Introduction:

Programming has been around for a very long time (relatively speaking). As such, developers have discovered many unique paradigms for how we think about and write code. Programming in many ways has become a free market of ideas, each person advocating for one pattern or algorithm to rule them all. It is one thing to learn how to read and write code, but it is another thing entirely to be able to reason about and engage with these ideas. Every good developer, at a base level, needs to understand at least some of these core concepts – how they are applicable, and what their advantages and disadvantages are. It is this knowledge that I hope to be able to impart unto you, which will hopefully get you up to speed quicker than it took me.  
  
Paradigms:

If you read the word paradigm and weren’t entirely certain as to what I meant, then allow me to explain. A programming paradigm generally refers to a philosophy that is adhered to as we write software. Paradigms are not necessarlily tied to any particular language, though many languages are written with specific paradigms in mind. As an aside, this teeters into the realm of Domain Specific Lanuages (DSLs) compared to General Purpose Languages (GPLs). A domain specific language is a language which is written and used for a specific field or domain. An example of a GPL is R or Matlab, which are intended to be used for data analysis. A general purpose language, on the other hand, might be something like Python. Anyhow, a common language paradigm with which most people are familiar is Object Oritented Programming (OOP). OOP is the philosophy or paradigm which is adhered to as we program. As I stated, it is not necessarily tethered to the language, but the language typically pushes us towards the paradigm it was designed for. C++ is an example of a language which loosely pushes towards using OOP, whereas Java more strongly encourages using OOP. In neither case, however, are you forced to write OOP code. It is important to note that classes do not make a language OOP by any means. OOP entails a lot more than that, which we will cover explicitly at a later point. Paradigms don’t have to just apply to programming either. There are many database paradigms, for example, the most prominent being relational databases. We will be focusing specifically on programming paradigms for the purposes of this document though. Here is an heirarchical overview of the paradigms we will cover:

* **Imperative**: The code direcly controls execution flow and state change
  + **Procedural**: Organized as procedures which call each other
  + **OOP**: Organized as objects that contain both structure and associated behavior
* **Declarative**: Code declares properties of the desired result, but not how to compute it
  + **Functional**: A desired result is declared as the value of a series of function evaluations
  + **Logic**: A desired result is declared as the answer to a question about a system of facts and rules
  + **Reactive**: A desired result is declared with data streams and the propogation of change
  + **Domain Specific:** Languages which are specifically designed for a particular domain or branch of applied science, math, etc.

These may seem like just a word salad as you read them, but as we cover them individually, the concepts will hopefully become more concrete.

Imperative:

You may think of the word imperative as meaning urgent or necessary, but another way that the word can be defined is command or instruction (which is, of course, the way it’s intended in this context). When we say imperative language, we’re being a little more abstract, as an imperative language can be referring to one of a few imperative paradigms. Imperative languages are typically one of two language paradigms: procedural, or OOP. There are perhaps others, but these are the most common. We’ll cover both of these separately, but for now let’s focus on what makes these paradigms imperative. The

Declarative:

The declarative paradigm is basically a catch-all for any type of language that does not adhere to the core tenants of an imperative language. Typically declarative languages emphasize the use of keywords or descriptive functions to perform operations on a set of data. SQL is a good example of a declarative language that uses keywords to describe the set of operations to perform on the data set. We’ll look at 4 kinds of declarative paradigms: functional, logic, reactive, and domain specific (DSL). These are not all of the declarative paradigms, but they are probably the most common.

Functional:

Functional programming, as the name suggests, makes functions the first class citizen of programs. Functional languages tend to consistently rely upon the same patterns such as lambda expressions, filters, maps/functors, applicatives, monads, recursion, pattern matching, etc. If you’re not familiar with all those terms, it probably means you’ve not used a functional language or any language which borrows functional idioms e.g. Rust or TS. Haskell is a classic example of what is called a “pure functional language”. The word ‘pure’ in this context refers to the immutability of an object. Functions in haskell are not allowed to modify input data directly; they must return new data instead.

Let’s clarify some terms that were used previously. Lambda expressions are based off lambda calculus, which is a system of formal logic used in mathematics for expressing computations. Lamda calculus is proven to be turing complete. I recommend looking into lambda calculus a little bit if you’re interested. From the perspective of a programmer, lamda functions are just tiny functions… They are generally most useful when we need to pass a function as an argument to another function. With filters, we can omit certain data from the outbound set based on whether the element passes a specific condition. Functors, A.K.A. maps, take a set of initial values and performs an operation on all members of the set. We call this mapping since we map the original values in the set to their transformed values post-operation.

Design Patterns (Gang of Four):

Design patterns are simply patterns which naturally emerge when attempting to solve problems. Of course, design patterns were not coined by any one person, but the design patterns that have been popularized in the industry were proposed by the Gang of Four (GOF) in the book “Design Patterns: Elements of Reusable Object-Oriented Software”. The GOF is comprised of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The book splits its list of 23 patterns into 3 categories: creational, structural, and behavioral. I’ll list all 23 patterns in terms of these categories and then we can go over each individually.

**Creational:**

* Singleton
* Factory
* Abstract factory
* Builder
* Prototype

**Structural:**

* Adapter
* Composite
* Proxy
* Flyweight
* Facade
* Bridge
* Decorator

**Behavioral:**

* Template method
* Mediator
* Chain of Responsibility
* Observer
* Strategy
* Command
* State
* Visitor
* Interpreter
* Iterator
* Memento

Singleton (Creational):

The singleton pattern is simple, yet elegant. Sometimes, we don’t desire to have multiple instances of a class. For example, you may want some sort of state/context object or manager class e.g. a LogManager, which oversees the process of logging messages to the logger. The idea within an OOP paradigm is to limit the users ability to create multiple instances of a class by returning a static reference to an already instantiated instance of the class via some method or function. That’s fancy talk to say that we create a single instance of the class that gets returned when we call into some method/function. The constructor is intentionally made private so that no external objects may invoke it. Here’s some pseudo code:

public class LogManager {

private static LogManager instance;

// Constructor

private LogManager() {

}

public static LogManager getInstance() {

if (instance == null) {

instance = new LogManager();

}

return instance;

}

}

As you can see, the default constructor is effectively deleted by marking it private. Instead, the only way to access the only instance/singleton of the class is via our exposed getInstance() function, which will initialize the instance if this is the first time getInstance() is being invoked, otherwise it just returns the already initialized instance.

Factory (Creational):

The idea behind the factory pattern is to generate instances of a class with pre-defined or pre-templated data. A classic example is the Hamburger class. Assume you’re creating a menu for a burger joint such as McDonalds. The menu should only allow for constructions of a burger with the unique set of toppings for each item on the menu. For example, a big mac should have a set of predefined toppings that differ from those of a quarter pounder. To do this, we declare two classes: one which acts as the data class for the burger itself, and one which is the burger factory for generating predefined instances of said burger class. Here is an example:

public class McDonaldsBurger {

private int numPickles;

private boolean hasCheese;

private boolean doublePatty;

private boolean onions;

private Condiment condiments;

public McDonaldsBurger(

int numPickles,

boolean hasCheese,

boolean doublePatty,

boolean onions,

Condiment condiments

) {

this.numPickles = numPickles;

this.hasCheese = hasCheese;

this.doublePatty = doublePatty;

this.onions = onions;

this.condiments = condiments;

}

}

public class McDonaldsBurgerFactory {

public McDonaldsBurger orderQuarterPounder() {

return new McDonaldsBurger(4, true, false, true, (Condiment::Mustard | Condiment::Ketchup));

}

public McDonaldsBurger orderBigMac() {

return new McDonaldsBurger(8, true, true, true, Condiment::BigMacSauce);

}

}

AbstractFactory (Creational):

As you could probably guess, an abstract factory is just an abstract class used for defining additional factories. Typically abstract factories are overkill for smaller projects, though when we require a lot of abstraction in our codebase, it may be useful to be able to define some default factory methods via an abstract class and then override the methods that require changes per factory that extends the abstract factory class.

Builder (Creational):

The builder pattern builds (no pun intended) off of the factory pattern. It allows for added flexibility by allowing the user to tweak the parameters that they want to set. This really isn’t much different from just allowing the user to enter in parameters into a parameterized constructor, though it does have a couple slight benefits. First of all, the builder allows us to modify an already intialized instance of the class by allowing us to update data at a later time rather than creating a completely new instance of the class. Second, the builder pattern is syntactically more pleasing to work with for constructors with many parameters. Let’s assume we have the same McDonaldsBurger data class. Now we want to create a builder class for the burger rather than use a factory:

public class McDonaldsBurgerBuilder {

private McDonaldsBurger burger;

public McDonaldsBurgerBuilder addPickles(int numPickles) {

burger.setPickles(burger.getPickles() + numPickles);

return this;

}

public McDonaldsBurgerBuilder addCheese() {

burger.setCheese(true);

return this;

}

public McDonaldsBurgerBuilder addPatty() {

burger.setDoublePatty(true);

return this;

}

public McDonaldsBurgerBuilder addOnions() {

burger.setOnions(true);

return this;

}

public McDonaldsBurgerBuilder addCondiments(Condiment condiments) {

burger.setCondiments(condiments);

return this;

}

public McDonaldsBurger build() {

return burger;

}

}

Note how we return a reference to this in each of the add methods. This is to enable method chaining, meaning that we can do something like the following:

burgerBuilder.addCheese().addOnions(5).addPatty();

Once we’re done adding toppings, we can construct the actual McDonaldsBurger instance using the builder class’ build() method.

Prototype (Creational):

The prototype pattern is basically just a copy constructor. It’s the idea that we can take a pre-existing instance of a class (known as the prototype) and then create an identical copy that retains all of the prototype’s state, which we can further modify if desired. A basic copy constructor looks something like the following:

public class MyClass {

int a;

boolean b;

// Default constructor

public MyClass() {

}

// Parameterized constructor

public MyClass(int a, boolean b) {

this.a = a;

this.b = b;

}

// Copy constructor

public MyClass(MyClass copy) {

copy.a = this.a;

copy.b = this.b;

}

}

Rather than making the user create an uninitialized instance of the class that they then have to pass into the copy constructor, we could alternatively create a separate function which just returns something like this.copy() or new MyClass(this.a, this.b) depending on what your language supports.

Adapter (Structural):

The adapter pattern is a fairly simple concept. We use adapters in real life when we have a part we want to use which is formatted differently than what the item accepts as an interface. For instance, a VGA to HDMI adapter allows you to connect to a television that accepts HDMI as its primary interface even though the part we’re using is a VGA cable. In code, this is really quite simple to accomplish. We create an interface which both classes must implement so that they share common behavior. Here’s some pseudo code:

public interface VgaHdmiAdapter {

public Result connect();

public Result disconnect();

}

public class VgaCable implements VgaHdmiAdapter {

@Override

public Result connect() {

println(“Connected VGA”);

return Result::OK;

}

@Override

public Result disconnect() {

println(“Disconnected VGA”);

return Result::OK;

}

}

public class HdmiCable implements VgaHdmiAdapter {

@Override

public Result connect() {

println(“Connected HDMI”);

return Result::OK;

}

@Override

public Result disconnect() {

println(“Disconnected HDMI”);

return Result::OK;

}

}

public void pluginCable<T where T implements VgaHdmiAdapter>() {

T cable = new T();

cable.connect();

// Do stuff…

cable.disconnect();

}

Composite (Structural):

The idea behind composites is to essentially combine smaller units into a larger composite object or data class. One way of accomplishing this might be to create child classes of the composite class. For example, in a GUI framework you may have a window class which is composed of smaller sub panels that make up the window. Those sub panels might also be composite items build from individual components.

Continuous Integration/Delivery:

Waterfall:

Agile:

GitFlow:

GitHub Flow: