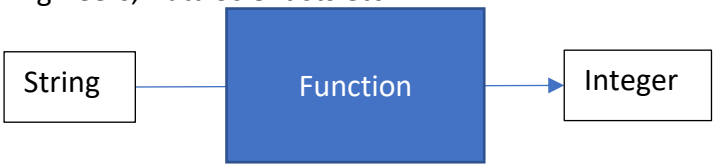
CS2030 Overview: Design Principles
Having completed CS2030, this is my review/ what I have learnt from the module.

At the most fundamental level:
CS2030 is about writing code and software that is easily maintainable, readable and useful. It could be said to be a foundational module to understand the processes involved in software development.

This is expounded through the use of the core concepts of **Abstraction & Encapsulation.**

Abstraction:

The core idea that a function or a piece of software is a blackbox from which a certain input goes in and a known output comes out. How this output results from the input is a result of the function and the "client" using the function/program does not need to know how it was done.
Analogy:
Similar to how one might use a programming language Python, which is based on C, which is itself a piece of software derived from binary code read by the computer. Most of us would not know **how** python was created from C, and we just take and use the programming language as a matter of fact to perform our tasks as Software Engineers, Data Scientists etc..



When a piece of software depends on another piece of software in order to achieve its functionalities, it is said to be a 'client' of the lower level function, and the implementer of the lower level as the 'implementer'.

Thus this creates the idea of **hieraechy** in which a piece of software depends on the output of it's dependencies to run.

Hence to this end, abstraction is the principle of hiding the **Implementation** of a piece of software from the client, to preserve flexibility and choice when we want to alter the blackbox to work in a different manner.

All that is then relevant to us, the client would be the interface of eg, python
Where we can simply just:
Print('Hello World')

TL,DR
Implementation of a piece of software is irrelevant to the usage of the functionalities upon it to build higher levels of software. => Client's view vs Implementer's view

Encapsulation:

Encapsulation naturally flows with the ideas of abstraction, through the grouping of functions that work towards a similar functionality into what is termed as a **class.**

This is useful to hide the internal state and structure of a "**Object**" from the client view. This is known as: '**Information hiding'**. Information hiding allows the implementer to hide the underlying implementation of a piece of software through encapsulating a variable inside the class with the access modifier : "private"

This ensures that outside of the class itself, nothing can change the state of the variable except methods (functions associated with the class) associated with that method.

Therefore, instead of allowing the client to access all variables inside a class, instead they use getters -methods to display said data, and setters – methods that assist in changing the data, through an provided interface.

Such an implementation prevents the client from perversely changing the state of said variable to cause an unintended effect, through the setter method which can catch said unintended state changes.

Modifier	Class	Package	Subclass	Other Classes
Private	Yes	No	No	No
No modifier	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

To this end, the Java language provides certain access modifiers to allow one to control such access.

TL,DR
Bundle related functions into a class.
Don't use public or no modifier when coding for CS2030 unless necessary.

Now that we have our foundational principles, we can move on to **Inheritance** and **Polymorphism.**

Inheritance:

Inheritance is an extension of the heuristic of DRY- Don't Repeat Yourself.
Through inheritance of classes in Java and other OOP languages, the parent class's methods are inherited by the child class, and the child class extends the functionality of the parent class.
Eg: Circle inherits from Shape

Hence if Shape has a method .getPerimeter()

Circle inherits the method implicitly from Shape and does not need to include a .getPerimeter() method. Unless the way to get said perimeter changes, which then links to….

Polymorphism:
Polymorphism is where a child class eg Circle inherits the getPerimeter() method, however the way that Circle gets the perimeter is different, through calculating the Circumference rather than how the Shape class gets said perimeter.

This is a case of Overriding, where the method from the parent class is overrided by the local implementation of the class.

Overloading is another concept in java such that methods with the same name, eg
add(int x, int y)
add(double x, double y)

can exist in the same class without any complaints by the java compiler.

Wrapping up:
These basic principles are the building blocks for us to understand the SOLID principles, a set of software design principles to adhere to when designing Object Orientated Software.

THE SOLID PRINCIPLES

SOLID is one of the most popular sets of design principles in object-oriented software development. It's a mnemonic acronym for the following five design principles:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

Single Responsibility Principle:

As quoted by Richard C. Martin, *A class should have one, and only one, reason to change.*

This is a foundational principle to building robust and maintainable software.

Why?
Imagine you implement one function that handles everything related to a certain functionality in python 3.7.
Then, an update to python 3.8 changes the way a certain key data interaction in your occurs.
You as the implementer now have to change everything within that function all at once, and have to trace through hundreds of lines of code to resolve that one bug. All of which do not allow the program to function.

Now imagine this, but with hundreds of more changing interactions.

Hence we come to this idea of the responsibility of a class.

Linking to encapsulation, all the methods directly related to a class should be tied to itself.

Anything that does not directly affect this class is not the responsibility of this class. Hence when designing software we should ask ourselves, is this method directly affecting the class?

Furthermore when extending software, we should also ask if this functionality can be broken down and modularized in order to allow future changes to be made with the single responsibility in mind.

This extends from the basis of abstraction layers to ensure that software created can be modified and extended easily.

Open Closed Principle:

The Open Closed Principle by Bertrand Meyer states: "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*"

The general idea of this principle is sound, to write your code in such a way that one can add new functionality without changing the original code, hence preventing scenarios where a change in one class requires changing every depending class. However this is proposed to be implemented through tight coupling if the subclasses depend on implementation details of the parent class.

Hence, Robert C. Martin redefined it to be the polymorphic Open-Closed Principle, utilizing Java's Interfaces to allow different implementations which can be easily substituted without changing the code which uses it. The interface acts as a contract that binds the class to implement a set of methods clearly stated within the interface.

Eg List<T> vs ArrayList<T>

The interface hence acts as an additional layer of abstraction barrier that enables loose coupling of different classes.

The implementations of an interface to two unrelated classes are totally independent but both behave in a certain way.

Liskov Substitution Principle:

Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.

In practical terms, what this means is that objects of the subclass should behave in the same way as that of the superclass, when used in the same context as the superclass.

Hence overridden methods in the subclass should also take in the same parameters but have less strict validation rules than that of the overridden class, and hence extends functionality.

LSP is a way to standardize polymorphism, such that we know the expected basic behavior of a class, from the inheritance of its parent class.

Interface Segregation Principle:
"Clients should not be forced to depend upon interfaces that they do not use."

The basic idea behind this principle is the same as that of the SRP, to reduce the side effects and frequency of required changes, through organising your software into multiple independent parts.

For Eg:

```
Interface Stringable{
Public String toString();
}

Interface Printable{
Public void print();
}

Interface Shape extends Stringable,Printable{
Double getPerimeter();
}
```

From this example, we can see that the different responsibilities of calling toString() and print() are segregated into separate interfaces, and then implemented in the Shape interface.

By following this principle, we prevent bloated interfaces that define methods for multiple responsibilities, and maintain flexibility to choose what functionalities we want to implement for a piece of software.

Dependency Inversion Principle:
Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

This goes back to the idea of abstraction, to establish a hieraiechy of dependency through the use of the black box. Taking the output from a lower level module, to transform in the current level of abstraction.

It also means that we should avoid cyclic dependencies in our code wherever possible.

Cyclic Dependencies should be instead managed by a Manager class, or via some other Software Design Pattern.

Wrapping up:
Overall, the SOLID principles are a method to design robust and maintainable code, and are a key tool in the development of robust and maintainable software.