Java Programming

* Most Java code starts as follows:
  + public class FileName {

public/private returnType methodName(parameters) {}

}

* Most lines end with a semicolon ;
* Boolean
  + Can either be true or false
  + Conditional Operators:
  + ||(OR), ==, !=, &&(AND)
* Operators(Numbers)
  + +, addition
  + – subtraction
  + \* multiplication
  + / division
  + % Modulus (Reminder this gets the integer remainder)
* Java uses String(starts uppercase) to store a sequences of chars
  + EX: String name = “Hi”;
  + Can use + to concat strings/numbers (many times)
    - Doesn’t change value of OG strings, creates a new string
  + All objects have a toString() method which returns a String representation
  + Length() returns length of string
  + IsEmpty returns a bool if string is empty
  + indexOf(str) returns the index if str is in string or -1 if not
  + substring(int1,int2) returns a substring starting from index int1 and ends at int2
  + toUpperCase() and toLowerCase() perform intended changes to substring
* To print:
  + System.out.println()
* Var names are case sensitive and have no spaces/reserved words/start with a digit or have symbols
  + Start with lower cass for vars and classes should be capitalized
* Syntax: Grammar of programming language
* Statement: Code that does something
* Expression: Code which gives an answer
  + Conditional ones use bool operators
  + If all data types are the same, then that is the type of expression. Otherwise, if all parts are the same type but not data type, then it is the largest type
* A table with black text

  Description automatically generated
* utoboxing automatically converts a primitive type to its wrapper object (e.g., int to Integer), while unboxing converts a wrapper object back to its corresponding primitive type.
* Static: can call a method without making an object
  + In Javas case, you use it to provide functionality where making an object is cumbersome
  + Mostly for utility
* If statement
  + if (expression){  
    } else if {  
    } else {  
    }
* switch statement is like a series of if-else statements
  + Pass an expression into the statement and it will choose the correct case to execute
  + switch(expression){

case expression:

statement(s)

break;

default:

statement(s);

}

* while loop
  + while (condition){

repeat}

* for loop
  + for (int i=x,i<n;i++){}
* Arrays
  + Must declare and construct
  + Fixed size at construction
  + Syntax: new type[arraysize]
  + To set a val array[x]= val and to access val at index x we use array[x]
  + .length gives length
  + We can resize an array by making a new larger array and use System.arrraycopy(Object source, sourcestartposition, destination, destinationStartPostion, itemstocopy) then reassign original var to new array
  + A white background with black text

    Description automatically generated
* Better to fix one error at a time then recompile
* Common syntax errors:
  + Misspelled keyword or var name
  + No semicolon
  + FileName!=ClassName
  + Import Library forgotten
* Cohesion: Classes contain set of related functionalities and functionalities arer related to purpose of class
* Classes broken down into methods
* Class can contain instance variables and methods
  + Object is an instance of a class
  + Every object based on a class
  + Difference between object is values in instance variables
  + Class Date{

int day;

int month;

int year;};

* + - Has 3 instance variables
  + Define methods inside classes
  + In methods inside class no need pass an object as a parameter
  + As methods and instance vars are part of the same class, it will always change the values for the object that method is called for
  + Methods must:
    - Specify return type
    - Name of method
    - Parameters
    - Then add code after above three
* Struct used to group variables
  + Has to be passed as a parameter
  + Struct Date{

int day;

int month;

int year;};

* + If you wish to increment month per say, youd do the following
    - Void incrementMonth(struct Date\* d){

d🡪day=1;

if (d🡪month<12){

d🡪month++;

}

}

* To create an object, we need a constructor, which need correct params
  + A default constructor is provided only if no other constructors are explicitly defined in the class. If you define any constructor, the default no-argument constructor is not generated automatically
    - No return type
    - Must match class
    - Defined inside class
    - Example:
      * Date(int d, int m, int y){ constructor here, we can set vars here}
  + Can add many constructors
  + Forces users to add info
  + We use the keyword new to create an object as follows:
  + Class var =new Class()
  + Definition of a class decides what an object can do
* ObjName,varName access value in object
* objctName.Methods() calls the named Method of a object
* Every obj has same instances vars and each object can have different values for instance vars
* Pointers=References in Java
* Every object is a reference
* When using new, obj is created and stored in memory and the constructor will return a reference to where object is in memory
* Primitive vars stored directly in memory
* When we copy a primitive var into another value( copy x into y) and incremented one, only that value changes
  + Note we cant copy objects and must create a new one
* Reference has no val(null) until we assign it one
* Packages help group code and each package is a folder in which we put related code into
* We use the keyword package a the beginning of the file if we want it to belong to a package
  + Form: package packagename;
  + Can separate files into smaller packages caledd sub-packages
  + Eg a package declaration can be examples.week1 which the file is stored in a folder week1 which is inside examples folder
  + Same idea for storing all of its libraries
  + A screenshot of a computer

    Description automatically generated
* Application Programmer Interface(API) is a list of all libraries available in a language
  + Lists all classes and their instance vars, constructor, functionality description and methods while including a detailed description of each constructor and method
  + Usually import the closs and then construct the object based of class
  + Form: import package.ClassName
* Example: Program
  + A screenshot of a computer program

    Description automatically generated
* Enums allow you to define a fixed set of constants. Syntax: enum EnumName { VALUE1, VALUE2, ... }
* Classes are an objects structure while objects are the building blocks of a program
* Encapsulation: Hiding implementation details inside a class but publishing an interface to class
  + Idea is to manipulate private/hidden vars through a method
  + Accessor methods allow us to access said hidden methods
  + Mutators allow us to change private variables
* OOP is usually easy to understand and allows us to re-use working code
* Good ways to test code:
  + Establish a test case, make an instance of the class to reflect baseline, call the method and check if output is correct. Afterwards make more test cases to check other possible scenarios.
  + Try avoid just using print statements as a one and done, instead use it at key points to identify where it goes wrong, Do so gradually
* Unit testing:
  + Testing individual parts of code(units of code)
  + Done to validate each part works as intended
  + Error Handling:  
    try { // Code that might throw an exception }  
    catch (ExceptionType1 e1) { // Handle specific exception type 1 }   
    catch (ExceptionType2 e2) { // Handle specific exception type 2 }   
    finally { // Optional: Code that will always execute, regardless of exception occurrence }
  + Smallest part is a method thus unit testing revolves around them in OOP
  + JUnit is used for this
  + @Test  
    void function() {}  
    @Test   
    void function2{  
    try {} catch (exception ex) {}  
    }
  + @Test is an annotation which informs the Junit framework that current method is a test and then tests are executed order in which they are declared
  + assertEquals(x,y) tests that the two values are indeed the same and informs framework if the same or not
  + fail() method allows us to be alerted if code goes wrong
  + @BeforeEach/@AfterEach: Execute method before/after each @Test method
  + @BeforeAll/@AfterAll: Method called before/after test and before/after test class is constructed. Former is used for general set up while latter for general cleanup
* Class Diagrams:
  + Structure diagram denoted structure of program by showing classes, instance variables, methods and relationship among classes
  + UML
* Cohesion: How much the elements in a component belong together
  + Measure strength of a relationship
  + IN terms of classes
  + High cohesion classes are good as they robustness, reliability, reusability and understandability
* Access level modifiers :
  + public: Accessible anywhere. For classes mostly. Declared for methods if we expect others to use it
  + protected: Accessible only within same class/package or by subclass of class. Used if we define a method that is used in other parts but don’t want others to use
  + default: Accessible within same class/package
  + private: Only accessible/changed or even used from same file. Only change values via methods given. Only used if we write a methods from within class. Most instance variables are this.
* Visibility: Places where one can modify code
* Access level modifiers can be applied to many parts of the code
* Default isnt defined by a keyword

A white box with black text

Description automatically generated

* Class= Single concept
* Java passes by reference
* Accessor: Ask object to get a result without changing internal state
* Mutator: Change internal state of object
  + Return void
  + Shouldn’t modify parameter values
* Immutable: No mutators
  + Strings are this. They usually return a new string return than modify prev one
* Methods should follow consistent scheme
* Patterns in most Java programs:
  + Keep total
  + Count events
  + Collecting values
  + Manage properties
  + Model distinct states either via enum or a constant (public stat final)
  + Describe key properties or features
* Instance variables have separate value for each object and class variables share a single value over all objects
* Class variables are usually static.
* Static: All object’s instance variables share the same value
  + Methods can be static but rarely used other than for utility, if so all variables in method must be static
  + Can make programs easier to read by the form import static package and any method in package turns from package.method() to method()
* Final: Only assigned a value once
* Constant: Both final and static
  + Usually uppercase and separated by \_
* Singleton variables are a special case of static class variables and are almost always an object
  + Runtime is most well-known variable
    - Used to interact with OS
    - All applications have a single instance of this class
* Reusing software is cheaper than starting from scratch
* Interfaces: List of all actions an object can do and methods an object must have to be behave
  + Can be used for software reuse
  + public interface InterfaceName{ method signature}
  + All are abstract and public
  + No instance variable
  + To know if an object behaves like another we use the following syntax:
    - public class Name implements InterfaceName, InterfacrName2 {}
    - must provide an implementation for each method in interface and must match signatures.
    - Methods must be public as All methods in interfaces are implicitly public and abstract, so specifying these modifiers is optional
    - Return types, name and parameter must be the same
    - If using more than one, implement all methods from all interfaces. Even if they share the same name
  + Commons ones are:
    - Comparable: Used in collections for sorting
    - Cloneable: Make true copies of object
    - Serializable: Store/transmit instances of object
    - Runnable: Threading and concurrency
* Can convert objects to interface types
* Type casting: If we have an object that has a type, we can convert the type of the variable. Object wont be changed
  + (newType) varName
* instanceOf always to check if it is safe to typecast via syntax:
  + if (X instanceof Y){ Y a = (Y)X }
* Classes can implement methods differently
* Dynamic method lookup helps to choose correct method
* Polymorphism: Allows object references to be stored as their interface or parent type. At runtime, Java dynamically resolves and executes the method corresponding to the object's actual class.
* Can put static methods in interfaces (no instance, non-final final static vars are allowed)
  + Usually comparing two params and can only call methods
* Can put Default methods:
  + No instance vars and can only can methods
  + Like defining methods for classes
* Interfaces can use constants
* Interfaces aren’t classes so no objects are of type interface
* Mock objects: Simple object which respond appropriately to specific input
* Associate things to each other via hierarchies, as we go higher, more general whereas the lower we go, the more specific.
* Classes get functionality of classes above it (Note only one parent)
* Inheritance: When a classes is able to use methods from its superclass (and thoses superclasses’ superclass)
* Superclass is parent while subclass is children
* Inheritance syntax:   
  public class ClassName extends Superclass{}
* The extend keywords means it uses the functionality of the superclasses without changing the superclass
* Uses protected to better encapsulation
* If one doesn’t specify superclass, it is automatically of class Object (usually the top of hierarchy)
* Polymorphism can be used with inheritance although it can only use the methods of the class its stored in
* Instanceof checks to see if the new class is a version of OG class
* We can override functionalities declared in superclass
  + Name, return type, visibility and params must be the same
* Super keyword can be used:
  + To access an instance variable declared in the superclass
  + To call methods declared in the superclass
  + During construction of a subclass to pass parameters to the superclass constructor
* Super.MethodName() calls the methodname in superclass
* Use either overridden version or using super
* Only use super if you use the same variable name but avoid
* In constructor of subclass, call constructor of superclasss via super (parameters)
* Method overloading: Idea that methods that have same name but different parameters
* Methods inherited from object class can be called on any object (these include toString(), equals(Object obj) )
  + Has default implementation but not good
  + When using equals, we need to make sure they are of the correct class (use typecasting)
* When one interface inherits from another, it just adds further requirements for any class that implements the sub-interface
* The abstract keyword means something different if its used in a class or method
  + An abstract class means a class that is only accessible if inherited from another class. Cant be used to create a class.
    - Can have functional methods and instance variable
    - Once extended, must provide implementation for abstract methods or also be declared abstract
  + An abstract method can only be used in an abstract class and has no body. (Only has a body in subclass)
    - No implementation🡪must b declared abstract
* Abstraction: Process of hiding implementation details and only showing specific information to the user
* Final keyword can also change inheritance (classes can’t be extended but we want them to use)
* Final methods can’t be changed/overridden when class is extended
* Method called in constructor must be final
* Streams are used to make input and output easier (IO)
* All input and output are abstracted through stream
* Streams are:
  + Sequence of data composed of bytes
  + Is usually unidirectional
  + Output Stream: Writes data to destination
    - Java.io.OutputStream: Accepts output stream of bytes and sends to some sink
      * Core methods:  
        void write(int or byte[]): Writes byte or array of bytes to current output stream  
        void flush(): Flushes output stream and forces buffered bytes to be written  
        void close()
    - Java.io.Writer: Writes to character streams
      * Has the void write(), void flush() and void close(). The write method here is an overloaded method which writes one or more bytes to output stream
    - java.io.FileOutputStream – for Files  
      java.io.PrintWriter – useful in general for any text-based OutputStream  
      java.io.ObjectOutputStream – used to read serialized Objects
  + Input Stream: Reads data from source
    - Java.io.InputStream : Abstract Superclass representing input stream of bytes
      * Methods:  
        int read(): Reads next byte from stream and returns -1 at end of file  
        int available(): gets an estimate of number of bytes readabl from current input stream  
        void close()
    - Java.io.Reader : Reads character streams  
      Has an int read() and void close method which are similar to above
    - java.io.FileInputStream – for Files  
      java.io.BufferedReader – useful in general for any text-based InputStream  
      java.io.InputStreamReader – a bridge from byte streams to character streams: It reads bytes and decodes them into characters.  
      java.io.ObjectInputStream – used to read serialized Objects   
      java.util.Scanner – similar to java.io.BufferedReader
* Java isn’t platform independent with file and path names
  + Path names are represented differently than macOS and Unix
* Creating a file:   
  import java.io.File ;  
  File f = new File ( path\\fileName)
* Methods in File class:  
  createNewFile(), delete(), mkdir(), mkdirs(),renameTo(File dest) –all return true if successful,  
  false otherwise  
  exists() – Tests whether the file or directory exists  
  getAbsoluteFile() / getAbsolutePath()  
  getName()  
  getParent() / getParentFile() – returns the parent pathname String / File, or null if this pathname does not name a parent directory.  
  isDirectory() / isFile()  
  lastModified()  
  list() / listFiles() – returns an array of Strings /Files in the a directory
* Using BuffferedReader:  
  try{  
  File f = new File(path);  
  BufferedReader in = new BufferedReader(new FileReader f);  
  String line;  
  while((line = in.readline()) != null) {   
  do something  
  }  
  in.close();}  
  catch (IOException ex) {  
  do something  
  }
* Using Scanner:  
  try{  
  File f = new File(path);  
  Scanner in = new Scanner(f);  
  while(in.hasNext()) {   
  do something  
  }  
  in.close();}  
  catch (IOException ex) {  
  do something  
  }
* Using PrintWriter:  
  try{  
  File f = new File(path);  
  PrintWriter out = new PrintWriter ( new FileOutputStream(f));  
  if(!f.exists()) {   
  f.createNewFile()  
  }  
  for (String s: lines) {   
  out.println(s)  
  }  
  out.close();}  
  catch (IOException ex) {  
  do something  
  }
* Can use JFileChose to select file  
  try {  
  JFileChooser chooser = new JFileChooser( ) ;  
  Scanner in ;  
  if (chooser.showOpenDialog (null) ==JFileChooser.APPROVE\_OPTION) {  
  File selectedFile = chooser.getSelectedFile( ) ;  
  in = new Scanner( selectedFile) ;  
  }  
  }catch (IOException ex ) {  
  ex.printStackTrace( ) ;  
  }
* Can read from a URL using both Scanner and BufferedReader only difference is we use URL.openStream() instead of FileReader
* Scanner can read many input types in thus when constructing it, we must tell it what to read as a parameter  
  public Scanner(InputStream source)
  + Reads group of characters called tokens which are separated by whitespace chars  
    A white text with black text

    Description automatically generated with medium confidence  
    To prevent errors reading a line, we combine multiple next Methods to read parts of the line