* **Static –** can be accessed without an instantiation
* **Overloaded –** same method name, different parameters are passed in
* **Call by reference –** n.x \*= n.x
* **Call by value –** x \*= x
* **Sub class –** receives all features from parent class automatically,
* **Polymorphism –** one reference can refer to various different classes, can reference parent class and then use child class (add child class type to list of type parent class)
* **Dynamic dispatch –** when overridden method is called via reference, actual method to execute is selected based on type of object referenced
* **Multiple dispatch –** multiple methods of same name are created, but different parameters are passed in, then dynamic dispatch is done to select which one to use
* **Abstract –** prevent us from making instances of a class, or defining a method
* **Visitor pattern –** represents an operation to be performed on elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates
* **Interface –** can’t instantiate, a class must implement it
* **Generics –** allow compiler to keep track of objects types at compile time. Can parameterise classes with types. Removes need for downcasting.
* **Encapsulation –** only object can see or interact or change structures/variables
* **Singleton design pattern –** ensures a class only has one object instance, provide a global point of access to it. Turn all constructors private, provide static method that returns a unique object instance.
* **Reflection –** objects can extract/manipulate info of objects they hold a reference of. Brakes encapsulation
* **Iterators –** want to iterate over all elements independent of the structure. Class needs to implement Iterable to be iterated over using ‘:’ notation, the interface demands to get hold of an iterator for the structure. Can then use in for loops
* **Strategy pattern –** defines a set of encapsulated algorithms that can be swapped to carry out a specific behaviour
* **Observer pattern –** used to allow an object to publish changes to it state. Other objects subscribe to be immediately notified of any changes
* **Composite pattern –** allows to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
* **Model-View-Controller –** a meta pattern: it describes a general design for GUI realisation
* **Lambda expression –** unnamed function, a piece of re-usable code that can be treated as functionality data that is passed around. It has a type signature and body but no name
* **Bulk operations –** apply a lambda operation to an entire collection automatically (forEach)
* **Factory pattern –** define an interface or abstract class for creating an object, but let sub-classes decide which class to instantiate. Factory Method lets a class defer instantiations it used to sub-classes. This allows for an API to return and use objects without exposing their classes. Can be used to promote encapsulation by shielding off the process of instantiation.
* **Abstract factory –** provide an interface for creating families of related or dependant objects without specifying their concrete classes.
* **Bridge pattern –** decouples an abstraction from its implementation so the two can vary independently. Avoids any permanent binding by removing the direct dependency between abstraction and implementation.
* **Adapter pattern –** used to provide a link between two incompatible interfaces by wrapping the ‘adaptee’ with a class that supports the interface required by the client
* **Facade pattern –** provides a unifies access interface to a set of interfaces in a sub system. It defines a higher-level interface that makes the sub system easier to use
* **Proxy pattern –** provides a surrogate or placeholder proxy for an object to control access to another object
* **Mediator pattern –** defines an object that encapsulates how a set of objects interact. It avoids objects referring to each other directly
* **Memento pattern –** without violating encapsulation, captures and externalizes an object’s internal state so that the object can be restored to this state later
* **Command pattern –** encapsulates requests as objects … supporting to queue, log or undo requests
* **State pattern –** basically bridge but allows an object’s behaviour to change along with its state
* **Flyweight pattern –** facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient