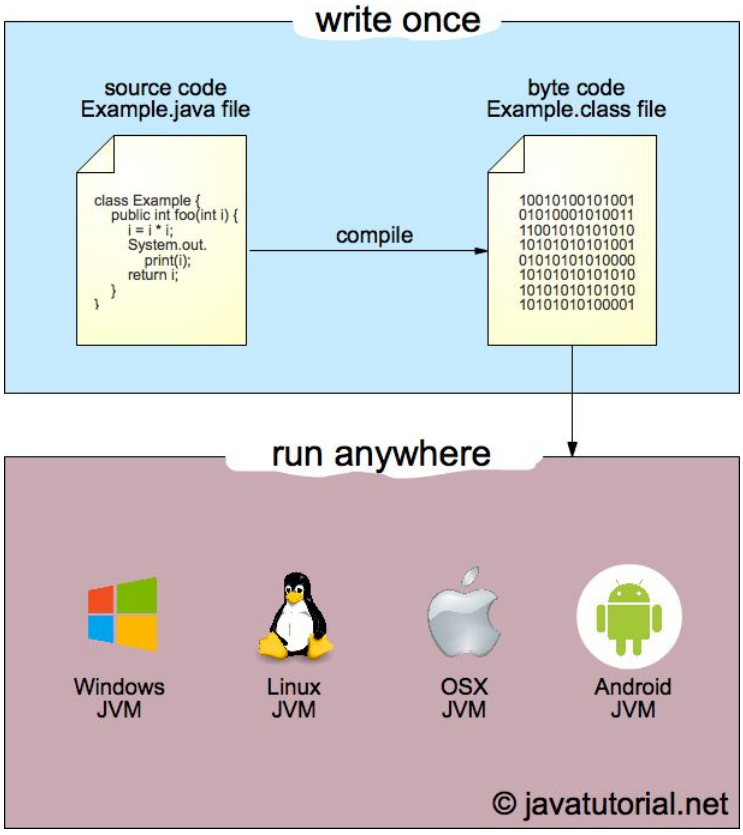
To read a Java class file and parse the headers means : 1. Java = ClassFile() and 2. print('magic: ', java.get_magic()), minor_version, major_version etc. To have beginnings of implementation.

LDC Opcode : Loads a constant from the constant pool (through the index) and then pushes that onto the stack

JVM is basically “write once, run anywhere.” This means unlike programming languages like C++ where code is compiled for specific platforms only, Java source code is first compiled into bytecode (.class file). Once compiled, the class file is interpreted by the virtual machine.

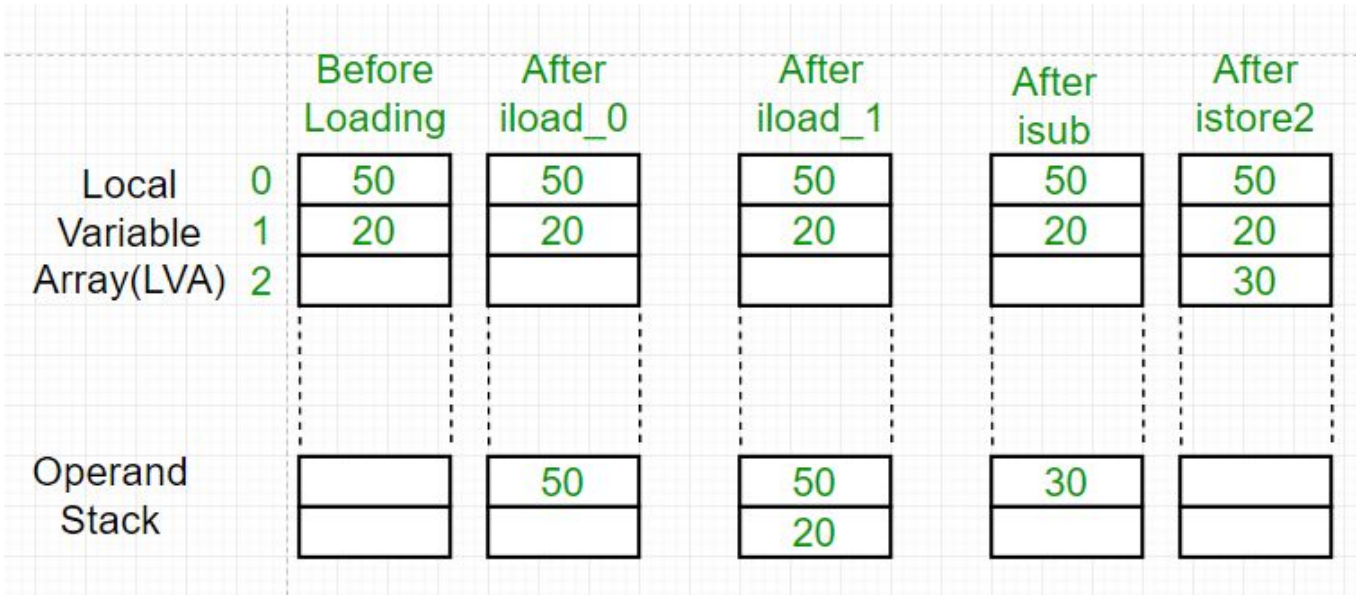


First the java source code is compiled (.java file) into bytecode (.class file). Bytecode is just a middle language between java and machine language.

Runtime Data Area : There are 5 components inside here.

1. Method Area : All class level data is stored here including static variables. There is only one method area per JVM and it’s a shared resource.
2. Heap Area : All objects and their corresponding instance variables and arrays will be stored here. Also, one heap per JVM. Since Method and Heap areas share memory for multiple threads, data stored is not thread safe.

3. Stack Area : Every thread, a separate runtime stack will be created. Every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. Stack area is thread safe it is not a shared resource. The stack frame is divided into three sub-entities : Local Variable, Operand Stack, Frame data. Local variable array is related to the method of how many local variables are involved and the corresponding values will be stored here. Operand stack is if any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation. Frame data are all symbols corresponding to the method will be stored here.



4. Garbage collector collects and removes unreferenced objects. This can be triggered by calling System.gc() but execution is not guaranteed.

Decorator example in python: @gfg_decorator

```
def hello_decorator():
    print("Gfg")
```

Above code is the same as:

```
def  hello_decorator():
     print("Gfg")
hello_decorator = gfg_decorator(hello_decorator)
```

A function that take other functions as arguments are called higher order functions.

Higher order function example:
```
def inc(x):
    return x + 1
def dec(x):
    return x - 1
def operate(func, x):
    result = func(x)
    return = result
```
operate(inc, 3) = 4
Operate dec, 3) = 2

Decorators - have the same super type of the objects they decorate, one or more decorators,

The Decorator Pattern - Attaches additional responsibilities to an object dynamically. Provide flexible alternative to sub-classing for extending functionality

Inheritance - Using inheritance to get type matching but not behavior. Java requires this, but other languages don't

The factory pattern :

Factories - Handles the details of object creation (encapsulates the creation in a subclass) (decouples interface from creation). Can return a variety of types. Client doesn't care which type. Can add additional types easily. If static, can't subtype.

Factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

Abstraction - Factories don't have to be abstract. Can have default and that can call down if necessary

Example of factory design pattern is The Dependency Inversion Principle. It depends upon abstractions and doesn't depend upon concrete classes.

The Dependency Inversion Principle; High level modules should not depend on low level modules. Both should depend abstractions. Abstractions should not depend on details.

Traditional View - Policy Layer -> Mechanism Later -> Utility Layer

Traditional View Inverted- Policy Layer -> (<<interface>> Policy Service interface) <- Mechanism Layer -> (<<interface>> Mechanism Service interface) <- Utility Layer

Therefore, no variable should hold a reference to a concrete class. No class should derive from a concrete class. No method should override an implemented method of its base classes.

Dependency Injection - A technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) the would use it. The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern. This fundamental requirement means that using values produced within the class from new or static methods is prohibited. The client should accept values passed in from outside. This allows the client to make acquiring dependencies someone else's problem. The intent behind dependency injection is to decouple objets to the extend that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

Inversion of Control - A design principle in which custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming. (In traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks._(With inversion of control, its the framework that calls into the custom, or task-specific, code.)

Abstract Factory Pattern - Provides an interface for creating families of related or dependent objects without …

Abstract vs Non - Factory - creation through inheritance and creates objects of a single type. Abstract factory - creation through composition, instantiated via new and passed and creates families of related objects via factories.

Midterm review - GIT (revision control). Design patterns. Because we're doing high level complexity Software engineering -> Traditional waterfall model. Scrum, retrospective, roles, (scrum guide). High level picture, interfaces, inheritance, Big 4 OO, Composition vs inheritance. Observer pattern. Decorator pattern (open close principle)(wrap an object). NO ABSTRACT PATTERN. 5 DYSFUNCTIONS AND PYRAMID

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

the **decorator pattern** is a design **pattern** that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

CHAPTER 5 Singleton - need only one of thread pools, caches, dialog boxes, preferences, logging, device drivers, I/O. Pattern - Ensures a class has only one instance and provides global access to it. Threading - synchronize getInstance, eagerly create, double checked locking. Volatile (in code) deals with visibility. It reads and writes happen to main memory. Writes to and reads from volatile all thread visible variables from/to main memory. Visibility and synchronization - Visibility assures that all threads read the same value of a variable. Synchonization makes sure that only one thread can write to variable.

Atomic - any write to volatile variable establishes a happens before relationship with subsequent reads of that same variable, this means that changes to a volatile variables are always visible to other threads. Also, when a thread reads a volatile variable, it sees the side effects of the code that led up to the change not just the latest change. Reads and writes are atomic for reference variables and for most primitive variables (except long and double variables). Reads and writes are atomic for all variables declared volatile (including long and double variables). There are AtomicInt, AtomicLong..

Anything after for example private volatile int days will not be written out. Anything before will. For example private int years; private int months; private volatile int days will write out all three.

Volatile not always enough - if there is a read/modify/write such as variable ++. Must use synchronize keyword to remove race conditions.

JAVA concurrency in practice book for java thread coding.

CHAPTER 6 Command pattern - a behavioral design pattern in which an object is used to encapsulate.. Client is responsible for creating a concrete command and setting its receiver.

Invoker holds a command object and at some point calls it execute() method. Command declares an interface that has at least an execute() method.

CHAPTER 7 Adapter and Facade. Adapter pattern - converts the interface of a class into another interface the clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces. Two types, object adapters use composition and class adapters use inheritance. Facade - provides a unified interface to a set of interfaces in a subsystem, it defines a high-level interface that makes the subsystem easier to use. There difference between these two is that adapter alters and interface to make it usable while facade makes a complicated interface easier to use. Both of these are called principle of least knowledge. Methods may talk to their own object, objects passed as parameters, objects they instantiate, instance variables.

CHAPTER 8 Template method - defines the skeleton of an algorithm in a method. Prepare recipe is a template method, all steps are present, some are handled by base class and some in present class. Hooks can define concrete methods that do nothing unless subclass overrides them. Uses abstract when subclass must implement, hooks when optional. Hollywood principle - low level hooks into system, high level class at the appropriate time. Summary - To prevent subclasses from changing the algorithm, make the template method final. Both the strategy and template patterns encapsulate algorithms. Strategy via composition, Template via inheritance. Factory is a very specialized template, it returns result from subclass.

Chapter 9 Iterator pattern - provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation. This places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be. Design Principle - A class should have only one reason to change (single responsibility principle. High cohesion (all methods related to purpose). Composite pattern - Allows you to compose object into tree structures to represent part/whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly. It can ignore differences between the two, think recursion.

Chapter 10 State pattern - The combination of the value of all the variables in an object. We use state machines all the time (NFAS, DFAS). Automata (combinatorial logic, FSMs, Pushdown, Turing machines). State - vending machines, elevator, locks, traffic, lights, etc. FSMS limited to the amount of memory states it has. State pattern - allows an object to alter its behavior when its internal state changes. The object will appear to change its class. Very similar to strategy pattern, Strategy an alternative to sub classing as it uses composition. State is an alternative to having lots of conditionals.

Chapter 2 Effective Java - Consider static factory methods instead of constructors. One advantage is that they have names, constructors do not and one has to differentiate via parameters. This can be confusing and lead to errors. A class can have only one constructor with a given name. Don't change order of constructor parameters to differentiate. Static factory method don't have to create a new object, constructors always do and maybe there's an object already created that works, helps with immutable classes and pre-constructed. Singletons, flyweights, non instantiable. Can return a subtype, java.util.collections contains all static methods that work on many types. Polymorphic, addAll, binarySearch, disjoint, frequency, min, max, sort, shuffle, reverse. Type returned can be non public. Can vary implementation. Returned class need not exist at the time the class is written, allows run time specification, JDBC example. Disadvantages - classes without public or protected constructors cannot be sub classed. Not call out in JavaDoc.
Popular java static factory name valueOf, getInstance, newInstance, getType, etc.

Consider a builder when faced with many constructor parameters. If a class that has many fields that need initializing. Create empty instance ns have many set()s . Builder pattern, build() is a parameter less static method, required parameters passed in to constructor - optionals set(), other languages have optional parameters instead.

Enforce the singleton property with a private constructor or an enum type.

1.5: Enumeration. Public enum Elvis {

```
        INSTANCE;

        public void leaveTheBuilding(){
                System.out.println("Outta here");
        }
        public static void main(String[] args){
                Elvis elvis = Elvis.INSTANCE;
                elvis.leaveTheBuilding();
}
}
```

Enforce non instantiability with private constructor. Just have a private no args constructor. Class cannot be subclassed, subclasses would need to call the constructor. Might want to have constructor throw an AssertionError just to be safe.

Avoid creating unnecessary objects. Use Iterates and valueOf(). So prefer primitives to boxed primitives, be careful of unintended auto-boxing

Avoid finalizers. Unpredictable, often dangerous generally unnecessary, unlike C++ destructors these are called immediately and java uses try/finally for these types of uses.
One never knows when a finalizer is called, part of garbage collection or might not be called at all.

Chapter 3 Effective java - obey general contract when overriding equals. Sometimes don't need to when all object are unique such as threads, don't need it like rngs, superclass equals works well such as sets, lists, and maps getting from AbstractList, etc. Class is private or package private and you know you don't need it (consider implementing with assertion). WHEN TO IMPLEMENT - when logical equality (.equals) is different from simple object identity (==), when we need the class to be map keys or set elements. .equals compare character by character in a string for example. == asks if same thing while .equal asks if same behavior. Must be reflexive: x.equals(x) must return true. Must be symmetric: x.equals(y) must be true if and only if y.equals(x). Must be transitive: if x.equals(y) is true and y.equals(z) is true, x.equals(z) must be true.

color.java

```
Public enum Color {
        RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

Recipe, check for object == this. Use instance to check for correct type, cast argument all fields. Use @Override

Creating a hash code. Set result = 17 b/c prime number. For all the fields, if boolean, c= f?1:0, if byte, char, short, or int, c = (int)f. If long, c=(int)(f^(f>>>32)). If float, c = Float.floattointbits(f). Update result = 31 * result + c.

Always override toString. It makes class much more pleasant to use.

x.clone() != x, x.clone().getClass() == x.getclass(), x.clone().equals(x)

Consider implementing Comparable. Similar to equals but provides ordering information, generic, useful in Arrays.sort(). x.compareTo(y) == -y.compareTo(x)

Instance field should never be public, it limits typing, limits invariants, and not thread safe. Try to avoid protected too, it exposes implementation detail to subclasses, it should be rare. Arrays are always mutable, never have a public static final array field, or an accessor that returns such a

beast, be careful of ides that create accessors automatically. In public classes, use Accessor methods , not public fields. Minimize mutability, all information provided at construction, any changes result in new objects, don't provide methods an object's state (mutators), ensure class cannot be extended, make all fields final, make all fields private. Immutable objects are thread safe.

Prefer interfaces to abstract classes. Java is single inheritance.

Abstract classes permit multiple implementations, easier to evolve, if want to add a method, can add and implement, everything else still works.

Use interfaces only to define types.

Private Integer i if return i it'll become global and not private anymore.

4 classes - Static, non static, anonymous, local. Last 3 are inner classes. Anonymous class have no names, not a member of enclosing class, declared and instantiated at the same place, permitted where ever expressions are allowed, cannot have static members, useful for creating function objects on the fly.

*EFFECTIVE JAVA 5 GENERICS - Java's type system is very complex, it adds various mechanisms to add "generics", other languages simply have references to objects and duck typing. Generic classes and interfaces are known as generic types. Generic types define sets of parameterized types (List <String>, Raw type is List). Arrays are covariant, if sub is subtype of super, sub[] is a subtype of super[]. Generics are invariant, List<t1> is never a subtype of List<t2>. Arrays vs Generics, Arrays are reified meaning their element types are enforced at runtime, generics are implemented by type erasure meaning types enforced at compile time and erase type at run time.

Use enumerations instead of constant numbers. Java enum provide compile type safety.

Prefer annotations to naming patterns. Unit is a major example. Consistently use @Override because it makes sure you are actually overriding especially for equals, toString, and hashCode.

Use Marker interfaces to define types. A Marker interface is one with no methods. Serializable is an example (indicates object can be written via ObjectOutputStream).

***Effective java Chapter 7 METHODS. Most parameters have restrictions on their validity like positive, non-null, not zero length, etc. AKA preconditions, program defensively, catch problems asap.
Assertions are optional in java, must enable with -ea.
Static{ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);}

Service provide Framework 0 service interface, provider registration. Service chess.

JOBS - Linked in, meet up, denverdevs.org.
Research on the company and position, what tech do they use? If possible, who they bought and been bought by, their strengths and weaknesses of their competitors. OK to be few years below minimum experience level, but not TOO far below, same with education level. Just make sure you can actually DO the job. Fizzbuzz for interview, Follow up the same day from interview, say thanks you know how much work it is thanks for having me etc.
TYPES OF QUESTIONS - computer science Algorithms and data structures, O of algorithms and choosing, imperative and functional, what you've done, language and frameworks. Nerdparadise.com for interview questions. Be communicative. Know OO (abstraction, encapsulation, inheritance, polymorphism). What is interface, what is virtual method, what is multiple inheritance. Deep vs shallow copies, design patterns. Threads vs process. Networks - IP v4/6 addressing, TP vs UDP, Clientserver vs P2P. Software engineering - git, agile manifesto, tdd. Web - MVC, ORM, AJAX

Git

Branching- Need to start branching. A branch for each person, not working on MASTER

- Can have branches of branches (Have to merge when on lower level)
- Git Branch + name of file – creates a brand new reference, not looking at it, but its created
- Switching – Everything I see and do happens to master, to get out of branch use "git checkout name of file"

Scrum

- Maximize teams abilities
- Work on requirements ASAP
- Product owner, Find out which user story if the most important and make sure that is known in the presentation
- Keep product backlogs visible to everyone (working on project)
- Sprint backlog is the list of work the dev team must address during the next sprint
    - Comes from the product backlog and select priority of the product backlogs
    - Once a sprint backlog is committed, no additional work can be added unless the team says so
- Three Pillars
    - Transparency
    - Inspection
    - Adaptation
- Five Values
    - Commitment: Team members commit to achieving team goal each and every sprint
    - Courage: Courage to work through conflict and challenges together to do the right thing
    - Focus: Focus exclusively on the teams goals and sprint backlogs. No work done other than backlog'
    - Openness: Team members and stakeholders agree to be transparent about their work and challenges
    - Respect: Team members respect each other to be technically capable and to work with good intent
- Roles
    - Product Owner
        - Relates to customer
    - Scrum Master
        - Unblocks dev team
    - Dev team
- Retrospective
    - Reflect on previous sprint, go over what can be improved

The Real Syllabus

- Important stuff

- Act confidently and secure
- Respect rights and properties of others
- Work cooperatively
- Courteously respects authority; polite
- Practices self-control
- Accepts responsibility for behavior
- Resolves conflict appropriately
- Exhibits positive attitude toward learning
- Participates in class discussions
- Engaged listener
- Follows all directions
- Completes work in timely manner
- Works independently
- Responsible for property
- Asks for help when needed
- Produces neat, accurate, quality work
- Positive work ethic
- Good organization skill
- Demonstrates technology skills
- Uses library media to enhance learning

AEIP

A- Abstraction

Grouping your code into sections, like classes, objects & APIs where the complex processing is happening "behind the scenes", and all that you need to know or all that communicating objects need to know are a small number of input or output variables

E- Encapsulation

Hides internal details of one object from another

I- Inheritance

Instead of rewriting code, have classes inherit from super classes. This means your program will require less code and be easier to add onto or update.

P- Polymorphism

The ability of one object to be generic and used in many different ways. Taking a different form is called polymorphism

Design Patterns (incomplete- refer to Design Patterns PDF for further info)

General / abstract factories: the dependency Inversion Principle

General rules:

- No variable should hold a reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of it base classes.

Dependency Injection:

- Where one object (or static method) supplies the dependencies of another object.
- A dependency is an object that can be used(a service)
- An injection is the passing of a dependency to a dependent object.
- Passing the service to the client rather than allowing the client to find it.
- Inject information into an object

Inversion of Control (*crucial to perform)

- A system that records changes to a file or a set of files so that you can recall specific versions later on (like GitHub)

Abstract Factory Pattern:

- Creation through inheritance
- Creates object of single types

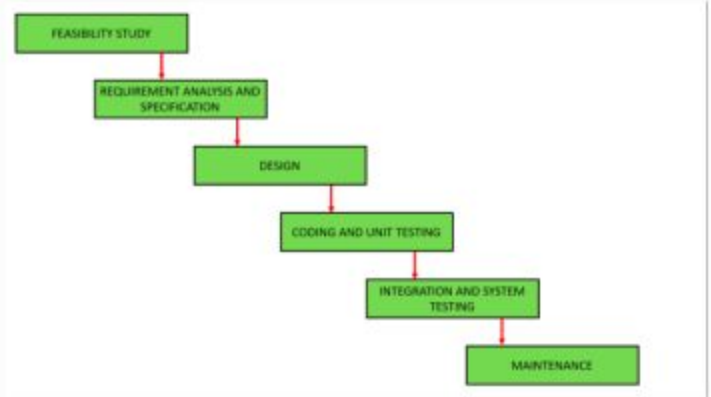
Midterm Review

- Crucial that we version control
    - Not capable of handling the complexity of the code we are writing
- Communication
- Design Patterns
- High Level software engineering, haven't gotten a lot better at it.
    - When he showed up how many lines there were per item.
    - Test driven development
    - Waterfall model (good for small software projects)
- Scrum
    - Retrospective
    - Roles
    - Scrum guide (download that shit)
- Design Patterns
    - Interfaces
    - Big 4 of OO, AEIP
- Observer patterns
    - Subject to which we subscribe
        - Each observer implements and interface, when subject changes the observer is activated
    - Event driven
- Decorator Pattern
    - Open close principle
    - Wrap object inside another object of the same type
- Factory Pattern/No abstract pattern
- AEIP
    - Abstraction
        - Grouping your code into sections, like classes, objects & APIs where the complex processing is happening "behind the scenes", and all that you need to know or all that communicating objects need to know are a small number of input or output variables
    - Encapsulation
        - Hides internal details of one object from another
    - Inheritance
        - Instead of rewriting code, have classes inherit from super classes. This means your program will require less code and be easier to add onto or update.

- Polymorphism
    - The ability of one object to be generic and used in many different ways. Taking a different form is called polymorphism

Software Engineering | Classical Waterfall Model

Classical waterfall model is the basic **software development life cycle** model. It is very simple but idealistic. Earlier this model was very popular but nowadays it is not used. But it is very important because all the other software development life cycle models are based on the classical waterfall model.
Classical waterfall model divides the life cycle into a set of phases. This model considers that one phase can be started after completion of the previous phase. That is the output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other. The different sequential phases of the classical waterfall model are shown in the below figure:



About Version Control

What is version control, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. Even though the examples in this book show software source code as the files under version control, in reality any type of file on a computer can be placed under version control.

If you are a graphic or web designer and want to keep every version of an image or layout (which you certainly would), it is very wise to use a Version Control System (VCS). A VCS allows you to: revert files back to a previous state, revert the entire project back to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also means that if you screw things up or lose files, you can generally recover easily. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control (see Figure 1-1).

1. **Feasibility Study**: The main goal of this phase is to determine whether it would be financially and technically feasible to develop the software.
   The feasibility study involves understanding the problem and then determine the various possible strategies to solve the problem. These different identified solutions are analyzed based on their benefits and drawbacks, The best solution is chosen and all the other phases are carried out as per this solution strategy.
2. **Requirements analysis and specification**: The aim of the requirement analysis and specification phase is to understand the exact requirements of the customer and document them properly. This phase consists of two different activities.
   - **Requirement gathering and analysis:** Firstly all the requirements regarding the software are gathered from the customer and then the gathered requirements are analyzed. The goal of the analysis part is to remove incompleteness (an incomplete requirement is one in which some parts of the actual requirements have been omitted) and inconsistencies (inconsistent requirement is one in which some part of the requirement contradicts with some other part).
   - **Requirement specification:** These analyzed requirements are documented in a software requirement specification (SRS) document. SRS document serves as a contract between development team and customers. Any future dispute between the customers and the developers can be settled by examining the SRS document.
3. **Design**: The aim of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
4. **Coding and Unit testing:** In coding phase software design is translated into source code using any suitable programming language. Thus each designed module is coded. The aim of the unit testing phase is to check whether each module is working properly or not.
5. **Integration and System testing**: Integration of different modules are undertaken soon after they have been coded and unit tested. Integration of various modules is carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained and system testing is carried out on this.
   System testing consists three different kinds of testing activities as described below :
6. **Maintainence:** Maintenance is the most important phase of a software life cycle. The effort spent on maintenance is the 60% of the total effort spent to develop a full software. There are basically three types of maintenance :
   - **Corrective Maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
   - **Perfective Maintenance:** This type of maintenance is carried out to enhance the functionalities of the system based on the customer's request.
   - **Adaptive Maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment such as work on a new computer platform or with a new operating system.

Advantages of Classical Waterfall Model

lassical waterfall model is an idealistic model for software development. It is very simple, so it can be considered as the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model:

- This model is very simple and is easy to understand.
- Phases in this model are processed one at a time.
- Each stage in the model is clearly defined.
- This model has very clear and well understood milestones.
- Process, actions and results are very well documented.
- Reinforces good habits: define-before- design, design-before-code.
- This model works well for smaller projects and projects where requirements are well understood.

Drawbacks of Classical Waterfall Model

lassical waterfall model suffers from various shortcomings, basically we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model. Below are some major drawbacks of this odel:

- **No feedback path:** In classical waterfall model evolution of a software from one phase to another phase is like a waterfall. It assumes that no error is ever committed by developers during any phases. Therefore, it does not incorporate any mechanism for error correction.
- **Difficult to accommodate change requests:** This model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers' requirements keep on changing with time. It is difficult to accommodate any change requests after the requirements specification phase is complete.
- **No overlapping of phases:** This model recommends that new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase the efficiency and reduce the cost, phases may overlap.

Types of Design Patterns

There are mainly three types of design patterns:

1. Creational

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool and Prototype.

2. Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

3. Behavioral

Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

Design Patterns | Set 1 (Introduction)

Design pattern is a general reusable solution or template to a commonly occurring problem in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing tested, proven development paradigm.

Goal:

• Understand the problem and matching it with some pattern.
• Reusage of old interface or making the present design reusable for the future usage.

Example:

For example, in many real world situations we want to create only one instance of a class. For example, there can be only one active president of country at a time regardless of personal identity. This pattern is called Singleton pattern. Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.