# Java Basics:

* Most lines end with a semicolon ;
* Operators(Numbers)
  + +, addition
  + – subtraction
  + \* multiplication
  + / division
  + % Modulus
* Print: System.out.println()
* String Methods:
  + 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
* Autoboxing automatically converts a primitive type to its wrapper object (e.g., int to Integer),
* Unboxing converts a wrapper object back to its corresponding primitive type.

A table with black text

Description automatically generated

* switch(expression){

case expression:

statement(s)

break;

default:

statement(s);

}

* Arrays:
  + Type [] arr = new type [arraysize]
  + arr.length gives length
  + System.arrraycopy(Object source, sourcestartposition, destination, destinationStartPostion, itemstocopy)
* enum name {

val1,

val2,

val3

}

* Cohesion: Classes contain set of related functionalities and functionalities are related to purpose of class
* Coupling: How closely linked two classes are to each other
* Application Programmer Interface(API) is a list of all libraries available in a language

# Object Design

* Every object is an instance of a class
* Design-time coupling refers to when a dependency between classes is established at compile time
* Classes are composed of instance variables and methods
* Each class should have one and only one responsibility.
* Classes should be open for extension but closed for modification.
* Struct structName{

type a;

type b;  
};

structName->a accesses or modifies var

* public class C {  
   private type a;  
   private type b;  
   public C( parameters){  
   this.a=a;  
   this.b=b;  
   }  
   public return\_type method (parameters){  
   do something  
   }  
  }
* 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
* 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
* Strings are immutable objects
* 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 while mutators allow us to change private variables
  + Bundling data and methods operating on the data into a single unit.
  + Never let the internal state of class be exposed as it would violate encapsulation
* Encapsulate data and behaviour within a class to prevent or minimize behaviour between unrelated classes
* Visibility: Places where one can modify code
* 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 changes values via methods given and if we write methods within the class. Instance variables are this.
* Static Keyword:
  + All object’s instance variables share the same value
  + Methods can be static as well but just used for utility (all instance variables must be static)
  + Static interfaces can exist
* Final Keyword:
  + Variables are assigned a variable once
  + Final methods can’t b changed nor overridden when a class is extended
* Constants:
  + A static, final variable
* Method hiding: If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

# Interface and Lambdas

* Definition: List of all actions an object can do and methods an object must have to be behave
* Syntax:  
  public interface InterfaceName {  
   // method signatures  
  }
* Methods must be public as all methods in interfaces are implicitly public and abstract, so specifying these modifiers is optional
* Interfaces have no instance variables
* 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.
  + 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
* Type Casting: (new\_type) VarName
* The method instanceOf checks to see if an object is an instance of another
* A default method in an interface provides a body with a default implementation that can be inherited by implementing classes.
* Common Interfaces:
  + Comparable: Used in collections for sorting
  + Cloneable: Make true copies of object
  + Serializable: Store/transmit instances of object
  + Runnable: Threading and concurrency
* Can use constants
* Interfaces aren’t classes so no objects are of type interface
* Lambda Syntax: Class name = (params) -> {code}
* Lambdas act like an anonymous inner class that is a single method which returns an object
* Lambdas allow us to combine switch cases together or assign outputs
* Benefits of lambda:
  + Reduce verbosity of anonymous inner classes
  + Enables functional programming by allowing Java to support functional-style operations
  + Encourages declarative style
* Functional interface example:  
  @FunctionalInterface  
  public interface Calculator {

double calculate(double w, double h);

}

* Functional interfaces allow the use of lambda expressions by defining exactly one abstract method
* Anonymous inner class syntax:  
  Class c = new Class() {  
   @Override  
   public void method(){  
   }  
  }
  + These prevent us from having a concrete implementation

# IO Serialization

* Streams are unidirectional data compromised of bytes
* Two types of Streams:
  + 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
* 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( ) ;  
   }
* After working with an input stream, always close it or use a try-with-resources block to prevent leaks
* Can read from a URL using both Scanner and BufferedReader only difference is we use URL.openStream() instead of FileReader
* Keyword transient means the variable is avoided during serialization
* 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)  
  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
* Input = Deserialization, Output = Serialization

# Polymorphism and Inheritance:

* 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.
* Benefits of polymorphism.
  + Simplify code
  + Lets methods or objects be treated differently depending on run time
* Polymorphism Example:  
  interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

System.out.println("Drawing a Circle");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("Drawing a Rectangle");

}

}

public class Main {

public static void main(String[] args) {

Shape s1 = new Circle();

Shape s2 = new Rectangle();

s1.draw(); // Output: Drawing a Circle

s2.draw(); // Output: Drawing a Rectangle

}

}

* Inheritance: When a classes is able to use methods from its superclass (and those super classes’ superclass)
* Inherience Example:

class Animal {

void sound() {

System.out.println("Animals make sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dogs bark");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Dog();

a.sound(); // Output: Dogs bark

}

}

* Superclass is parent class while subclass is children class
* Keyword super:
  + 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 method name in superclass
* Methods can have same name, if some one can ither use super or override it
* Abstraction: Process of hiding implementation details and only showing specific data to the user
* The abstract keyword means something different if it’s 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
    - Can be extended only once
  + 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
* Example: abstract class Animal {

// Abstract method

abstract void makeSound();

// Concrete method

void sleep() {

System.out.println("Sleeping...");

}

}

class Dog extends Animal {

// Implementing the abstract method

void makeSound() {

System.out.println("Bark");

}

}

public class AbstractExample {

public static void main(String[] args) {

Dog dog = new Dog();

dog.makeSound(); // Outputs: Bark

dog.sleep(); // Outputs: Sleeping...

}

}

# Unit Testing and Exception Handling

* Testing individual parts of code(units of code)
* Done to validate each part works as intended
* Two methods of testing:
  + Traditional: Run whole program and seeing result
    - Debugger: Allows execution to be paused at locations to view value of variables
    - Print Statements
    - Testing scripts: Automated script which will execute program and provide it with required input
  + Unit: Test each component individually
    - Idea that all tests should be isolatable, repeatable, automatable and easy to write
    - Faster at development and debugging while having better design and reduction in future costs
* Simple Error Handling:  
  try { // Code that might throw an exception }  
   catch (ExceptionType1 e1) { }   
   catch (ExceptionType2 e2) { }   
  finally { }
* JUnit is used for testing
* Annotations used are:
  + @Test: Current method is a test and then tests are executed order in which they are declared
  + @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
* Assert lists in JUnit:
  + assertEquals
  + assertTrue
  + assertFalse
  + fail: Always fails a test, used to make sure part of code that shouldn’t be executed isn’t
  + assertSame: Checks to see if two objects share the same references
* Java has 3 types of errors:
  + Internal errors: Cant recover from them
  + Unchecked Errors: Error in code (e.g. IllegalArguementException). Extends RuntimeException class
  + Checked Exceptions: Something has gone wrong which you cant control. Extends Exception class
* Custom Exception:
  + Public class CustomExtension extends RuntimeException {}
* Code Coverage( Measured as a percentage) :
  + Function coverage Has each function been called by test code
  + Statement Coverage: Has each statement in application been called by test code
  + Branch Coverage: Has each branch of each control structure been executed
  + Condition Coverage: Has each Boolean sub-expression evaluated both to true and false
* List of errors:  
  A diagram of an error

  Description automatically generated

# Threads and Reflection

* Always terminate stream to let it run
* Thread: Single sequential activity being executed
* Process can create or destroy multiple threads
* Threads allow parallel computation and shared memory spaces for computations
* Deadlock: Threads are waiting for each other, but neither is running
* Livelock: When Threads are running but not responding
* Two ways to implement threads in Java as it doesn’t support multiple inheritence:
  + Implementing Runnable interface, define run() method and pass it to a thread instance  
    public class MyRunnable implements Runnable {

public void run() {

// Task code

}

}

* + Extend Thread class and override run method   
    public class MyThread extends Thread {

public void run() {

// Task code

}

}

* Methods in Thread class:
  + Run: Defines what thread does, process body
  + Start: Schedules thread and later calls run
  + Sleep: Suspends thread for n milliseconds
  + Wait: suspends Thread for period of time until another state is reached
  + Join: Suspends thread until other thread has been terminated
* To allow two thread to access same data without causing race conditions, use synchronized methods and blocks as it only allows one thread to access method at a time
* All objects have a lock which threads use for synchronisation
* An interrupted Exception error is thrown whenever a thread is interrupted (via sleeping, waiting or suspended). Checked exception
* OutOfMemoryError is when too many threads collectively consume memory allocated and will cause the virtual machine to terminate
* Reflection allows us to inspect Objects at runtime
* Part of java.lang.reflect package
* Used to analyse class info and details, allow us to invoke methods and constructors dynamically, invoke test methods dynamically, debugging etc.
* To invoke a method if we know the name, we can use:
  + Class.getDeclaredMethod(name, parametertype)
  + Method.invoke(Object, parameter)
* Reflection example:  
  Class<?> clazz = object.getClass();

Method method = clazz.getDeclaredMethod("methodName", paramTypes);

method.invoke(object, params);

* Applications of Reflection:
  + Debugging.
  + Dynamic method invocation.
  + Testing frameworks: Dynamically discover and execute test methods.
  + Serialization frameworks: Access private fields to serialize objects.
* Reflection Limitations:
  + Performance overhead: Reflection is slower compared to direct code execution.
  + Security risks: Can bypass encapsulation, exposing private members.
  + Reduced maintainability: Reflection-based code can be harder to understand and debug.