be a data value, an entire list, compound data, or more.

However, you must make sure your function outputs whatever you have set it to. Say, if you want your function to output a String, the very last line of that function MUST return a String data type. If you set your function to have no outputs, your function MUST NOT return any data types after it executes.

For practice, let's use comments to declare what outputs we want our function to have. We'll continue designing our function. We now have an input, now we want to have an output.

// INPUT: - a name (String)

// OUTPUT: - an ID (Number)

Defining what your Function will Do

Now we figure out what EFFECT our function will have once we run it.

This is your function’s main purpose - it’s the reason why you’re going to write these lines of code!

Your function’s effect will be whatever you intend it to do. Change data, create new data, calculate a few values together, whatever you want.

But isn’t an Output and Effect the same thing? Well, no.

There is a difference between a function’s OUTPUT and EFFECT. A function’s output is the data it returns, while a function’s effect is anything that the function does or anything the function’s action has affected.

For practice, let’s use comments to declare what effects we want our function to have. We’ll continue designing our function. We now have an input and output. Now we figure out what it does when we run it.

Let’s say we want it to come up with a random number. We first put in the comments of what we intend it to do

// INPUT: - a name (String)

// OUTPUT: - an ID (Number)

// EFFECT: generates a random number for a given name

Key Function Rule-of-Thumb:

Make sure your function only does the only one thing you want it to do. A function that does too many things will not only complicate your code and make it look bad, but it will cause headaches and frustration for programming teammates.

However, your function can include and call on many other functions to help process something. These are called Helper Functions (we’ll cover this later!)

A Function’s Signature

Here is when we start writing our function’s lines of code.

In most programming languages, a function’s signature defines a function’s name, inputs, outputs, and even particular traits it has.

Remember when we declared our Composite data? We first started out designing the name of our whole data structure, then we started designing what it consisted of.

For function signatures, we first code what its name is - as well as any inputs and outputs it has.

Now, let’s look back at the function we were designing for practice. We now know what it does, what it requires and what it returns.

For now, we’ll use Pseudocode to design our function’s Signature. Our signatures will then be structured in this form:

(OutputType) functionName(InputType inputName)

So our function’s signature will look like this:

// INPUT: - a name (String)

// OUTPUT: - an ID (Number)

// EFFECT: generates a random number for a given name

number createID(string name)

Implementing your Function

This can be the tricky part - unless you know exactly what you’re doing.

In designing functions, the last thing you do is to program your function’s actual functionality. You would now know what inputs & outputs it has, as well as what it’s trying to do. You’ve essentially planned what your function will do.

Now you’ll have your function do what you planned it to.

You program the functionality in the next few lines after your function’s signature.

In pseudocode, we'll use curly brackets ({ }) right after the function's signature to include its functionality code. Our functions will then be structured in this form:

(OutputType) functionName(InputType inputName) {

(your function's code)

return OutputType if any

}

Finally, let's look back at the function we were designing for practice. We are ready to finish it.

Let's say there's such thing as a function named randomNumber() that creates a random number for us.

// INPUT: - a name (String)

// OUTPUT: - an ID (Number)

// EFFECT: generates a random number for a given name

number createID(string name) {

randomNumber()

}

A Common Error in Function Design

But wait. The above code isn't going to work. Can you guess why?

Oh right.

In our signature, our function is supposed to RETURN a number. So in order for this function to work, it needs to actually return a number.

So we make the function RETURN whatever random number is generated by the inside function randomNumber()

// INPUT: - a name (String)

// OUTPUT: - an ID (Number)

// EFFECT: generates a random number for a given name

number createID(string name) {

return randomNumber()

}

and now, we finally complete our function - from design to code.

Calling your Function Procedure

To have your code execute whatever effect or procedure you've defined in your functions, you simply put your function name in your code - but in a particular way

Keep in mind that you can only place your function wherever its output data type is expected to be. The exception is when your function has no data output (returns void). You can place this function on lines by itself.

For example, take these two functions:

// INPUT: - none

// OUTPUT: - an ID (Number)

// EFFECT: ?????

number procedureA() {…}

// INPUT: - none

// OUTPUT: - none

// EFFECT: ?????

void procedureB() {…}

Procedure A outputs a number.

Place the function wherever the data type Number is expected:

Number x = procedureA();

*Number y = 10.213 * procedureA();*

Procedure A outputs nothing. Place the function on its own line of code.

procedureB();

If a function belongs to a data class, make sure to access that function from an instance of that data class:

ClassC instanceC = new ClassC;

instanceC.procedureC();

Chapter 5b: Matching Data with Functions

Here’s an inevitable truth when it comes to functions: they will almost always involve data in any way.

Later on, you’ll find that all sorts of different data structures will have at least one key function associated with it.

Also, some data structures, by default, will have associated go-to templates to use in programming. Remember this well; if you’re given a certain data structure to work with, you should already have the function structure you’ll need in mind.

Functions using Atomic Data

There’s usually no structure or template involved when dealing with Atomic Data.

Functions may have atomic data as inputs or outputs when necessary.

Here are some pseudocode to demonstrate:

// INPUT: - a Name (String)

// OUTPUT: - an ID (Number)

// EFFECT: ?????

number procedureA(String name) {… return ID}

// INPUT: - none

// OUTPUT: - none

// EFFECT: ?????

void procedureB() {…}

Also, Functions can even modify existing variables.

// Player1 Score, as an Integer

Integer Score = 0

// INPUT: - none

// OUTPUT: - none

// EFFECT: increments the score by one

void score1() { Score += 1}

Functions using Composite Data

The key thing to remember here is that, for every component a composite data structure has, its associated function will have a template that accesses and deals with each component (regardless of what data type each component is). Also, each component will be treated as whatever data type it is; a String treated as a String, composite data as composite data, and so on.

We demonstrate this in pseudocode:

```
// Structure of a Book:
CompositeStructure Book {
String AUTHOR
String TITLE
Integer PAGECOUNT
}

// INPUT: - a Book
// OUTPUT: - none
// EFFECT: ?????
void bookTemplateFunc(Book b) {
b.AUTHOR //do something
b.TITLE // do something
```

b.PAGECOUNT // do something

}

Here’s an example of a printing function, based on the above template:

// INPUT: - a Book

// OUTPUT: - none

// EFFECT: prints book details

void printDetails(Book b) {

printString(b.AUTHOR)

printString(b.TITLE)

printInteger(b.PAGECOUNT)

}

Methods: Functions for Object-Oriented Programming

In Object-oriented programming, data and procedures are bundled in data structures called classes.

Functions are called Methods and class variables are called Fields.

You can think of Methods within a class as ‘behaviours’ - or what actions an instance of that class can do.

To describe Class Methods in comments, simply list the behaviours it can do:

// A Space Invaders Tank Class can:

// - move left

// - move right

// - shoot a missile

// A Dog Class can:

// - walk

// - bark

// - sit

// - eat

Functions for Sequences

(You'll cover this in later chapters…)

Basically, data elements of the same type can be grouped together into sequenced collections. Examples can be lists and strings.

For these, functions process each element one by one until all data elements are covered.

Functions for more Sophisticated Data Structures

(You'll cover this in later chapters…)

JAVA-04: Function Structure

Functions in JAVA

The syntax structure in Java looks very similar to our pseudo code.

Java function structure looks like so:

<output's data type> functionName(<input data type> input1Name) {

// any code here,

return <data or variable with output's data type>;

}

Here are a few key notes in Java functions:

- Since Java is purely Object-Oriented Programming, Functions in Java are mainly called Methods. They belong within classes and they describe behaviors that the class does.
- You place your function code within the curly brackets of your function.

- If your Java function returns data, you put the word ‘return’, then a variable name or some data value you want to return. Once the function reaches the ‘return’ line, it will output whatever data you’ve set it to, then the function will finish running.
- If your Java function returns nothing (the output value is called “void”) you don’t have to put a return line.

Function Example

Here’s what the createID() function looks like in Java. We’ll also consider numbers as data type Double:

```
// INPUT: - a name (String)
// OUTPUT: - an ID (Number)
// EFFECT: generates a random number for a given name
double createID(String input) {
return rand();
}
```

JAVA Workshop #3

Designing & Calling Functions

First, go to an IDE of your choice. Online IDE's include Rextester (rextester.com), CodeChef (www.codechef.com/ide), and Ideone (https://ideone.com/).

On your Main Function, replace ALL code within it, then copy-paste all the code within the dotted lines below. Make sure it's between the curly brackets of the main() function.

```
public static void main(String args[]) {
// - - - - - - - - - - - - - - - - - - - - - - - - -
class calculator{
// INPUT: - two Integers
// OUTPUT: - a result(Integer)
// EFFECT: add two integers together & give result
__ add(__ a, __ b){
return a + b;
}
```

```
// INPUT: - two Integers
// OUTPUT: - a result(Integer)
// EFFECT: subtract 1st integer from 2nd & give result
__ subtract(__ a, __ b){
return ____;
}
};

// Your Bank Account is an Integer
int BANKACCOUNT;

// Create a Calculator Object named ‘c’
calculator c = new calculator();
// Income & Expenses, as Integers
int PAYCHEQUE = 6000;
int LIVINGEXP = 3000;
int FUNSTUFF = 1000;
int TRAVEL = 3000;

// How much would fun stuff and travel be together?
int FUNTRAVEL = c.add(___, ___);
```

System.out.println(“Fun Stuff & Travel Together: ” + FUNTRAVEL);

// Your Paycheque, after paying your living expenses?

// (HINT: call the calculator object c, then access one of its methods…)

BANKACCOUNT = c._____(_____, _____);

System.out.println(“Bank Account Balance, normal:” + BANKACCOUNT);

// Your Paycheque, after paying your living expenses AND fun stuff?

// (HINT: call the calculator object c, then access one of its methods…)

BANKACCOUNT = _____

BANKACCOUNT = _____

System.out.println(“Bank Account Balance, w/ Fun Stuff: ” + BANKACCOUNT);

// -

}

Before you begin, your code’s main function should look exactly to the code above.

Now, fill in the blanks. If you filled in the blanks with proper code, you should have the printed lines below:

Fun Stuff & Travel Together: 4000

Bank Account Balance, normal: 3000

Bank Account Balance, w/ Fun Stuff: 2000

Good luck!

Chapter 6: Intro to Designing Worlds & Simple Apps, PT1

At this point, you know how to interpret real-life objects as computer data representations, as well as interpreting actions & procedures as programming functions.

Now, you'll begin the design process. You start to describe virtual worlds and apps, almost always visually. You prepare to translate your ideas into code and data.

Just as if you were a carpenter thinking of how to build your house, you use the same approach towards developing apps and game worlds. You identify all the components required to build your project.

Simple App Design Process

There are three key things about your idea that you need to identify: the facts behind your idea, what remains constant, and what will change/vary.

The reason for this is to give you and other programmers as much control and stability over your data as possible. You would know what type of data you would be working with and if/how that data would or would not change. Experienced programmers can then check if a particular data structure or function is designed the way it was intended to be.

To guide you through this process, we'll describe the classic Snake game - and identify plenty of details about the game from a programmer's standpoint. It will be as if we're designing the game for the first time. As if we're in the 70's.

Identifying the FACTS

First, you need the facts. You need to describe what your idea is about, what's involved, etc. Describe as much detail as you can, including the environment and key figures you provide.

Example:

In a single game of Snake, there exists a snake with a head and body, as well as food items that appear randomly. The snake's head, all segments

of its body, and the food pieces have an X-Y coordinate and a visual representation (since they have similar traits, they can be grouped together as a game sprite). An X-Y coordinate represents a location within the playable zone: a rectangular area with a height and width. A score keeps track of how many food items have been eaten.

The player changes which direction the snake will travel to: up, down, left, and right. The game ends when the snake either hits a wall or runs into one of its body segments.

The snake grows by one body segment after the snake head “eats a food item” (appears on the same coordinate as a food item).

[food] + [head] [body] [body]

=

[head] [body] [body] [body]

Identifying what's CONSTANT

Second, you need to identify what elements in your world or app remain consistent throughout the programming. Also, identify what will exist in your idea unconditionally.

Example:

The food items, snake head, and body segments have consistent images.

The playing board has a fixed width and height.

Identifying what CHANGES/VARIES

Third, identify what will change or vary when the program runs.

Example:

The food items, snake head, and body segments all have varying X-Y coordinates throughout the game. The number of body segments vary, from the initial number of 2, all the way to as much as possible.

Turning your ideas into code

Now, you take note of all the facts, descriptions, and ideas you've come up with. You'll be making data representations of them.

Early in the book, there is a great reason why we've used comments to describe all the information we are going to create. It's because they help select the best possible data representation for each pint of information you have. And after you've described what your app or world will be about, you'll start developing that digital world first by using comments.

Example:

For each fact and idea about the Snake game that we've come up with, we'll use comments to hint how they should be represented by data.

// A Game Sprite has:

// - an X-coordinate (Integer)

// - a Y-coordinate (Integer)

// - an image to represent itself (choose your visual representation)

// Snake Heads, Snake Body Segments, and Food Pieces

// are each represented by Game Sprites

// An EntireSnake contains:

// - a Snake Head (GameSprite)

// - a list of Snake Body Segments (list of GameSprites)

// A FullSnake can:

// - eat food

// - grow

// - move on the board

// The Playable Board has:

// - a fixed length (Integer)

// - a fixed width (Integer)

// A Game of Snake has:

// - a playable board (Playable Board Class)

// - food items (list of Food)

// - the Snake (EntireSnake)

// - a Score Count (Integer)

// A Game of Snake can:

// - start a new game

// - end the game (as a loss or win)

// - update a game in progress

// - update the score

// - move the snake on keypresses

// - create or delete food items

Notice how there is no actual code written yet; only comments.

The beauty of this process is that, given the facts and ideas (as well as their data representations), we can start creating the code in any language we want. We now know what will be data structures, classes, integers, and so on.

We do need to go over more tools, so we will continue with design later on.

Meanwhile, in the next big workshop, you will be converting idea comments into actual code. Good luck and have fun!

JAVA BIG Workshop A

Game Design: the Data & Functions

First, go to an IDE of your choice. Online IDE's include Rextester (rextester.com), CodeChef (www.codechef.com/ide), and Ideone (https://ideone.com/).

You're going to practice designing Data Structures and Functions - as if you were designing an app yourself.

All you'll be given is a set of comments describing the data objects within a very, very simple video game. If you can, you may continue developing it into a full-blown game.

Copy and paste all the comment code below, then start writing code for the data definitions.

As you improve your programming and learn more tricks over time, you can revisit this workshop and re-create it using your new skills.

For example: for the functions and class methods, they might need more intermediate functionality. So you can come back to them later - after going through the necessary chapters.

If you feel like you want to create or remove new Fields, Classes, or Methods, feel free to do so.

Now, let’s move forward.

The game we’ll be designing is…

PONG!

Good luck!

```
// - - - - - - - - - - - - - - - - - - - - - - - - -

// A Game Sprite Class has:
// - a Width (Integer)
// - a Height (Integer)
// - an X-coordinate (Integer)
// - a Y-coordinate (Integer)


// A Ball Class has:
// - all properties of a Game Sprite
// - either an UP or DOWN direction (String)
// - either a LEFT or RIGHT direction (String)
// A Ball Class can:
// - move in all 4 diagonal directions
// - bounce off Paddles or the Game Board Up/Down Walls

// A Paddle Class has:
// - all properties of a Game Sprite
// A Paddle Class can:
// - move up & move down
```

// A Player Class has:

// - a Paddle

// - a Score (Integer)

// A Player Class Can:

// - Score a Goal

// a Game Board has:

// - a Width (Integer)

// - a Height (Integer)

// - two X-Coordinates for Player-side Edges (Integers, set to 0 and Width)

// - two Y-Coordinates for Up-Down Walls (Integers, set to 0 and Height)

// A Game Class has:

// - two Paddles

// - a Ball

// - a Game Board

// - a Player on the Left

// - a Player on the Right

// - a Game Speed (Strings: "SLOW" "MEDIUM", or "FAST")

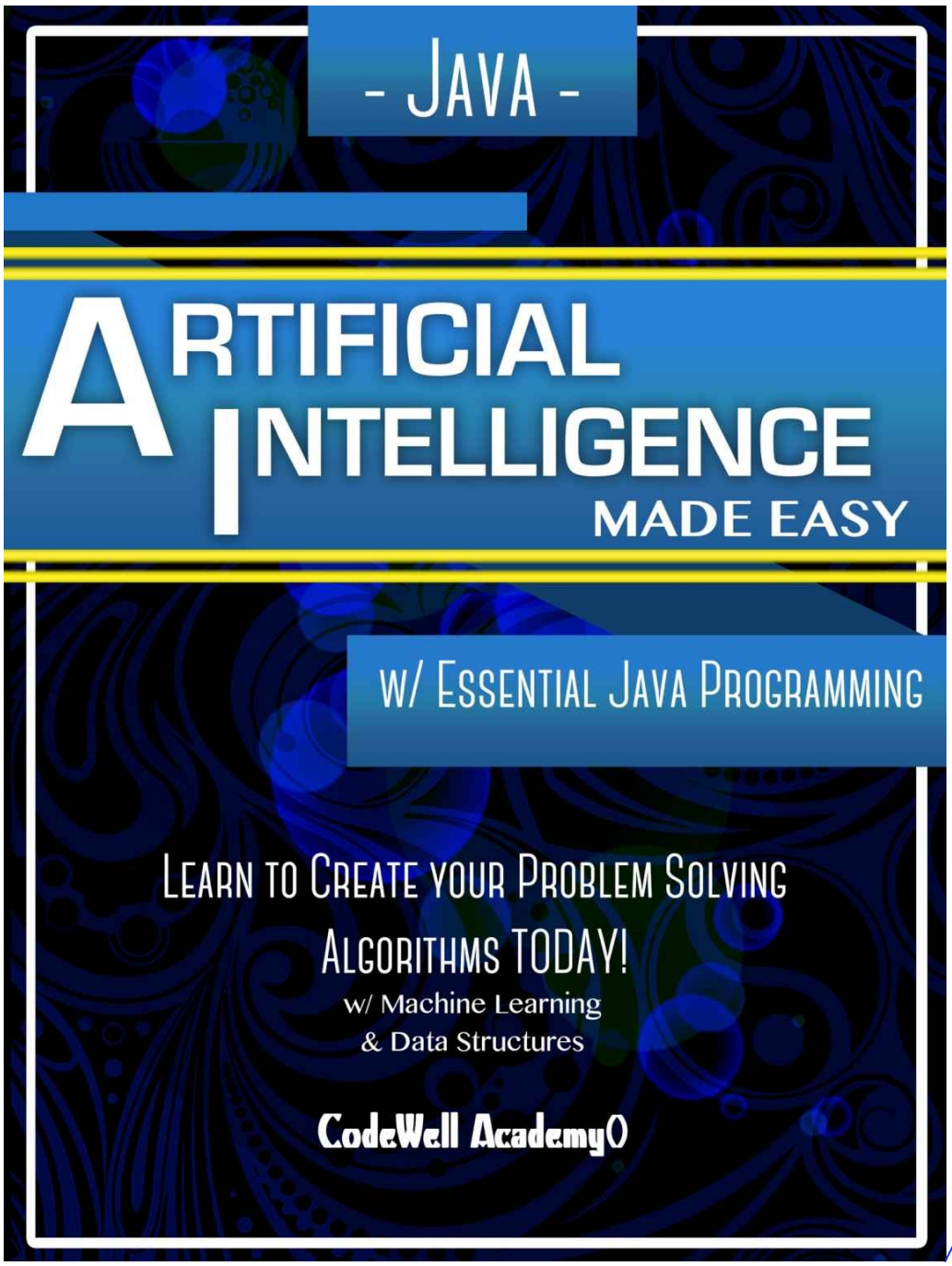
// - a Timer (a Number, Integer, or any other Data Type you like)

// When Created, A Game Instance also creates:

// - a Paddle for the Left Player

// - a Left Player

// - a Right Player



// - a Paddle for the Right Player

// - a Ball, moving at a given random direction

// A Game Class can:

// - manage a goal scored by either player

// - end the game

// - declare a winning player

// -

CodeWell Academy() and R.M.Z.
present:

Artificial Intelligence in Java
Made Easy
w/ Essential Java Programming

1st Edition

Learn to Create your * Problem Solving * Algorithms!
TODAY! w/ Algorithms & Data Structures

Artificial Intelligence Series

INCLUDES BONUS: Easiest Way to Learn Java

Preface: JAVA as Artificial Intelligence

========== ======

There’s no doubt in our minds that Artificial Intelligence is perhaps one of the greatest tools created by Humans today. From drones to smartphone apps, from video games to robotics, the extent of their applicability will be endless for the years to come.

Thus, it’s best to start learning about them and, as programmers, start applying their functionality to our projects.

How you’ll progress through this book

We’ll start by going through the fundamental Search Algorithm. This is one of the most important tools in Artificial Intelligence; as it gives the computing system the ability to think and generate a solution. Then, we start to analyze its variants - including how they work, as well as their strengths and weakness. We examine the best scenarios to apply the variants, and even implement some of them in Java code.

A Quick Start

Included is a quick kit to cover Basic Java Programming. You'll find what you need to get a good start in programming Java. Then, you'll be prepared to use the many tools and components of AI throughout the book.

========== ======

Introduction

==== ==== ==== ==== ====

Logic. Rationality. Reasoning. Thought. Analysis. Calculation. Decision-making.

All this is within the mind of a human being, correct? Humanity has been blessed with the ability to think and act so intelligently.

Then came Machine. Humanity has also blessed it the gift of intelligence.

And in today’s world, you can see firsthand what an intelligent mind can do for you; carry a conversation, give you directions to a certain location, play a video game as an opponent, and so on.

In essence, only our imaginations will limit us from what’s truly possible

An Artificial Intelligence Agent

In terms of Artificial Intelligence, an agent can be anything that, given an environment to focus on, can think intelligently and act independently. It can continue observing and learning through experience. It can calculate and independently decide the best course of action, whether it has perfect knowledge of the situation or just a part of it. It can also take note and adapt to a changing environment.

So you might wonder, how has mankind ever developed something so complex?

Well, it’s not as complex as you think.

If you understand the process of how a computer can observe, learn, and expand its knowledge - and how it can take all this information and come up with an ideal solution or decision - then an artificially created mind won’t be as complex as you think.

Sometimes, it can take as little as a few lines of code to have a computer come up with solutions for you. Sometimes it can take hundreds. Sometimes, thousands.

Chapter 1: Algorithms: The Essentials

==== ==== ==== ==== ====

In essence, how an AI agent will contemplate, process, rationalize, apply logic, & ultimately generate solutions will mainly be through the use of algorithms.

If you’re new to programming, don’t be intimidated. An algorithm is essentially a procedure to handle data. As long as you understand how a certain algorithm processes its data, you’ll be fine.

Algorithm Traits

First, you’ll want your algorithms to satisfy four key factors:

- Completeness

- Optimization

- Time Complexity

- Space Complexity

Now we'll go through each of these and explain them all. Afterwards, you'll explore some algorithm ideas and determine how they fit in to each of these factors.

Completeness

If an algorithm is guaranteed to find at least one existing solution or conclusion within a certain time frame, we can say that an algorithm is complete.

Optimization

If an algorithm finds a solution and guarantees that it is the optimal one, then that algorithm is considered optimal.

Time Complexity

For an algorithm, this is an expression for the longest possible time it will take to complete. In other words, the worst-case scenario when it runs and finds a suitable solution.

Space Complexity

This expression is similar to Time Complexity, but instead it represents the maximum amount of memory the algorithm may use in order to find a solution. This is also considered the worst-case scenario.

Your Ideal Algorithm

After discussing the traits your algorithm can have, you'll get an idea in what to look for when creating an AI algorithm. You want to design yours to find at least one solution (completeness), and the best solution it can create given data it has (optimization) while using up as little computational effort as you can (Time & Space Complexity)

Chapter 2: How to Create a Problem-Solving AI

==== ==== ==== ==== ====

Let’s start developing our Search algorithm: an automatic problem solver. We’ll have a general overview of it in Pseudocode. Then you’ll get to code and run it on your own, with this book’s primary programming language.

Abstract Search Algorithm

In its most basic procedure, a search algorithm will have a default condition and a goal condition. It will then evaluate each option it can take, starting from the default condition, step-by-step, until it eventually finds a full set of options to achieve the goal condition.

The General Frontier Search Algorithm:

The Frontier Search algorithm follows the same procedure as above. Given a start node, goal nodes, and an entire network, it will incrementally assess and explore pathways from the start node until it reaches the goal node.

The Frontier is simply a list of paths to be checked. The Frontier Search algorithm will keep adding paths to the Frontier until it either finds a solution or has explored the entire network

For example, this is just like giving a Search Algorithm a map of your local city, your current location, and a restaurant you're about to go to. That search algorithm will give you directions to get there.

The standard data structure to use a Search Algorithm with is a Network of interconnected Nodes. Each node contains an amount of data and a list of connected nodes:

// A Node has:

// - its data (any data type you want)

// - a set of connected nodes

class Node

<some data type>: contents

Array of Nodes: connected

The Frontier Search algorithm also uses Paths: a list of connected Nodes, with the first node as the starting point:

// A Path has:

// - a List of Nodes

class Path

Array of Paths: contents

And finally, here is a generalized algorithm for Frontier Search:

INPUT:

- a Start Node (can be a class method in OOP)

- a graph network (only requires start node to have a network)

- a goal-checking procedure OR a solution query

OUTPUT:

- a Path from start to Goal (a List of Nodes)

- return FALSE or NULL if no paths found (wherever applicable)

EFFECT:

Frontier Search Algorithm: Returns a set of nodes that lead from the input Node to a solution node if found

PROCEDURE:

- frontier:= {new array of Nodes}

- create a new Path and put the Start node in it

- put the new Path into the frontier

While frontier is not empty {

- select and remove a Path <s0, s1,….,sk> from frontier;

If node (sk) is a goal, return selected Path <s0, s1, ….,sk>;

Else:

For every connected node of end node sk:

- Make a copy of the selected Path

- Add connected node of sk onto path copy

- add copied Path <s0, s1,….,sk, s> to frontier;

}

- indicate ‘NO SOLUTION’ if frontier empties

Further Search Strategies

This will be covered later on, but how the algorithm picks a Path from the Frontier will determine how the Search Algorithm works.

For now, let’s apply the Frontier Search Algorithm.

JAVA 02a: Fundamental Frontier Search Algorithm

For the procedure below, select an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

========== ======

Below are the fundamental parts to a Frontier Search algorithm: the major algorithm and its data structures.

First, we start with the Data Structures. We only need to use two essential classes: a network node and a path. A Path contains is what the algorithm uses to store connected, sequenced nodes. It will also be the output type for the algorithm.

```
// A Path has:
// - A List of Nodes
// (can be modified to include more Methods/Fields)
class Path {
ArrayList<Node> contents = new ArrayList<Node>();
}
```

Nodes will be the main data structure the Algorithm will operate through; the algorithm will search through the node and its connected nodes for a solution:

```
// A Node has:
// - Some Contents (Data type of your choice)
// - A List of other connected Nodes
// It can:
// - Search all its descendant nodes to find a solution
// (Our Search Algorithm as a Class Method)
class Node {
<choose a data type> contents;
ArrayList<Node> children;

// CONSTRUCTOR:
public Node(String c) {
this.contents = c;
this.children = new ArrayList<Node>();
}
}
```

Next are two Helper Functions you'll need. This first function helps the algorithm pick a path to check from a list:

```
/*
// HELPER FUNCTION #1:
// INPUT: a List of Paths
// OUTPUT: a Single Path
// EFFECT: based on positioning of your choice:
// - Select & remove a path
// - return that path
// NOTE: you can modify the position assignment to change the Search Strategy
*/
private Path pickPath(ArrayList<Path> f) {
int position = 0;
Path ret = f.get(position);
f.remove(position);
return ret;
}
```

This second function checks if the last node in the path is a solution. One of the function inputs supplies the solution:

```
/*
// HELPER FUNCTION #2:
// INPUTs:
// - a Path
// - Node contents that have a solution
// <same data type as Node's container>
// OUTPUT: boolean
// EFFECT: outputs True if path contains a Goal
*/
private boolean hasGoal(<data type> s, Path p) {
for (Node n: p.contents) {
if (n.contents == s) return true;
}
return false;
}
```

And finally, the Search Algorithm. The pseudocode is attached to the lines as comments so you can see how the procedure works. The algorithm also uses both helper functions described earlier.

```
/*
// MAIN ALGORITHM:
// INPUT:
// - a goal query
// <has same data type as node contents>
// - a Starting Node
// OUTPUT:
// - a Path from start to Goal (a List of Nodes)
// (multiple output types not acceptable in Java;
// empty Path as output if no solution found)
// EFFECT:
// Frontier Search Algorithm: Returns a set of
// nodes that lead from the input Node to a solution node if found
*/
public Path search(<data type> query, Node n1) {
// - frontier:= {new array of Paths}
ArrayList<Path> frontier = new ArrayList<Path>();

// - create a new Path and put the Start node in it
Path p = new Path();
p.contents.add(n1);
// - put the new Path into the frontier
frontier.add(p);
while (!frontier.isEmpty()) {
// - select and remove a Path <s0, s1,….,sk> from frontier;
// (use helper function pickPath() )
Path pick = pickPath(frontier);
// If node (sk) is a goal, return selected Path
if (hasGoal(query, pick)) {
```

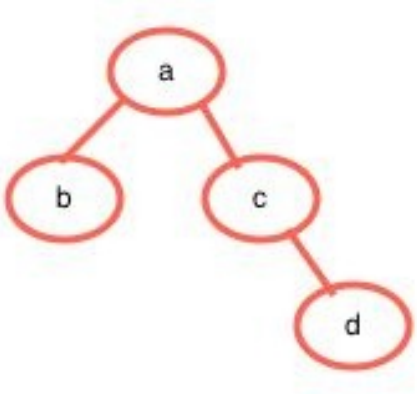
```
return pick;
}
else {
// Otherwise, for every connected node of end node sk:
// 1. Make a copy of the selected Path
// 2. Add connected node of sk onto path copy
// 3. add copied Path <s0, s1,….,sk, s> to frontier;
int size = pick.contents.size();
Node last = pick.contents.get(size - 1);
for (Node n: last.children) {

Path toAdd = new Path();
toAdd.contents.addAll(pick.contents);
toAdd.contents.add(n);
frontier.add(toAdd);
}

}
}
// - indicate ‘NO SOLUTION’ if frontier empties
// (empty path as ‘NO SOLUTION’ here)
return new Path();
}
```

JAVA 02b: Using Frontier Search

For the procedure below, select an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

========== ======

Here, we implement the Frontier Search Algorithm step-by-step onto a simple node network:



Each circle a, b, c, and d each represent interconnected nodes, which will only have their respective letters as contents.

We ask the algorithm if there is a path to a particular letter from a starting point.

The Search Algorithm would have a letter as an input. It would then check the nodes row-by-row until it either finds a suitable path from the input node to that letter - or notify you that a path couldn’t be found.

So we now know what to expect. If we have, say ,’d’ as the input, the algorithm is supposed to find it, and output the node path a->c->d. If we have ‘g’ or some other irrelevant letter as an input, the algorithm will say that it’s not found.

Artificial Intelligence will heavily rely on Search Algorithms to come up with solutions and best decisions for its given situations. We will explore more about this later on.

Meanwhile, let’s start building our algorithm.

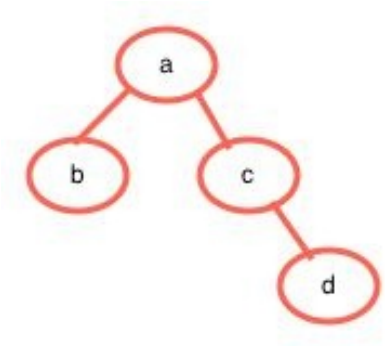
Step 1: Understand & Create the Node Structure

First, we have to build the underlying data structure for our Network. Our Nodes contain the data type we want to use, as well as a list of its connected nodes. Since we’re only using letters, our Node data type can be String.

Copy the class code below to your IDE. We'll build on from here.

```
// A Node has:
// - Some Contents (Data type of your choice)
// - A List of other connected Nodes
// It can:
// - Search all its descendant nodes to find a solution
// (Our Search Algorithm as a Class Method)
class Node {
String contents;
ArrayList<Node> children;

// CONSTRUCTOR:
public Node(String c) {
this.contents = c;
this.children = new ArrayList<Node>();
}
}
```

Afterwards, we need the node network.



So it looks like ‘d’ is a connected to ‘c’, while ‘b’ and ‘c’ are connected to ‘a’. We’ll be re-creating this network in code.

Copy the code below and place it within the main() method:

```
// Creates & connects nodes a, b, c, d
Node a = new Node(“a”);
Node b = new Node(“b”);
Node c = new Node(“c”);
Node d = new Node(“d”);
a.children.add(b);
a.children.add(c);
c.children.add(d);
```

Next, we need to design the Path.

Step 2: Create the Path Structure

The Path will contain an ordered list of Nodes. Each node will be connected to the ones next to it, while the first node in the Path is the Start node.

Copy the class code below to your IDE, just before where you put the Node class.

```
// A Path has:
// - A List of Nodes
// (can be modified to include more Methods/Fields)
class Path {
ArrayList<Node> contents = new ArrayList<Node>();
}
```

Step 3: Start coding the Algorithm

(Before we move forward, it’s highly recommended to review the Algorithm procedure in the past few chapters.)

Here, our search algorithm will be a method from the Node class. We can simply access it from one of the nodes created. And since we're doing it this way, we won't have to use a Node object as an input.

Since we are checking for letters, the search query will be a String - the same data type for the Node containers.

Place the Main Algorithm below within the Node Class you've created. Remember to have it within the curly brackets '{ }'.

```
/*
// MAIN ALGORITHM:
// INPUT:
// - a goal query
// <has same data type as node contents>
//
// (Start Node & its graph network accessed thru this method)
// OUTPUT:
// - a Path from start to Goal (a List of Nodes)
// (multiple output types not acceptable in Java;
// empty Path as output if no solution found)
```

```
// EFFECT:
// Frontier Search Algorithm: Returns a set of
// nodes that lead from the input Node to a solution node if found
*/
public Path search(String query) {
// - frontier:= {new array of Nodes}
ArrayList<Path> frontier = new ArrayList<Path>();

// - create a new Path and put the Start node in it
Path p = new Path();
p.contents.add(this);
// - put the new Path into the frontier
frontier.add(p);
while (!frontier.isEmpty()) {
// - select and remove a Path <s0, s1,….,sk> from frontier;
// (use helper function pickPath() )
Path pick = pickPath(frontier);
// If node (sk) is a goal, return selected Path
if (hasGoal(query, pick)) {
return pick;
}
else {
// For every connected node of end node sk:
// - Make a copy of the selected Path
// - Add connected node of sk onto path copy
// - add copied Path <s0, s1,….,sk, s> to frontier;
int size = pick.contents.size();
Node last = pick.contents.get(size - 1);
for (Node n: last.children) {
Path toAdd = new Path();
toAdd.contents.addAll(pick.contents);
```

```
toAdd.contents.add(n);
frontier.add(toAdd);
}
}
}
// - indicate ‘NO SOLUTION’ if frontier empties
// (empty path as ‘NO SOLUTION’ here)
return new Path();
}
```

It’s considered a good programming practice to simplify what a function does. So instead of our algorithm function doing a lot of different things, it will call specialized helper functions to simplify the workload. This also makes it easier for programmers to check, edit, debug, and modify the code.

Step 4: Add the Frontier Path Picker Function

The path picking function for the Frontier Search algorithm is a customizable one in it own right, as modifying this function will affect the search strategy. We’ll go over more search strategies later on. For now, just ensure that the function selects, removes, and outputs the path correctly.

Place this helper function within the same Node class you placed the Search Algorithm.

```
/*
// HELPER FUNCTION #1:
// INPUT: a List of Paths
// OUTPUT: a Single Path
// EFFECT: based on positioning of your choice:
// - Select & remove a path
// - return that path
// NOTE: you can modify the position assignment to change the Search Strategy
*/
private Path pickPath(ArrayList<Path> f) {
int position = 0;
Path ret = f.get(position);
f.remove(position);
return ret;
}
```

Step 5: Add the Goal-checking Function

This function helps the algorithm check a path for solutions. It will check all the nodes in a path to see if one of them has the contents the algorithm is looking for.

Since both our node contents and search query are Strings, we’ll use that data type as a function input.

Like the other helper function, place this one within the same Node class you placed the Search Algorithm.

```
/*
// HELPER FUNCTION #2:
// INPUTs:
// - a Path
// - Node contents that have a solution
// <same data type as Node's container>
// OUTPUT: boolean
// EFFECT: outputs True if path contains a Goal
*/
private boolean hasGoal(String s, Path p) {
for (Node n: p.contents) {
if (n.contents == s) return true;
}
return false;
}
```

Algorithm Testing

And finally, we test our algorithm. There's at least three major scenarios to think of: searching for the starting node's letter; searching for a letter further down the network; and searching for a letter that's not in the network. For each of these times, we want the algorithm to run through the scenario properly.

Within your Main Class, and just above your main() method, insert this testing method:

```
static String printer(Path p) {
if (p.contents.isEmpty()) return “NOTE: No Solution Found”;
else {
// System.out.println(“FOUND A SOLUTION!”);
String s = “Solution Found! Path: “;
for (int i = 0; i<p.contents.size(); i++) {
s += p.contents.get(i).contents + “, “;
}
return s;
}
}
```

Afterwards, we’ll create a few paths generated from our Search Algorithm. Place the code below just after your node network from Step 1. They should be inside the main() method.

```
// test search()
Path pa = a.search("a");
Path pc = a.search("c");
Path pd = a.search("d");
Path pg = a.search("g");
```

A search for the starting node's letter should output a Path with just the starting node. So Path 'pa' should have a path with only node 'a' in it. If everything was done right, the code below should print out, "Solution Found! Path: a, ".

```
System.out.println(printer(pa));
```

A search for a letter somewhere down the network should output a Path with a sequenced list of Nodes. Paths 'pc' and 'pd' should have nodes 'a, c' and 'a, c, d' respectfully. If the codes below run, then they should print out their respective paths:

```
System.out.println(printer(pc));
System.out.println(printer(pd));
```

A search for a letter that isn't in the network should output an empty path, according to our current Search algorithm. So 'pg' should be an empty path.

If you run the line of code below, it should notify you that a solution isn’t found (‘g’ currently isn’t in the network)

System.out.println(printer(pg));

And there we have it. A successfully operating Frontier Search algorithm.

Chapter 3: Search Strategies

==== ==== ==== ==== ====

As mentioned earlier, the way the search algorithm picks a path from the frontier will determine the overall algorithm search strategy.

Now that we've developed our search algorithm, we can now modify it to suit any situation that arises.

Below are the four main ways that the search algorithm will pick a path to explore. The path picked from the frontier is either:

- the most recently added (Stack)

- the least recently added (Queue)

- the one with the least cost (Priority Queue)

- the one with the most value (Priority Queue)

We'll explore and analyze each strategy and implement them with our search algorithm.

Chapter 3.1: Depth-First Search

==== ==== ==== ==== ====

In Depth-First Search, the algorithm treats the Frontier Options as a Stack. Therefore, if the algorithm has a list of unexplored options it has yet to examine, it will explore the options and sub-options first.

Use Depth-First Search When:

- You expect long path lengths; in other words, the solutions will have long sets of options to get there
- You don’t expect any nodes that are subnodes to each other)
- You don’t have much space available

Don’t use Depth-First Search When:

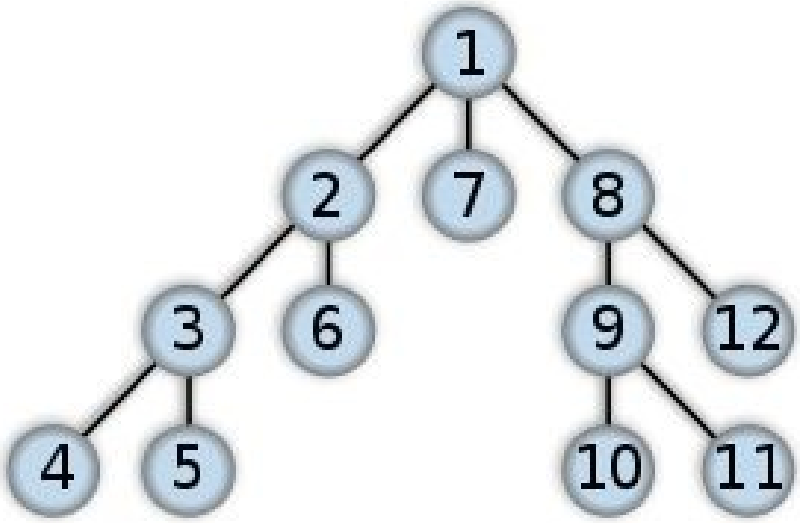
• The three looks fairly shallow. In other words, there aren’t many levels of Option nodes in the tree

• If having the best possible solution is very important

Example: The Depth-First Search Algorithm

Consider the graph below. If all these nodes are placed on a to-do list for the algorithm, the last node added to the frontier would be processed first.

Node #1's options are added to the frontier: #2, #7, and #8. If Node #2 was added last, it would be processed first. So Nodes #3 and #6 would be added. If #3 was added last, then it would be processed first - so that means adding #4 and #5 to the frontier. The node depths are explored first - hence, why it's called DEPTH-FIRST search.



Algorithm Analysis: Depth-First Search

Is it Complete?

Sort of. Why? If there are any loops or cycles in the graph (meaning one of the end nodes link up to the beginning nodes, thus creating a loop) the algorithm might be stuck. It will keep exploring the end node, the beginning node, the path between them, the end node again, and so on and so on - probably in a sort of infinite loop.

On the other hand, if there are no loops or cycles in the node network, this algorithm is complete.

Is it Optimal?

No. If it gives off the first solution it encounters, it may not necessarily be the best one. There may be better solutions that have yet to be encountered before the algorithm gets to it.

What is its Time Complexity?

O(b^m)

Meaning, at the worst-case scenario, the algorithm will explore every node and reach the furthest tree depth. For example, if a node in a tree has up to 2 options and the entire tree can be up to 4 levels deep, the worst-case complexity will be 2x2x2x2 = 16 nodes possibly explored.

What is its Space Complexity?

O(b*m)

Meaning, at the worst-case scenario, a path for unexplored nodes will be stored in memory for every node explored.

The longest path possible is the furthest tree depth. Also, every node has a maximum amount of nodes it can explore.

For example, if a tree will have up to 2 options per node and the tree can be 4 levels deep, then the algorithm will store 2x4 = 8 units of memory

Chapter 3.2: Breadth-First Search

==== ==== ==== ==== ====

While Depth-first search has a Stack for the frontier, Breadth-first search has a Queue. In other words, if the algorithm has a list of options to explore, it will select the earliest added options and sub-options.

Use Breadth-First Search When:

• you don’t have to worry about memory space

• you NEED a solution with the least amount of options chosen

• there are some options that can be explored that don’t need depth

Don’t use Breadth-First Search When:

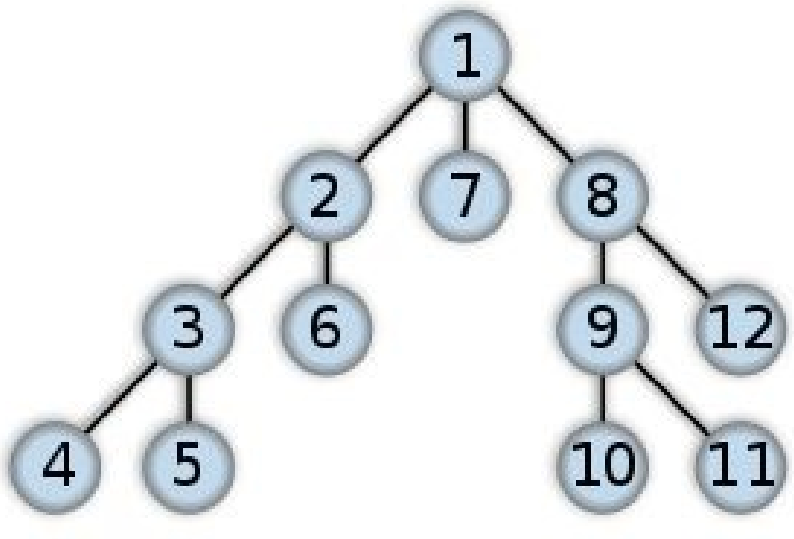
• solutions tend to need a lot of options chosen (i.e. they’re deep into the tree)

• you have a limited amount of space

• There’s a high branching factor (nodes with many subnodes/options)

Example: The Breadth-First Search Algorithm

Consider the graph below. If all these nodes are placed on a to-do list for the algorithm, Node #1 would be processed first, then it's sub-nodes #8, #7, #2 would be added to the frontier in that order. Since Node #8 was entered first, it will be processed first. So Nodes #9 and #12 are added to the frontier. Then Node #7 is processed. Then Node #2, which adds Nodes #6 and #3 to the frontier. Then node #12 is processed, and so on. Overall, the algorithm will explore all nodes per level first - hence why it's called BREADTH-FIRST search.



Algorithm Analysis: Breadth-First Search

Is it Complete?

Yes, as long as there's a limited number of subnodes. When there are nodes that are children to each other (for example: Node X is a sub-node to Node Y, and vice-versa), this would normally create a loop for Depth-first search. Node X would be added to the frontier, processed, then Node Y would be added and processed, then Node X is added, and so on - in an infinite loop. This won't happen in BFS. If Node X was added, all other nodes in the frontier would have been processed first.

However, if a tree has infinite subnodes per node, then BFS certainly won't stop. There will be just too many nodes to explore.

Hence, as long as there's a finite number of subnodes per node, you can guarantee that BFS will not loop indefinitely.

Is it Optimal?

Possibly. Because BFS is likely to find the solutions with the least number of options/steps, there is a chance the solution will be optimal.

What is its Time Complexity?

O(b^m)

Just like DFS, BFS will, at the worst-case scenario, explore every node and reach the furthest tree depth. For example, if a node in a tree has up to 3 options and the entire tree can be up to 4 levels deep, the worst-case complexity will be 3x3x3x3 = 81 nodes possibly explored.

What is its Space Complexity?

O(b^m)

At the worst-case scenario, BFS will explore every single node in the tree.

For example, if a tree will have up to 4 options per node and the tree can be 4 levels deep, then the algorithm will store 4^4 = 256 units of memory if it explores every node in that tree.

JAVA 03: Frontier Search as DFS and BFS

For the procedure below, select an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

========== ======

When implementing Frontier Search, the algorithm will actually, by default, either be Depth-First Search or Breadth-First Search. This will depend on one key factor, as we will demonstrate below.

First, let's recall the path picker from earlier:

```
/*
// HELPER FUNCTION #1:
// INPUT: a List of Paths
// OUTPUT: a Single Path
// EFFECT: based on positioning of your choice:
// - Select & remove a path
// - return that path
// NOTE: you can modify the position assignment to change the Search Strategy
*/
```

```
private Path pickPath(ArrayList<Path> f) {
int position = 0;
Path ret = f.get(position);
f.remove(position);
return ret;
}
```

Note how the path picker works. The output path, chosen from the input array (the frontier), is based on an index position. That index is set on the first line of the procedure:

```
int position = 0;
```

When items are entered into an array, they are sent to the end of the array like so:

[0: a][1: b][2: c] <- inserting [d]

[0: a][1: b][2: c][3: d]

So setting the position to 0 means that the front (earliest) path on the frontier is selected. In other words, the path picker will treat the frontier as a queue. Therefore it will be BFS.

int position = 0;

Before path picker call: [0: a][1: b][2: c][3: d]

After path picker call: [0: a][1: b][2: c] Selected For Processing: [d]

Otherwise, setting the position to the back (latest) path will need this line instead:

int position = f.size()-1;

The line above will set the position to the latest path added to the frontier. In other words, the path picker will treat the frontier as a stack. Therefore, it will be DFS.

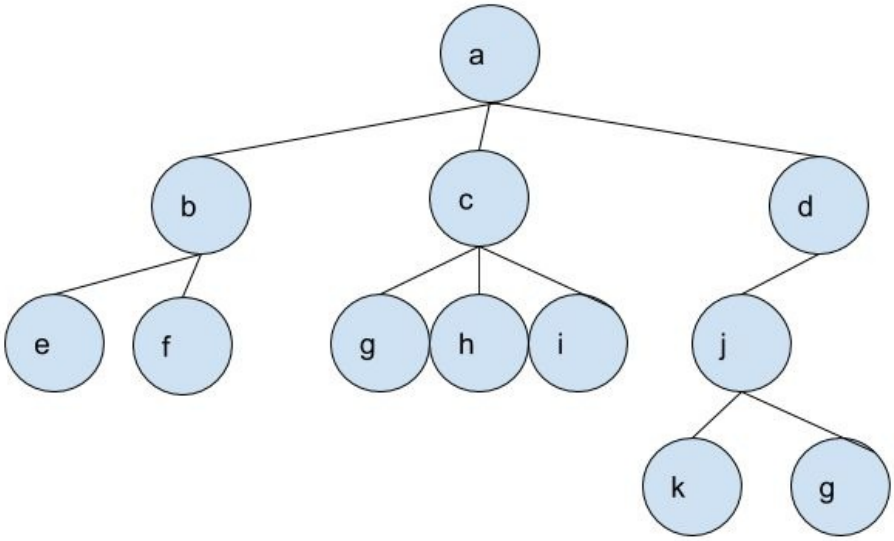
int position = f.size()-1;

Before path picker call: [0: a][1: b][2: c][3: d]

After path picker call: [0: b][1: c][2: d] Selected For Processing: [a]

Now, let’s test everything we know so far on the below graph.

Bigger Search Graph:



(The code for the graph above is found on ARCHIVE A02)

Place the code above into your Main() method to create the graph.

We start with the completed Node and Path classes from the previous chapter. If you need to copy the Search Algorithm, Nodes, and Paths, see ARCHIVE A01.

We will then modify the pickPath() method to be either DFS and BFS.

Step 1:

We'll modify our pickPath() in the code below:

```
/*
// HELPER FUNCTION #1: (Modified)
*/
private Path pickPath(ArrayList<Path> f) {
int position;
// Breadth-First: uncomment line below to use
// position = 0;
// Depth-First: uncomment line below to use
// position = f.size()-1;
Path ret = f.get(position);
f.remove(position);
return ret;
}
```

Uncomment either of the lines above to set the position.

Step 2:

Next, we'll run and test the algorithm.

Before we do this, make sure you have the printer() method in your Main() method. You can find this from either JAVA-02 or ARCHIVE A-01.

Moving on, simply add these two lines within your Main() method, just after the code for creating the search graph:

Path pg = a.search("g");

System.out.println(printer(pg));

There are two nodes that have "g" as their content. The algorithm will output either one as the solution depending on which search strategy you use.

If you set your pickPath() to DFS, the output lines should be:

Solution Found! Path: a, d, j, g,

Here's what happened after the algorithm processed Node a:

- the algorithm added Nodes b, c, and d into the frontier

- Node d was added most recently; so it'll be processed first

- Its subnode j is added

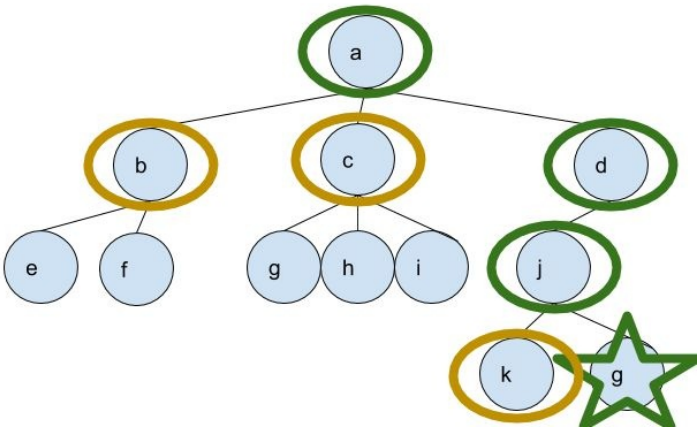
- Node j is added most recently; it'll be processed first

- Nodes k and g2 are added

- Node g2 is added most recently; it'll be processed first

- Node g2 is a goal. So a path with it and all its ancestor nodes is the solution path.

And if you look at the nodes carefully, you'll notice that the algorithm went "Depth-First":



Otherwise, if you set it to BFS, the output lines should be:

Solution Found! Path: a, c, g,

Here’s what happens after the algorithm processes Node a:

- the algorithm added Nodes b, c, and d into the frontier

- Node b is added first, so it’s processed first

- So Nodes e and f are added to the frontier

- Now the earliest node is c, so it’s then processed

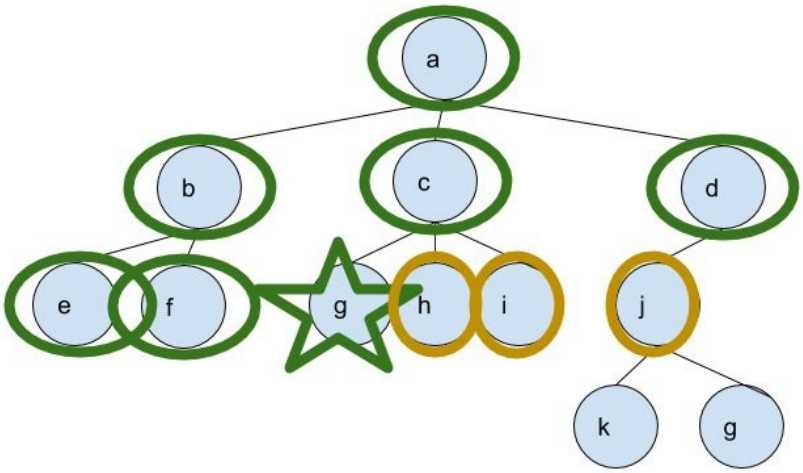
- Nodes g1, h, and i are added

- Then Node d is processed, so Node j is added.

- Nodes e, f, and g1 are processed in that order, since they’re now the oldest nodes in the frontier

- Node g1 is a goal, so a path including it and its ancestors is the solution path.

And if you look at the nodes carefully, you'll notice that the algorithm went "Breadth-First":



where the green-circled nodes have been processed already and the yellow-circled ones are in the frontier. They were supposed to be processed as well but the algorithm found a solution and finished instead.

Chapter 3.3: Lowest-Cost First Search

==== ==== ==== ==== ====

Sometimes, there can be costs between nodes and subnodes. For example, if a node had three subnodes, one of them would cost 10 to reach and the other two would cost 15.

So if the algorithm finds a solution, the path will have a total sum of all the costs required to reach the solution.

In this case, we want the solution that takes least overall cost to reach.

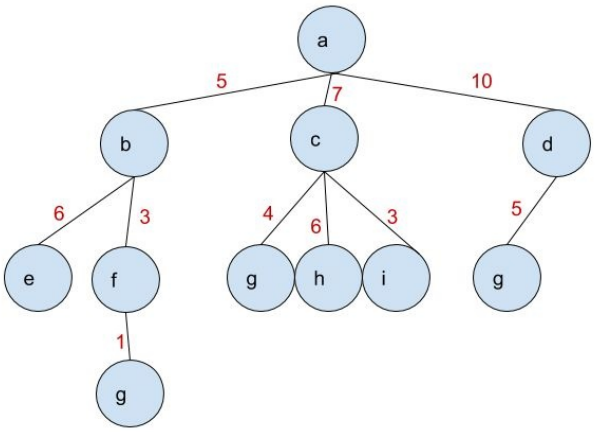
How it works:

The link between nodes and subnodes are called arcs. They can contain information vital for the algorithm to produce a viable, legal solution. For cost-based search algorithms, arcs will need costs between a node and a subnode.

[Node A] - - - -> arc A-B: cost=10 - - - ->[Node b]

Example: The Lowest-Cost-First Search Algorithm

Take note of the tree below. The red numbers indicate the cost to travel between nodes.



The algorithm will add Nodes B, C, and D to the frontier, as well as their respective costs, 5, 7, and 10. Which node will require the least cost to travel to? Node B. So Node B will be processed first, then Nodes E and F are added to the frontier, along with their respective total costs (Node E: 11 = 5+6, Node F: 8=3+5). Node C will be processed next, because it now has the lowest cost at 7. As you can see, the node that requires the least cost to reach is processed first.

Hence, why the algorithm is called LOWEST-COST FIRST.

Algorithm Analysis: Lowest-Cost-First Search

Is it Complete?

Yes, but there are certain conditions that need to be met. You can’t have arc costs be zero or any negative numbers. If this happens, you risk having the algorithm loop and run forever.

So as long as the arc costs have real, non-negative values, you can expect the algorithm to either deliver a solution or tell you that there isn’t any.

Is it Optimal?

Yes, and this is the algorithm variant’s main strength. You can guarantee that LCFS will give you a solution and a path that took the lowest cost to reach, as long as the arc costs are, again, real and non-negative values.

Otherwise, the path costs will be distorted and the solution produced might not be the optimal one.

What is its Time Complexity?

O(b^m)

At the worst-case, the LCFS algorithm will process all nodes in the tree. For example, if a tree had up to 5 subnodes per node and 4 levels down, you're looking at 625 nodes to explore.

What is its Space Complexity?

O(b^m)

At the worst case, the LCFS algorithm will have every node in the tree stored into memory. So if you have a tree with 3 subnodes per node and 3 levels down, then you may have up to 27 nodes stored into memory.

Chapter 3.4: Heuristic Search

==== ==== ==== ==== ====

Another way to determine how to get the best path is to add heuristics to the arcs. The heuristic values can represent two things:

- A very low estimate of the total cost to reach a solution

- A value to maximize: the solution should have the highest value possible

In the first case, you can estimate the total cost to reach the nearest goal node from the start. It will be admissible as long as the cost isn’t overestimated.

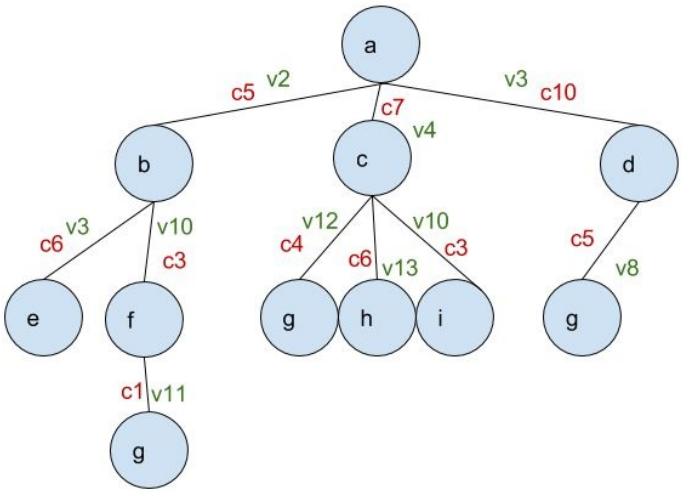
In the second case, it will be the opposite of LCFS - each node-to-subnode arc will then have a value and the algorithm will find a solution with the highest value possible.

How it works:

The link between nodes and subnodes are called arcs. They can contain information vital for the algorithm to produce a viable, legal solution. For cost-based search algorithms, arcs will need costs between a node and a subnode.

[Node A] - - - -> arc A-B: cost=10 - - - ->[Node b] Example: Heuristic Search Algorithm

The tree below is the same one from LCFS, but now with added values to the arcs.



The Heuristic search algorithm will choose its nodes based on either the closest cost to the estimate or a maximum value. First, it adds Nodes b, c, and d to the frontier.

If, say, we estimate that the cost to get to Node g is 7, the algorithm will process Node c first because its cost is 7. So Nodes g, h, and i are added to the frontier, each with respective total costs 11 for g (4+7), 13 for h (6+7) and 10 for i (7+3). So Node b will then be processed next, because its cost (5) is currently closest to 7. Then its subnodes e (11 = 6+5) and f (8=5+3) are added. Node f is then processed, so a second Node g is added to the frontier, with a cost of (9=1+3+5). The recently added Node g has the closes cost to 7, so it's then processed. It is a viable optimal solution, since the other g-nodes have costs of 11 and 15 respectively.

On the other hand, we can also have the algorithm pick a solution that creates the highest value. If we want to pick the path to Node g with the highest value, here's what happens. Node c is processed first, because of its value (4). This adds Nodes g, h, and i with values 12, 13 and 10, respectively. Node h will then be processed next, but with no subnodes to add to the frontier. And once Node g is processed, i would be a viable solution, at a value of 12.

Algorithm Analysis: Heuristic Search

Is it Complete?

No, because there is a chance that the algorithm will be in an infinite loop once it cycles between two high-value nodes or two nodes closest to the estimate.

Is it Optimal?

Unfortunately no, but this is because the value-based or cost-based algorithms can be "greedy" at times. Meaning, the algorithm will only prefer the best possible node it has at the moment, but ignoring all other options. If deeper nodes have higher values, but the algorithm can't get to them because it processes other nodes instead, then the algorithm might produce less optimal solutions instead.

What is its Time & Space Complexity?

O(b^m)

At the worst case, a heuristic search will explore every node in a tree and have each node stored in memory.

ARCHIVE A01: Frontier Search Algorithm

For the procedure below, select an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

========== ======

This is the default Frontier Search Algorithm used by most chapters throughout the book.

It is strongly recommended to view this only after you've finished building & successfully testing the algorithm already.

```
class Node {
String contents;
ArrayList<Node> children;

// CONSTRUCTOR:
public Node(String c) {
this.contents = c;
this.children = new ArrayList<Node>();
}
```

```
/*
// MAIN FRONTIER SEARCH ALGORITHM:
*/
public Path search(String query) {
// - frontier:= {new array of Nodes}
ArrayList<Path> frontier = new ArrayList<Path>();

// - create a new Path and put the Start node in it
Path p = new Path();
p.contents.add(this);
// - put the new Path into the frontier
frontier.add(p);
while (!frontier.isEmpty()) {
// - select and remove a Path <s0, s1,….,sk> from frontier;
// (use helper function pickPath() )
Path pick = pickPath(frontier);
// If node (sk) is a goal, return selected Path
if (hasGoal(query, pick)) {
return pick;
}
else {
// For every connected node of end node sk:
// - Make a copy of the selected Path
// - Add connected node of sk onto path copy
// - add copied Path <s0, s1,….,sk, s> to frontier;
int size = pick.contents.size();
Node last = pick.contents.get(size - 1);
for (Node n: last.children) {
Path toAdd = new Path();
toAdd.contents.addAll(pick.contents);
toAdd.contents.add(n);
```

```
frontier.add(toAdd);
}
}
}
// - indicate ‘NO SOLUTION’ if frontier empties
// (empty path as ‘NO SOLUTION’ here)
return new Path();
}
```

```
/*
// HELPER FUNCTION #1:
// NOTE: you can modify the position assignment to change the Search Strategy
*/
private Path pickPath(ArrayList<Path> f) {
// int position = 0;
int position = f.size()-1;
Path ret = f.get(position);
f.remove(position);
return ret;
}

/*
// HELPER FUNCTION #2:
*/
private boolean hasGoal(String s, Path p) {
for (Node n: p.contents) {
if (n.contents == s) return true;
}
return false;
}
}

// The Path Class:
class Path {
ArrayList<Node> contents = new ArrayList<Node>();
}
```

Printer Method:

Make sure to place this inside your Main Java Class.

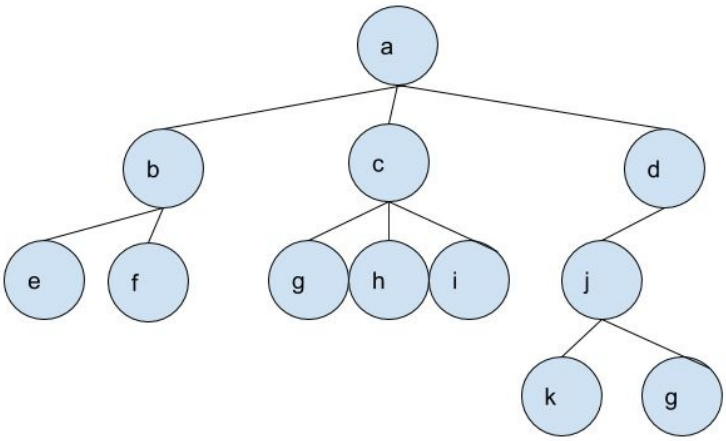
```
static String printer(Path p) {
if (p.contents.isEmpty()) return “NOTE: No Solution Found”;
else {
// System.out.println(“FOUND A SOLUTION!”);
String s = “Solution Found! Path: “;
for (int i = 0; i<p.contents.size(); i++) {
s += p.contents.get(i).contents + “, “;
}
return s;
}
}
```

ARCHIVE A02: Bigger Search Graph

For DFS & BFS, Chapter JAVA-03

For the procedure below, select an IDE of your choice. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

========== ======



Reference Image:

The code below is based on the default Node structure throughout the book.

```
Node a = new Node(“a”);
Node b = new Node(“b”);
Node c = new Node(“c”);
Node d = new Node(“d”);
Node e = new Node(“e”);
Node f = new Node(“f”);
Node g1 = new Node(“g”);
Node h = new Node(“h”);
Node i = new Node(“i”);
Node j = new Node(“j”);
Node k = new Node(“k”);
Node g2 = new Node(“g”);
a.children.add(b);
a.children.add(c);
a.children.add(d);
b.children.add(e);
b.children.add(f);
c.children.add(g1);
c.children.add(h);
c.children.add(i);
d.children.add(j);
j.children.add(k);
j.children.add(g2);
```