**Object-Oriented Programming:**

**Primitive Data:**

-   Primitive data types store single, simple values

-   Examples: byte, char, int, float, double, boolean

-   As programs became **larger** and more **complex** programmers began to need to **group similar pieces of data together** this led to structures and arrays

**Structures (Similar to Array):**

-   **Structures were the precursors to objects**

-   Structures store many pieces of data

-   **Can** store different types of data

**Arrays:**

-   Stores many pieces of data

-   **Cannot** store different types of data

**Objects:**

-   Objects are **instances of a class**

-   Classes are **templates for objects**

-   Example: The teacher class represents **any given teacher**, a teacher object represents **only one singular teacher**

-   **Object-oriented programming** helps programmers create complex programs by **grouping together related data and functions**

**Encapsulation:**

-   Refers to **bundling data with methods** that can operate on that data within a class

-   Essentially, it is the idea of **hiding data** within a class, **preventing anything outside that class from directly interacting with it**

-   This **does not mean** that members of other classes cannot interact at all with the attributes of another object

-   Members of other classes can **interact with the attributes of another object through its methods**

-   Methods are the **functions defined within the class**

**Getter and Setter Methods:**

-   Getting methods – retrieving information

-   Setting methods – changing information

-   Generally best to not allow **external classes** to **directly edit an objectʻs attributes**

-   Each piece should not **have access to** or **rely on** the **inner workings of other sections of code**

-   **Information hiding**, or keeping the data of one class hidden from external classes, helps you keep **control of your program** and prevent **from becoming to complicated**

**Abstraction:**

-   Refers to only **showing essential details** and keeping everything else hidden

-   Example: When you are driving a car all you need to know is how the steering wheel works or the gas pedal, you donʻt need to know how it works internally in order to drive it

-   The classes you create should act like your car. **Users of your classes should not worry about the inner details of those classes**

-   Modern programs are very complex to the point where **multiple programmers tend to work on one program.** In this case, itʻs best if the section that you work on is able to function without knowledge of the inner workings of a colleagueʻs section

**Interface:**

-   **The interface** refers to the way sections of code can **communicate with one another**

-   Typically done through methods each class is able to access

-   Interface allows classes to be independent of each other and only able to communicate in certain ways. If all classes relied on each other then if one fails, they all fail

**Implementation:**

-   **The** **implementation** of these methods, or how these methods are coded, should be hidden

-   If classes are **entangled**, then one change creates a **ripple effect that causes many more changes**

**Inheritance:**

-   Principle that allows classes to derive from other classes

-   **Example:** Weapon class has certain methods like damage or attack. Certain classes can inherit from this weapon class such as the club and sword. These classes of club and sword can have similar attributes but are also different in many ways, which can be described in its individual class. In this case Weapon is the **superclass** and the club and sword classes are **subclasses**

-   **Subclasses inherit from superclasses**

-   The **class hierarchy** acts as a web of classes with different relationships to one another

**Access modifiers:**

-   Change which classes have access to other classes, methods, or attributes

-   **Public:**

o   **Public members** can be accessed from anywhere in your program

o   This includes anywhere both inside of the class hierarchy it is defined as well as outside in the rest of the program

-   **Private:**

o   **Private members** can only be accessed from within the same class that the member is defined

o   Allows you to create multiple private members of the same name in different locations so that they do not conflict with one another

-   **Protected:**

o   **Protected members** can be accessed within the class it is defined, as well as any subclasses of that class

o   Essentially makes protected members private to the hierarchy in which they are defined

**Polymorphism:**

-   Describes **methods that are able to take on many forms**

**Dynamic polymorphism:**

-   Occurs **during the runtime** (when the program is being executed) of the program

-   This type of polymorphism described when a **method signature is in both a subclass and a superclass**

-   The methods share the **same name**/**parameters** but have **different implementation**

-   In dynamic polymorphism the implementation of the **subclass that the object is an instance of overrides that of the superclass**

-   **Example:** Class Car is the superclass and class sportsCar is the subclass. They both have a method called .drive. If you make an object (instance of a class) called mySportsCar and pass in mySportsCar.drive, the sportsCar implementation of the .drive method will override that of the Class Car method. And vise versa.

-   This works because the **form of the method** is decided **based on where in the class hierarchy it is called**

-   The implementation of a method signature that will be used is **determined dynamically as the program is run**

**Static Polymorphism:**

-   Occurs during the **compile-time** rather during the runtime of the program

-   This refers to when multiple **methods with the same name but different arguments** are defined in the same class

-   Overall, polymorphism allows methods to take on many different forms in the same class and in different classes. When utilizing polymorphism and method overloading, be sure that you are calling the correct form of the method

-   Ways to differentiate methods of the same name include:

o   Different number of parameters

o   Different types of parameters

o   Different order of parameters

-   This is known as **method overloading**

-   Despite the methods having the same name, their signatures are different due to the different parameters they accept (arguments)

-   **Example:** Class Car has 3 different .drive methods

o   1) .drive(int spd, string dest, int dist)

o   2) .drive(int spd, int dist)

o   3) .drive(string dest, int spd)

**Realize how these methods all have the same name(.drive), but can be implemented differently becuase they take different parameters, different order of parameters, and a different number of parameters.**

**Other Java Notes:**

Scanner format

type identifier = new type();

compare strings use name.equals()

compare primitives use name == name

Comparision operators: used for numbers (primitaves)

== equality (equals)

!= inequality (not equals)

<= less than or equal to

>= greater than or equal to

Logical operators: used for string

&& and (both need to be true)

|| or (either can be true)

! not (inverse the boolean)

ICU: (loops) while loop, do-while loop, for loop (most people use integer i)

initialization

comparison

update

**Argument vs Parameter:**

An **argument** is the instance passed to the method during runtime. The term **parameter** refers to any declaration within the parentheses following the method/function name **in a** method/function declaration or definition; the term **argument** refers to any expression within the parentheses of a method/function call.