# What's new in Java 8 (Pluralsight)

- one use of lambda is to replace anonymous classes
- java.util.function has 43 functional interfaces (across 4 categories)
    - Consumer/BiConsumer - takes an object (or 2) - doesn't return anything
    - Supplier - doesn't take anything, gives an object
    - Predicates/BiPredicate - takes an object (or 2) - returns a boolean
    - Function/BiFunction - takes an object (or 2), returns another object
        - Unary Operator - takes an object, returns same type of object
        - BinarayOperator - takes 2 object of same kind, return same type of object
- Lambdas can have types inferredz,
- Method References
    - System.out::println
- list.forEach (forEach is available for Iterable)
    - if we were to add this in iterable, it will need all implementations to be refactored
    - forEach got added to iterable w/o breaking anything by adding a default impl (only possible in Java 8)
        - Default methods: you can change old interfaces w/o breaking their implementations
        - java 8 interfaces can have static methods too (earlier only fields were allowed)
    - New Patterns based on that
        - Predicates:
            - p1.and(p2) - combining predicates
            - p1.andThen (operations are taken care one after the other)
- Stream Patterns
    - Map/Filter/Reduce
        - Map - takes a list of an object - returns a list of different type (same size)
        - Filter - takes an object, applies a predicate, filters it to a smaller list
        - Reduce - equivalent of sql aggregation - collapse a list into one value
    - Stream: is a java typed interface, stream is not a collection
        - gives a way to efficiently process large amounts of data
            - parallel processing is possible w/o any extra code
            - Pipelined- to avoid unnecessary intermediary computations
    - What is a stream:
        - An object on which one can define operations
        - an object that doesn't hold any data
        - an object that shouldn't change the data it processes (not enforced by compiler or jvm)
        - an object able to process data in one pass
        - optimized from algo point of view, and able to process data in parallel
    - Building a stream:
        - stream method on Collection interface
        - Stream.of()
    - Operations
        - stream.filter(predicate): it can combine predicates using, and, or negate.
            - Filter returns a stream (it's a new instance of stream)
            - predicate has an isEqual - can create a equal predicate using a specific object
        - map - returns a stream (so intermediary)
            - apply - takes an object, returns another object
            - compose and andThen can be used to combine/chain operations
            - identity - takes an object - returns itself
        - flatMap
            - takes an element type T, returns a Stream<T>
                - a map in this case would have returned a stream of streams
        - Reduction - terminal operations - they trigger processing of data
            - min/max, sum, average, count
                - .reduce(??, lambda): ?? is identity element of the reduction operation and second one is of type BinaryOperator<T>
                    - BinaryOperator is a special case of BiFunction
                - max doesn't have an identity element
                - Optionals - wrapper type - implies there might be no result
                    - it has an isPresent value, orelse defines a default value
                - Boolean reductions - allMatch, noneMatch, anyMatch
            - Collectors
                - .collect(takes a Collector here)
                    - Collectors.joining, Collectors.groupingBy, Collectors.toList(),
                    - we can even add a downstream collector to collect method
                -
    - (Stream cannot hold any data)
        - the calls to filter and other methods are lazy, when you write a filter, it's only a declaration
        - an operation on a Stream that returns a Streams is called an intermediary operation
        - final operations - force actual computation on the collection that a stream is connected to
            - forEach is final (peek is the intermediary equivalent of it)
        - one stream can have only one set of operations, two sets will throw a runtime exception

**JavaFX**

- Stage is the main screen <Top Level Window OR Full Screen>
    - Scene sits within a stage
    - Scene holds all the graphical components
        - Setting multiple components in one scene, you will need layout
    - an xml can be used as well to define the layout
    - To load the xml you have to use FXMMLLoader in start method of your application
- Controller does the work when controls are called should implement Initializable
- Supports CSS, touch interfaces, animation etc

**Nashorn**

- Javascript engine in JVM
- REPL: Java in JavaScript (repl-read, eval, print, loop)
- ScriptEngine: Java in JavaScript
- jjs is the command for repl - both java and javascript will work on commandline
- Running javascript in Java code
    - Use ScriptEngineManager
    - you can explicitly make the objects in java scope available in js scope

**Java Date Time**

- Java 7:
    - java.util.Date (and java.sql.Date) - only one constructor - sets the date to now
    - Calendar to create a specific date/time and cal objecg has getTime() to get back a Date
    - Date is mutable - so a value returned can be changed, thereby changing the state of the object containing it, even if it is meant to be immutable
- Java 8
    - package is java.time
    - Instant - a point in timeline
        - Instant 0: 1st Jan 1970, instant.min is 1 billion years ago, Instant.max is dec 31, if year 1 billion, .now is current time
        - Precision is nanosecond
        - immutable
    - Duration
        - time between two instants
    - LocalDate
        - precision is day
    - Period
        - time between two local Dates
    - DateAdjuster
        - useful to add/subtract an amount of time to an instant or a LocalDate
    - LocalTime
        - time in  a day (no date)
    - ZonedTime
        - ZonedDateTime
- Static methods available in most date time apis for initializing the different date time data types
- Formatter
    - DateTimeFormatter - has a set of predefined formats as constants
- Interoperability with older date/time APIs
    - date.toInstant, Date.from(instant) and more such conversion options

**Strings**

- join - StringJoiner - best way for concatenation (java 8 addition), other option is String.join
- from jdk7 - you don't need to use StringBuffer/Builder as string concatenation does it internally (as opposed to older versions, when it would create and delete multiple objects)

**Java IO**

- try with resources
- bufferedReader has .lines()
- Paths.get.. to get a path using different elements of the path
- static method on Files - have a look
- Streams implement AutoClosable and hence try with resources works with it
- Directory - Files.list() (ls on directory) - Fiiles.walk() - goes sub-directories, Files.walk() can take a parameter to restrict to a specific depth

Collection API

- spliterator - a parallel iterator on streas
- new methods on Collections: forEach, removeIf, replaceAll, sort

Comparator

- Java 7 pattern - create an anonymous class
- Java 8: Comparator.comparing(method reference - mostly getter of the key), thenComparing can chain comparators, .reversed to reverse a comparator, Comparator.naturalOrder, .nullsLast

Numbers

- Long::sum - useful in reduction operations

Map

- new methods: forEach (takes BiConsumer as a parameter), getOrDefault, put (erases the existing value), putIfAbsent, replace, more variants are available now, compute, computeIfPresent/Absent

Annotations

- Multiple Annotations - In java 7, a wrapper that takes in the array of annotations is applied
- Java 8 allows multiple annotations
- NonNull declarations for variables @NotNull

---

# Java Fundamentals: The Java Language (Plurlasight)

- JDK vs JRE (dev vs runtime)
- JRE allows it to run seamlessly on all platforms

Statements
- Separator/end of statement is ;
- Spaces and newlines are ignored
- Comments - line comments (//), Block Comments (/* */), Java Doc Comment (/** ... */). Commented stuff is not seen by the compiler
  - JavaDoc util can generate javadocs using the javadoc comment

Packages

- Provide organization
- Follows standard naming convention
  - impacts our source file/folder structure - enforced by IDE, java doesn't need a correlation between package structure and source code file structure
  - Convention - Use lowecase
  - Tries to guarantee uniqueness - Use reverse domain name to assure global uniqueness (just a convention)
    - Add further identifiers to identify areas with an org
- Stuff put in the package is identified using a full name which includes the full package name

Variables

- Named data storage
- Strongly Typed - Variables are typed, value can only take stuff compatible with the type defined
- Allows values to be modified in a variable
- Naming Rules/Conventions
  - Rules allow number, letter, $ and _
  - First character of the name cannot be a number
  - Conventionally we use only letter and numbers
  - convention - camel case - starts with a lowercase letter

Primitive Data Types

- Built into the language
- Foundation of all other types
- 4 categories
  - int - byte(8 bits), short(16), int (32), long(64).. range is -2^(no of bits -1) to -2^(no of bits -1) - 1
    - long literal values should have a L at the end
  - floating point - values with a fractional portion - float (32), double (64 bit)
    - literals - f and d at the end, default is assumed to be d
  - character - single unicode character - 'X', you can use any unicode character (even if it's not in keyboard)
  - boolean - stores true and false
- They are copied/stored by value

Arithmetic Operators - 3 types

- Basic Math operators - + - * / %
  - Divide - sensitive to floating point vs int (pure int operations will drop decimals)
- Prefix/Postfix operators - ++, --
- Compound Assignment operators +=, -=, *=, /= etc

Precedence Rules

- Postfix
- Prefix
- Multiplicative (*,/)
- Additive(+,-)
- move from left to right for equal precedence operations

Type Conversions

- Implicit: Assign a compatible types to each other
  - Widening conversion (safely moved) can be done by compiler automatically
    - mixed operations automatically cast to the widest type in the expressions
      - an error gets generated if lossy conversions come in to play due to assignment to a narrow type
- Explicit: put a cast like (int)
  - can perform both widening and narrowing casts, but you will lose value when you go narrow

Conditional Logic

- Relational operators - gt, lt, gt=, lt=
- Conditional Assignment - res = (condition)? valIfTrue:valIfFalse;
- if-else - conditional execution
- Block Statements - {group of statements} [called a compound statement]
    - Variable scope: a variable declared within is visible only within the block
- Logical Operators - &(and), |(or), (^)XOR (exclusive or - exactly one is true), !(negation)
- Conditional Logical operators (conditional & - &&) and conditional | - ||) executes second statement only if there is any hope of getting a different result

Loops

- while - run until the condition goes to false, checked at the beginning (may not run at all)
- do-while - executes at least once
- for - for(initialize;condition;update){}

Arrays

- ordered collection of elements of common type
- accessed via index (0-based)
- initializing during creation - int[] x = {1,2};
- Looping - for each - for(loop var declaration: arr)

Switch

- simplifies test against multiple values
- from primitives - char and int can be used as test value
- execution falls through multiple cases, unless there is break (avoid falling through)

Classes

- A template for creating an object
- Objects are things that encapsulate data, operations and usage semantics
    - State (Fields) and Behavior (Methods and constructor)
- Allows storage and manipulation details are hidden
- Separates what from how (users only need to know what, not how)

Using Classes

- Declaring a variable of a type (class) only allocates memory
- new creates the object, returns a reference to the object (address to where the actual object is stored)
- Classes are reference types

Encapsulation (hiding internals) and Access Modifiers [Class]

- Internal representation of object should be hidden
- no access modifier - package private - visibility is in package
- public - visible everywhere
- private - from within the class

Naming Classes

- A public class file has to be placed in a file with the classname(.java)
- Pascal Case - Camel case except the first character is uppercase
- Use only letters and numbers
- First character is always a letter
- Use nouns
- Avoid abbreviations unless they are well known

Method Basics

- Executable code that manipulates the state (behavior)
- Naming rules are same as variables
- usually verbs
- Return type - void if nothing is returned
- Typed parameter list (can be empty)
- Exits (3 reasons) - control goes back to the caller
    - reaches the end of the method
    - a return statement is encountered
    - an error occurs
- Return Types - can be primitive, reference to arrays or objects
- Special References
    - this - current object
    - null - reference literal - represents uncreated object, can be assigned to a reference variable

Field Encapsulation

- Fields have the state of an object
- Generally fields are not accessible, methods allow the interaction with fields
- Accessors and Mutators Pattern
    - have getters and setters

Class Initializers and Constructors

- Java provides a default initial state (may not be enough)
- Also it provides mechanism to initialize the state
    - field initializers
    - constructors
        - no return type
        - at least one constructor is needed, if none specified, java provides an empty one
        - code executed during object creation
    - Initialization blocks
- Fields if not initialized will have a default state (zero value based on the datatype, numbers get 0, char - null char, boolean - false, ref types - null)
- Field initializers can provide a custom initial value (literal or expression, can reference other fields too in expressions, can have method calls too)

Constructors - More stuff

- Objects should always be created in some useful state
- a class can have multiple constructors, parameter list needs to be distinct across different constructors
- once a constructor is written by programmer, java won't provide any (the empty one that gets generated when none is specified will no longer be created)
- Chaining
    - a constructor can call another constructor
        - this(param list)
        - this has to be the first line of the constructor code that is calling another constructor
- Visibility
    - access modifiers can be used with constructors too

Initialization Blocks

- Another way to initialize a class, it is added at the beginning of every constructor
- Put the code you want just in curly braces - it is shared across constructors and gets called in the beginning
    - you can put more than one such block and they are executed in the order they appear

Order of calls

- Field initialization
- Initialization block
- Constructor

Parameters

- Parameter Immutability
    - Parameters are pass by value: so any changes within the method have no effect outside
    - Classes are references, so the address is passed as value and the variables within the method point to the same objects
        - so any parameter made to point to a different object within method won't impact the object passed as a parameter
        - any changes to the fields/members in the objects made inside the method will reflect as we are touching the same objects
- Constructor and Method overloading
- Variable number of parameters

Overloading

- Ability to have multiple versions of a method based on parameter list (signature)
- Signature has 3 parts
    - name, number of parameters, type of parameters

Variable number of types

- type... - implies has variable number of parameters here
    - arguments can be passed as an array
    - or as a comma separated list of variables of that type
- can be used in only the last parameter

Inheritance

- extends keyword
    - derived class has the characteristics of base class and can extend it/specialize it
- Instances of derived class can be assigned to the references of the base class
- Methods are available only of the declared Type (not the one used to initialize an object if it's a subclass)
- A derived class can declare a field with the same name as base class, in that case, the field in base class is hidden when the subclass is referenced
    - If the declared type is superclass, then field of superclass is used
    - risky idea
- Override - a method has same signature as a method in superclass
    - method in derived class is called if the instance used is of derived class

    - for compile time checks use @Override annotation

Object Class

- Root of java class hierarchy
- can be used to point to any object created
- methods in object class -> clone, hashCode, getClass, finalize, toString, equals
- universal reference type

Equality

- == - equivalence operator - compares whether two variables point to the same object
- default equals is same as ==

Special References

- super is used to refer of the base class

Controlling Inheritance

- Final -> mark a class as final, you cannot extend it
- method -> mark it final to prevent overriding
- abstract - forces a class to be extended to be used, or a method to be forced to overriden
- if a class has an abstract method, the class needs to be marked as abstract

Inheritance

- Constructors are not inherited
- constructor from base class is mandatory called, if not explicitly then Java calls it (or can be explicitly called using super())

Data Types

- Strings
    - sequence of unicode characters (stored using UTF-16 encoding)
    - immutable
    - modifying a string means -> create a copy, modify and repoint the reference to the new location
    - equals in string does a character by character comparison
    - reference comparisons are fast
    - intern method returns the same object for a string
        - if you have a lot of comparisons to do
    - String.valueOf(type) to get string rep of a type variable, can be done implicitly too (if we do a concat for example)
    - toString is called for objects when we are printing, if tostring is not specified, you get an alphanumeric value representing your instance (classname@address)
- StringBuilder
    - provides mutable string buffer
        - for best performance pre-size the buffer
        - it resizes as needed, but that's inefficient
- Classes vs Primitives
    - Primitives are efficient
    - Objects have overhead
    - Objects have methods
- Primitive wrapper Classes
    - Each primitive has a corresponding wrapper class
    - Boolean, Number(is extended by byte, short, int, long, double, float), Character
    - primitives and their wrapper types are interoperable
    - there is a valueOf method in wrapper class - boxing
    - xxxValue - unboxing (xxx - type)
    - Double.parseDouble(string), Integer.parseInt(string)
    - Wrapper classes provide members too, like min val, max val, infinity and other useful methods
    - Equality - same as objects, so == will do reference comparison, equals will compare the value except for a few types
        - boolean, int, byte, short - a particular value will always point to the same object

Final Fields

- once assigned, cannot be changed
- Static final fields are named constants, cannot be changed ever - use all caps for the name (convention)
    - best to avoid magic numbers in code

Enumeration Types

- type with a finite list of valid values
- declared with enum keyword

Exceptions

- Traditional way of handling errors were intrusive (error codes, flags)

- Exceptions provide a non intrusive way to signal errors
- try (good), catch (error), finally (after both good/error scenario)
    - finally can itself throw exceptions, so either guard against or handle them
- Exception Class Hierarchy
    - Object -> Throwable
        - Error: represents a problem in JVM
        - Exception: related to code we write
            - RuntimeException: (unchecked exceptions) represent errors in your program - not necessary to handle them
            - Directly inheriting from exception (not from runtime) - Checked Exceptions - compilers forces you to handle them
    - catch exceptions go sequentially, derived classes will satisfy a base class in catch statement, so start with more specific one and move to more general ones
- With Methods
    - an unhandled exception gets bubbled up the call stack, to find a point where it can be handled
    - Methods are responsible for any checked exceptions
        - either handle it
        - or declare throws in signature
            - you can put a try and finally, and let exception go up
            - or do nothing
    - overriding - throws clause has to be compatible
        - can exclude exceptions
        - can throw the same exception
        - can have a derived exception
- Throwing Exceptions
    - are objects, create them and throw them
    - during creation provide details of what it is
    - catch an exception and add more to it, and rethrow
        - you can even provide the originating exception in the constructor
- Custom Exception Types
    - inherit from Exception class
    - mostly you just need a constructor

**Packages**

- Group of related types, 3 benefits
    - Create a namespace
    - Provide an access boundary
    - Act as a unit of distribution
- Package name becomes a part of the type name
    - Using reverse domain name can give us uniqueness as domain names are unique (assuming everyone follows this convention)
    - Types that are in the same package they don't need to be qualified, java.lang doesn't need to be specified either
- Importing packages
    - Import on Demand (import x.y.*), every class is checked in the packages imported like this
        - Type conflicts can happen if multiple packages have same types
    - Single Type imports are the preferred way to import

**Access Modifiers**

- private, protected, public, package private

**Packages provide a unit of distribution**

- Each part of package name gets a folder
- Archive Files
    - Takes the entire hierarchy of folders and store it in a file and optionally compress it
    - Manifest - name value pairs, that has information about the package (placed inside a folder called META-INF)
        - like starting point of the package

**Interfaces**

- is a type that defines a contract - no implementation is provided
- Classes implement interfaces (can implement more than one)
- interfaces can extend other interfaces
- Generic Interface
    - example - Comparable can be tied to a type
- declaring - use interface keyword
    - can have methods, but no implementation
    - methods are implicitly pubic
    - constants within an interface - always treated as public final static

**Static members**

- not associated with any individual instance - shared class wide
- declared using static keyword
- static methods can only use static fields
- storage area for static fields is outside the memory for any instance
- static import - "import static ....)
- Static initialization bocks
    - one time type initialization
    - executed before type's first use
        - different from instance initializer that are called for every instance
    - all exceptions need to be handled within the static initialization block

- declaration is similar to instance initialization blocks but with static keyword in the beginning

**Nested Types**

- a type within another type (class in another class/interface, interface in another class/interface)
- Nested types have visibility to private members of the enclosing type
- Nested type themselves can have any visibility
- Structure and scoping:
  - Static nested class: no relationship between the instance of enclosing class and the nested class
  - If nested class is public, can be used outside with enclosing type in the path "enclosing type.nested class"
- Inner Classes
  - Each instance of enclosing class is related to nested class
  - declare the nested class without static keyword
  - it can access the instance of enclosing class with "Enclosing Class.this"
- Anonymous Classes
  - declare class as part of their creation
  - inner classes
  - we create it as an inner class, as if we are creating an instance variable but with curly braces inside which the class definition is created
    - most common use is to create classes that are needed only once, on the fly instead of naming them and creating separately

---

# Random Interview Questions

## Why are SPs bad?

- SQL is not intelligent enough to describe a business processes. You can do it, but it's extremely crap-filled.
- A database is for handling data, not executing higher-level logic
- It's difficult to test SQL

**It mangles the 3 Tier structure**

Instead of having a structure which separates concerns in a tried and trusted way - GUI, business logic and storage - you now have logic intermingling with storage, and logic on multiple tiers within the architecture. This causes potential headaches down the road if that logic has to change.

**Stored procedures are a maintenance problem**

The reason for this is that stored procedures form an API by themselves. Changing an API is not that good, it will break a lot of code in some situations. Adding new functionality or new procedures is the "best" way to extend an existing API. A set of stored procedures is no different. This means that when a table changes, or behaviour of a stored procedure changes and it requires a new parameter, a new stored procedure has to be added. This might sound like a minor problem but it isn't, especially when your system is already large and has run for some time. Every system developed runs the risk of becoming a legacy system that has to be maintained for several years. This takes a lot of time, because the communication between the developer(s) who maintain/write the stored procedures and the developer(s) who write the DAL/BL code has to be intense: a new stored procedure will be saved fine, however it will not be called correctly until the DAL code is altered. When you have Dynamic SQL in your BL at your hands, it's not a problem. You change the code there, create a different filter, whatever you like and whatever fits the functionality to implement.

Microsoft also believes stored procedures are over: it's next generation business framework MBF is based on Objectspaces, which generates SQL on the fly.

**Stored procedures take longer to test**

Business logic in stored procedures is more work to test than the corresponding logic in the application. Referential integrity will often force you to setup a lot of other data just to be able to insert the data you need for a test (unless you're working in a legacy database without any foreign key constraints). Stored procedures are inherently procedural in nature, and hence harder to create isolated tests and prone to code duplication. Another consideration, and this matters a great deal in a sizable application, is that any automated test that hits the database is slower than a test that runs inside of the application. Slow tests lead to longer feedback cycles.

**BL in stored procedures does not scale**

If all the business logic is held in the database instead of the application then the database becomes the bottleneck. Once the load starts increasing the performance starts dropping. With business logic in the application it is easy to scale up simply by adding another processor or two, but that option is not readily available if all that logic is held in the database.

If you have a system with 100's of distributed databases it is far more difficult to keep all those stored procedures and triggers synchronized than it is to keep the application code synchronized.

**Stored procedures are not customisable**

This is a big issue if you want an application where the customer can insert their own business logic, or where different logic is required by different customers. Achieving this with application code is a piece of cake, but with database logic it is a can of worms.

**Database triggers are hidden from the application**

A big problem with database triggers is that the application does not know that they exist, therefore does not know whether they have run or not. This became a serious issue in one application (not written by me) which I was maintaining. A new DBA who was not aware of the existence of all these triggers did something which deactivated every trigger on the main database. The triggers were still there, they had not been deleted, but they had been turned off so did not fire and do what they were supposed to do. This mistake took several hours to spot and several days to fix.

**Version Control**

It is easy to control all changes to application code by running it through a proper version control system, but those facilities do not exist for stored procedures and triggers. How much damage could be caused if a stored procedure were to get out of sync with the application code? How easy is it to check that the application is running with the correct versions? How much more difficult would it be if the application you were supporting was running on a remote site with nothing more than a dial-up connection?

This is a reason why some teams avoid stored procedures like the plague - it eliminates an area of potentially disastrous screw-ups.

**Vendor lock-in**

You may think that this is not a problem if you build and maintain the applications for a single company where a change in database vendor is highly unlikely, but what happens should the company decide that their DBMS is no longer flavour of the month and they want to change to a different DBMS? This may be due to several factors, such as spiraling costs or poor performance, but when it happens you will find that a lot of code will have to be rewritten. Porting the data will be one exercise, but porting the stored procedures and triggers will be something else entirely. Now, if all that logic were held inside the application, how much simpler would it be?

Believe it or not there are people out there who write applications which are database-independent for the simple reason that the applications may be used by many different companies, and those many companies may not all use the same DBMS. Those that do use the same DBMS may not be using the same version, and stored procedures written for one version may not be compatible with another.

---

# Effective Java Second Edition

- Instance controlled classes: Classes that maintain strict control over what instances exist at any time
- Service Provider Framework: system in which multiple service providers implement a service, and the system makes the implementations available to clients, decoupling them from the implementations. 3 essential components
  - service interface (which providers implement)
  - provider registration API (system uses this to register implementations giving clients access to them)
  - service access API - used by clients to obtain an instance of the service
- Telescoping constructors - tries to have constructors with different number of arguments to cover different combinations of required and optional parameters to build the object (not great with more number of parameters)
- JavaBeans Pattern: Use an empty constructor and then call setters to set the fields.
  - Can lead to inconsistent state for the objects (midway during the calls, as the object is created bit by bit)
  - Cannot make the class immutable
- Builder pattern simulates named optional parameters
- Class.newInstance tries to build a object but wouldn't fail compilation if there is no accessible constructor, an exception will be throws at runtime (illegalAccess or instantiationException)

### Item 1: Consider static factory methods instead of constructors

- Static factory methods have more informative names than constructors
- Same parameters list could be applied
- Not required to create new objects, could return cached instance
- Static factory methods could return object subtype
- Reduced verbosity for generics due to type inference
- Classes without public/private constructor can't be subclassed, but it is good, because it enforces to "favor composition over inheritance"
- Hard to distinguish from other static methods. To avoid confusion use common names like `newIntance`, `valueOf`, etc.

### Item 2: Consider a builder when faced with many constructor parameters

- **Telescope Constructor** causes verbosity
- **JavaBeans** may cause inconsistent state, no possibbility to make a class immutable
- **Builder Pattern** is flexible and right way to handle optional parameters

### Item 3: Enforce the singleton property with a private constructor or an enum type

*Caution:* Discussed singleton **without** lazy initialization

- Throw an exception in a private constructor to avoid reflection call to constructor
- If standard serialization is needed make all fields transient and override `readResolve` method
- Best way to use single element enum as a singleton

```
enum Singleton {
  INSTANCE}
```

### Item 4: Enforce noninstantiability with a private constructor

- Include a single private constructor to a class to prevent it from instantiation
- Throw an exception in constructor if it is called
- Almost always used technique for utility classes

### Item 5: Avoid creating unnecessary objects

- `"hello"` is better than `new String("hello")`
- `Boolean.valueOf("true")` is better than `new Boolean("true")`
- Immutable objects could be reused for free
- Mutable objects could be reused if you *really* sure, they won't be modified
- prefer primitives to boxed primitives
- watch out for hidden autoboxing

### Item 6: Eliminate obsolete object references

- Garbage collector is not savior from memory leaks
- Nullify obsolete references
- Invalidate cache periodically
- Deregister outdated listeners and callbacks

### Item 7: Avoid finalizers

- Finalizers are not destructors
- No guarantee finalizers will be executed promptly
- No guarantee finalizers will be executed at all
- `System.gc` just a hint, not a gc call
- Finalizers cause **severe** performance penalty
- Use own explicit methods for finalization like `close()`

### Item 8: Obey the general contract when overriding equals

- Overridden `equals` should follow *equivalence relation*
  - Reflexive, `x.equals(x) == true`
  - Symmetric, `x.equals(y) == y.equals(x)`
  - Transitive, `x.equals(y) and y.equals(z) == x.equals(z)`
  - Consistent, consequent calls `x.equals(y)` should produce the same value
  - `x.equals(null) == false` if `x` not null
- There is no way to extend an instantiable class and add a value component while preserving the equals contract
- Do not write an equals method that depends on unreliable resources. This was discussed at the Java Magic: Part I
- Always override `hashCode` when you override `equals`
- Don't substitute param type in `equals`, that cause method **overload** instead of **override**. Use `@Override`annotation to be safe.

### Item 9: Always override hashCode when you override equals

- Equal objects must have equal hashcodes
- Unequal objects could have equal hashcodes
- Missing `hashCode` implementation breaks functionality of hash-based collections
- The worst possible legal hash function `return 42`
- Bad hashcode could degrade performance in hash-based collections
- Hashcode could be cached for immutable classes
- Do not try to develop your own `state-of-the-art` hash function unless you are a mathematician

### Item 10: Always override toString

- Actually, not required
- Easier to inspect objects
- Include all needed info to `toString`
- Document what exactly `toString` returns and in which format
- Provide programmatic access to all of the information contained in the value returned by `toString`

### Item 11: Override clone judiciously

- `Cloneable` is broken
- If you override the `clone` method in a nonfinal class, you should return an object obtained by invoking `super.clone`
- Class that implements `Cloneable` is expected to provide a properly functioning public `clone` method
- `clone` should not corrupt original object
- Pay attention to *deep* and *shallow* copy
- Provide copy constructor or copy factory instead of implementing `clone`

### Item 12: Consider implementing Comparable

- Implementing `Comparable` indicates that objects have natural ordering
- `Comparable` allow to use your class in many generic algorithms: search, sorting, etc.
- `compareTo` should be consistent with `equals`
- Implement `compareTo` that returns `-1`, `0` and `1` and do not cause integer overflow
- For non-natural ordering or inability to implement `Comparable` use `Comparator`

### Item 13: Minimize the accessibility of classes and members

- Make each class or member as inaccessible as possible
- If a package-private top-level class is used by only one class, consider making the top-level class a private nested class of the sole class that uses it
- If a method overrides a superclass method, it is not permitted to have a lower access level in the subclass than it does in the superclass
- Instance fields should never be public
- Classes with public mutable fields are not thread-safe
- public static final arrays are mutable

### Item 14: In public classes, use accessor methods, not public fields

- public fields are acceptable if class is not public
- if a class is accessible outside its package, provide accessor methods

### Item 15: Minimize mutability

- Immutable classes are easier to design, implement and use. They are less error-prone and more secure
- To make a class immutable follow the rules
    - Don't provide any mutators
    - Ensure that the class can't be extended
    - Make all fields `final`
    - Make all fields `private`
    - Ensure exclusive access to any mutable components. Return defensive copies
- Immutable objects are thread-safe
- Immutable objects are shared freely
- Not only can you share immutable objects, but you can share their internals
- The only real disadvantage of immutable classes is that they require a separate object for each distinct value
- Classes should be immutable unless there's a very good reason to make them mutable.
- If a class cannot be made immutable, limit its mutability as much as possible.
- Make every field final unless there is a compelling reason to make it nonfinal

### Item 16: Favor composition over inheritance

- Unlike method invocation, inheritance violates encapsulation
- Subclasses depend on their superclasses, which could be changed and as result broken functionality in subclasses
- Use composition and forwarding instead of inheritance, especially if an appropriate interface to implement a wrapper class exists.
- Use inheritance when class is designed for inheritance

### Item 17: Design and document for inheritance or else prohibit it

- Class must document its self-use of overridable methods
- Good API documentation *for inheritance* should describe what a given method does and how it does it.
- The only way to test a class designed for inheritance is to write subclasses
- Constructors must not invoke overridable methods
- If supperclass implements `Cloneable` or `Serializable` neither `clone` nor `readObject` may invoke an overridable method, directly or indirectly
- Prohibit subclassing in classes that are not designed and documented to be safely subclassed
- Prohibit subclassing by making class final

### Item 18: Prefer interfaces to abstract classes

- Existing classes can be easily retrofitted to implement a new interface
- Interfaces are ideal for defining mixins
- Interfaces allow the construction of nonhierarchical type frameworks
- Interfaces enable safe, powerful functionality enhancements via wrapper classes
- Abstract classes are useful for sceletal implementation
- You could safely add a method to abstract class with default implementation (*the same applies to interfaces since Java 8 release, with help of default methods*)

- Once an interface is released and widely implemented, it is almost impossible to change

### Item 19: Use interfaces only to define types

- Do not use interface for defining constants
- If in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility
- If a nonfinal class implements a constant interface, all of its subclasses will have their namespaces polluted by the constants in the interface
- Add constant to class if they are strongly tied to it
- Make constants as enum or noninstantiable utility classes

### Item 20: Prefer class hierarchies to tagged classes

- Tagged class use internal state to indicate its type
- Tagged classes are verbose, error-prone, and memory inefficient
- Hierarchy classes provide more compile time checks

### Item 21: Use function objects to represent strategies

- Function objects are simulate functions in OOP
- Function objects should be stateless
- Primary use of function objects is to implement the Strategy pattern

### Item 22: Favor static member classes over nonstatic

- A nested class should exist only to serve its enclosing class
- There are four kinds of nested classes
    - Static member classes
    - Nonstatic member classes
    - Anonymous classes
    - Local classes
- Static member classes could exist without enclosing *instance*
- If you declare a member class that does not require access to an enclosing instance, always put the static modifier in its declaration
- Anonymous classes could be instantiated *only* at the point they are declared
- Anonymous classes have enclosing instances if they are defined ina nonstatic context
- Local classes can be declared anywhere a local variable can be declared and have the same scoping rules

### Item 23: Don’t use raw types in new code

- Generic types provide compile-time checking for incompatible types
- Not needed manual cast to type when you retrieve element from collections
- Raw types exists only for backward compatibility
- You lose type safety if you use a raw type
- Raw types could be used with `instanceof` operator

### Item 24: Eliminate unchecked warnings

- Eliminate every unchecked warning that you can, that means your code is typesafe
- Use `@SuppressWarnings("unchecked")` only if you can prove the code is typesafe
- Always use the `@SuppressWarnings` annotation on the smallest scope possible
- Every time you use an `@SuppressWarnings` annotation, add a comment saying why it’s safe to do so
- Every unchecked warning represents the potential for a `ClassCastException` at runtime. Do not ignore them blindly

### Item 25: Prefer lists to arrays

- Arrays are covariant (if `Sub` is subtype of `Super`, then `Sub[]` is a subtype of `Super[]`)
- Generics are invariant (`List<Sub>` is not a subtype of `List<Super>`)
- Arrays are reified (enforce their element types at runtime)
- Generics are non-reified and implemented by erasure (enforce types at a compile time, but erased at a runtime)
- Generic array creation errors at compile time (`List<E>[]`)
- Array of non-reified types can not be created

### Item 26: Favor generic types

- `Object` type in collections are good candidate to replace with generic types

- `new E[]` cause compile time error, use `(E[]) new Object[]` instead

### Item 27: Favor generic methods

- Generic type parameter list, which declares the type parameter, goes between the method's modifiers and its return type (`public static <T> void method()`)
- Generic methods could infere type of arguments

### Item 28: Use bounded wildcards to increase API flexibility

- Generics are invariant (`List<Integer>` is not a subtype of `List<Number>`)
- For maximum flexibility, use wildcard types on input parameters that represent producers or consumers
- **PECS**: **P**roducer - **E**xtends , **C**onsumer - **S**uper
- Producer: `add(List<? extends Number>)`
- Consumer: `get(List<? super Number>)`
- `Comparable` and `Comparator` are consumers
- Do not use wildcard types as return types, it would force use wildcards in the client code
- Use explicit types if compiler can't infere them `Union<Number>.union()`
- if a type parameter appears only once in a method declaration, replace it with a wildcard

### Item 29: Consider typesafe heterogeneous containers

- Single-element containers could be parametrized (`ThreadLocal`, `AtomicReference`)
- `String.class` is of type `Class<String>`
- Typesafe heterogeneous container pattern

```
public class Favorites {
  public <T> void putFavorite(Class<T> type, T instance);
  public <T> T getFavorite(Class<T> type);}
```

- `String s = f.getFavorite(String.class)` is typesafe
- You can use `Class` objects as keys for typesafe heterogeneous containers.

### Item 30: Use enums instead of int constants

- *int enum pattern* just a class with int constants
- Compiler won't complain if you pass one int constant, where another expected
- If int constant number is changed, clients should be recompiled
- There is no easy way to translate int enum constants into printable strings
- There is no reliable way to obtain size or iterate over all the int enum constants in a group
- *String enum pattern* is even worse
- String comparisons is expensive
- Error is string constant lead to runtime error
- Use enums!
- Each enum internally is `public static final int` field
- Enums provide compile-time type safety
- Enum types with identically named constants coexist peacefully because each type has its own namespace
- You can add or reorder constants in an enum type without recompiling clients
- Translate enums into printable strings by calling their `toString` method
- Enum types let you add arbitrary methods and fields and implement arbitrary interfaces
- If an enum is generally useful, it should be a top-level class; if its use is tied to a specific top-level class, it should be a member class of that top-level class
- To avoid switch on enum constant use *constant-specific method implementations*. Add abstract method to enum type, and override that method for each constant
- Enums have auto-generated methods `valueOf(String)`, `values()`
- Switches on enums are good if you are client user of that enum

### Item 31: Use instance fields instead of ordinals

- Every enum constant associated with int value via `ordinal()` method
- Reordering, adding or deleting enum constant cause problems if you depend on `ordinal()`
- Use instance fields for enum (`APPLE(1)` instead of `APPLE.ordinal()`)

### Item 32: Use EnumSet instead of bit fields

- *Bit int enum pattern* use int constants as a power of two (1,2,4,8,...) This lets you to perform union and intersection with bitwise operations efficiently (`apply(STYLE_ITALIC | STYLE_BOLD)`)
- Hard to interpret bit int constants

- Hard to iterate over bit int constants
- Just because an enumerated type will be used in sets, there is no reason to represent it with bit fields
- Use `EnumSet` instead (`apply(EnumSet.of(Style.ITALIC, Style.BOLD))`)

### Item 33: Use EnumMap instead of ordinal indexing

- `ordinal()` for enums cause a lot of problems in array indexing
- ordinal indexing is not typesafe, may cause wrong associations or `IndexOutOfBoundsException`
- Use `EnumMap.get(APPLE)` instead of `array[APPLE.ordinal()]`
- If the relationship that you are representing is multidimensional, use `EnumMap<..., EnumMap<...>>`

### Item 34: Emulate extensible enums with interfaces

- There is no much useful use cases to extend enum functionality
- `enum` could inplement interfaces, therefore allow extensibility
- While you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface

### Item 35: Prefer annotations to naming patterns

- Prior to release 1.5, it was common to use naming patterns to indicate that some program elements demanded special treatment by a tool or framework (name test methods beginning with `test` for JUnit)
- No warning about typos, no control over program elements, ugly and fragile approach
- Annotations solve *naming patterns* problems
- Define annotation Test `public @interface Test`
- `@Retention(RetentionPolicy.RUNTIME)` meta-annotation indicates that Test annotations should be retained at runtime
- `@Target(ElementType.METHOD)` meta-annotation indicates that the Test annotation is legal only on method declarations
- Process marker annotations `Method.isAnnotationPresent(Test.class)`
- With the exception of toolsmiths, most programmers will have no need to define annotation types
- Consider using any annotations provided by your IDE or static analysis tools

### Item 36: Consistently use the Override annotation

- `@Override` can only be used on method declarations
- `@Override` indicates that the annotated method declaration overrides a declaration in a supertype
- `@Override` helps to catch tricky bugs (overloaded `equals`, `hashcode`)
- Use the `@Override` annotation on every method declaration that you believe to override a superclass declaration

### Item 37: Use marker interfaces to define types

- A marker interface is an interface that contains no method declarations (`Serializable`, `Cloneable`)
- Marker interfaces are not marker annotations
- Marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not
- Marker interfaces can be targeted more precisely by extending that interface
- `Set` is *restricted marker interface*. It extends `Collection` but does not add new methods. It only refines contract for some methods to be more targeted.
- The chief advantage of marker annotations over marker interfaces is that it is possible to add more information to an annotation type after it is already in use, by adding one or more annotation type elements with defaults (Java8 default methods)
- If you find yourself writing a marker annotation type whose target is `ElementType.TYPE`, take the time to figure out whether it really should be an annotation type, or whether a marker interface would be more appropriate.

### Item 38: Check parameters for validity

- Detect errors and wrong values as soon as possible
- For public methods, use the Javadoc `@throws` tag to document the exception that will be thrown if a restriction on parameter values is violated
- nonpublic methods should generally check their parameters using assertions
- Unlike normal validity checks, they have no effect and essentially no cost unless you enable them, which you do by passing the `-ea` (or `-enableassertions`) flag to the java interpreter
- Do not use validity check if it is impractical or performed implicitly in the process of doing the computation

### Item 39: Make defensive copies when needed

- You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants
- If you return mutable reference from class, then class is also mutable
- To make immutable class, make a defensive copy of each mutable parameter
- Defensive copies are made before checking the validity of the parameters and the validity check is performed on the copies rather than on the originals (protect against changes from another thread, TOCTOU *time-of-check/time-of-use* attack)
- Do not use the clone method to make a defensive copy of a parameter whose type is subclassable by untrusted parties
- Defensive copying of parameters is not just for immutable classes, think twice before returning a reference

- Arrays are always mutable
- Defensive copying can have a performance penalty associated with it and isn't always justified
- If the cost of the defensive copy would be prohibitive and the class trusts its clients not to modify the components inappropriately, then the defensive copy may be replaced by documentation outlining the client's responsibility not to modify the affected components

### Item 40: Design method signatures carefully

- Choose method names carefully
- Follow the code conventions
- Too many methods make a class difficult to learn, use, document, test, and maintain
- Avoid long parameter lists (four parameters or fewer)
- Use builder pattern, helper classes or helper methods to avoid long parameter lists
- For parameter types, favor interfaces over classes
- Prefer two-element enum types to boolean parameters

### Item 41: Use overloading judiciously

- The choice of which overloaded method to invoke is made at **compile time**
- Selection among overloaded methods is static, while selection among overridden methods is dynamic
- A safe, conservative policy is never to export two overloadings with the same number of parameters
- The rules that determine which overloading is selected are extremely complex. They take up thirty-three pages in the language specification [JLS, 15.12.1-3]

### Item 42: Use varargs judiciously

- Use `call(int...)` when you need zero or more arguments
- Use `call(int, int...)` when you need one or more arguments
- Don't use varargs for every method that has a final array parameter; use varargs only when a call really operates on a variable-length sequence of values
- Every invocation of a varargs method causes an array allocation and initialization
- Use overloaded methods with 2, 3, 4 params to cover most use-cases, otherwise use varargs

### Item 43: Return empty arrays or collections, not nulls

- Do not return `null`s!
- Return empty collection (`Collections.emptyList()`), or zero-length array (`new int[0]`) instead of nulls
- Zero-length arrays and empty collections are not performance problems, because they are immutable and only one instance could be used

### Item 44: Write doc comments for all exposed API elements

- Document API with the `javadoc` utility
- To document your API properly, you must precede every exported class, interface, constructor, method, and field declaration with a doc comment
- If a class is serializable, you should also document its serialized form
- The doc comment for a method should describe succinctly the contract between the method and its client
- With the exception of methods in classes designed for inheritance, the contract should say **what** the method does rather than **how** it does its job.
- Methods should document pre- and postconditions, side effects, thread safety, exceptions
- The first "sentence" of each doc comment should be the summary description
- When documenting a generic type or method, be sure to document all type parameters
- When documenting an enum type, be sure to document the constants
- When documenting an annotation type, be sure to document any members

### Item 45: Minimize the scope of local variables

- The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used
- Nearly every local variable declaration should contain an initializer (`try-catch` block is an exception)
- `for` loop allows to declare loop variable, prefer it over `while`
- Keep methods small and focused

### Item 46: Prefer for-each loops to traditional for loops

- `foreach` saves from subtle bugs, copy-paste errors, improves readability and maintainability
- `foreach` introduces no performance penalty
- Implement `Iterable` interface to use custom class in `foreach` loop
- `foreach` is not used in
  - filtering (no access to iterator to call `remove`)
  - transforming (no access to index element to apply change)
  - parallel iteration (needed few iterators and control locks)

### Item 47: Know and use the libraries

- By using a standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you
- If a flaw were to be discovered, it would be fixed in the next release
- With using libraries you don't have to waste your time writing ad hoc solutions to problems that are only marginally related to your work
- Performance of standard libraries tends to improve over time, with no effort on your part
- Libraries tend to gain new functionality over time

### Item 48: Avoid float and double if exact answers are required

- The `float` and `double` types are not suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a `float` or `double` exactly
- Use `BigDecimal`, `int`, or `long` for monetary calculations
- `BigDecimal` has full control over rounding
- If performance is crucial, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use `int` or `long`

### Item 49: Prefer primitive types to boxed primitives

- Primitives have only their values, whereas boxed primitives have identities distinct from their values.
- Boxed primitive may have `null` value
- Primitives more time and space-efficient than boxed primitives
- Applying the `==` operator to boxed primitives is almost always wrong
- When you mix primitives and boxed primitives in a single operation, the boxed primitive is auto-unboxed,
- Use boxed primitives as type parameters in parameterized types
- Use boxed primitives when making reflective method invocations

### Item 50: Avoid strings where other types are more appropriate

- Strings are poor substitutes for other value types (`5` is better than `"5"`)
- Strings are poor substitutes for enum types
- Strings are poor substitutes for aggregate types; to access individual fields you must parse string

### Item 51: Beware the performance of string concatenation

- Using the string concatenation operator repeatedly to concatenate n strings requires O(n^2) time
- To achieve acceptable performance, use a `StringBuilder` instead

### Item 52: Refer to objects by their interfaces

- If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types
- If you depend on any special properties of an implementation, document these requirements where you declare the variable

### Item 53: Prefer interfaces to reflection

- Reflection provides programmatic access to the class's member names, field types, method signatures, etc.
- Using reflection have disadvantages
    - No compile-time type checking
    - Code verbosity
    - Performance suffers
- As a rule, objects should not be accessed reflectively in normal applications at runtime
- Create instances reflectively and access them normally via their interface or superclass
- Using reflection cause compiler warnings

### Item 54: Use native methods judiciously

- The Java Native Interface (JNI) allows Java applications to call native methods, which are special methods written in native programming languages such as C or C++
- JNI has three main uses
    - Access to platform-specific facilities such as registries and file locks
    - Access to libraries of legacy code, which could in turn provide access to legacy data
    - Used to write performance-critical parts of applications
- Do not use native methods for improved performance

- Applications using native methods have disadvantages
  - programs are not immune to memory corruption errors
  - less portable
  - difficult to debug

### Item 55: Optimize judiciously

- Premature optimization is the root of all evil
- Strive to write good programs rather than fast ones
- Strive to avoid design decisions that limit performance
- Consider the performance consequences of your API design decisions
- Measure performance before and after each attempted optimization

### Item 56: Adhere to generally accepted naming conventions

- Typographical
  - Package: `com.google.inject`, `org.joda.time.format`
  - Class/Interface: `Timer`, `FutureTask`, `LinkedHashMap`, `HttpServlet`
  - Method/Field: `remove`, `ensureCapacity`, `getSrc`
  - Constants: `MIN_VALUE`, `NEGATIVE_INFINITY`
  - Local variable: `i`, `href`, `houseNumber`
  - Type parameter: `T`, `E`, `K`, `V`, `T1`, `T2`
- Grammatical
  - Class - *noun*: `Timer`, `Task`
  - Interface - *noun*, *adjective* ends with *able*: `Comparator`, `Comparable`
  - Annotation - *noun*, *verb*, *preposition*, *adjective*: `@Test`, `@Autowired`, `@ImplementedBy`, `@ThreadSafe`
  - Method - *verb*, rarely *noun*: `drawImage`, `getDimension`, `isInterrupted`, `size`
  - Field - *noun*: `height`, `capacity`
- Conventions should not be followed blinfly if long-held conventional usage dictates otherwise

### Item 57: Use exceptions only for exceptional conditions

- Exceptions slower than normal checks
- Placing code inside a try-catch block inhibits certain optimizations that modern JVM implementations might otherwise perform
- A well-designed API must not force its clients to use exceptions for ordinary control flow

### Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

- Java provides three kinds of throwables: checked exceptions, runtime exceptions, and errors
- Checked exceptions *force* the caller to handle them
- Use checked exceptions for conditions from which the caller can reasonably be expected to recover
- Unchecked exceptions indicate programming error and not needed to be handled *almost* always
- Errors indicated JVM problems and not needed to be handled at all

### Item 59: Avoid unnecessary use of checked exceptions

- Checked exceptions *force* the caller to handle them in `try-catch` block, or propagate outward
- If catching exception provide no benefit (*recovery*, *logging*) use unchecked exception
- One technique for turning a checked exception into an unchecked exception is to break the method that throws the exception into two methods, additional method returns a boolean that indicates whether the exception would be thrown

### Item 60: Favor the use of standard exceptions

- `IllegalArgumentException` caller passes in an argument whose value is inappropriate (e.g. negative value for square root)
- `IllegalStateException` invocation is illegal because of the state of the receiving object (e.g. partially initialized object)
- `NullPointerException` and `IndexOutOfBoundsException` are more specific versions of `IllegalArgumentException`
- `ConcurrentModificationException` object that was designed for use by a single thread or with external synchronization detects that it is being (or has been) concurrently modified.
- `UnsupportedOperationException` used by implementations that fail to implement one or more optional operations defined by an interface

### Item 61: Throw exceptions appropriate to the abstraction

- **Exception translation**. Higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction

```
try {
  // Use lower-level abstraction to do our bidding} catch(LowerLevelException e) {
  throw new HigherLevelException();}
```

- **Exception chaining**. Higher-level exception could contain reference to lower-level exception (e.g for debugging)
- While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused

### Item 62: Document all exceptions thrown by each method

- Always document checked exceptions with javadoc `@throws` tag
- Do not document multiple excetions by their common superclass (`@throws Exception` is bad)
- Document *expected* unchecked exceptions with javadoc `@throws` tag
- Do not include unchecked exceptions in method `throws` declaration
- If an exception is thrown by many methods in a class for the same reason, it is acceptable to document the exception in the class's documentation comment

### Item 63: Include failure-capture information in detail messages

- To capture the failure, the detail message of an exception should contain the values of all parameters and fields that "contributed to the exception"
- To avoid verbosity, include only *useful* information to exception message
- Exception detail message for programmers, not for users

### Item 64: Strive for failure atomicity

- **Failure atomicity** (failed method invocation should leave the object in the state that it was in prior to the invocation)
- If an object is immutable, failure atomicity is free
- If ab object is mutable
  - Check parameters for validity before performing the operation
  - Perform partial computations and then check for validity
  - Write recovery code to return object to its original state after exception
  - Make temporary code, apply changes and then replace original object if no exceptions are thrown
- Failure atomicity is not always desirable (implementation complexity, performance)
- If method is not failure atomic, reflect that in documentation

### Item 65: Don't ignore exceptions

- Don't ignore exceptions!
- An empty catch block defeats the purpose of exceptions
- At the very least, the catch block should contain a comment explaining why it is appropriate to ignore the exception

### Item 66: Synchronize access to shared mutable data

- Synchronization prevent a thread from observing an object in an inconsistent state
- Synchronization ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock
- Reading or writing a variable is atomic unless the variable is of type `long` or `double`
- Do not use `Thread.stop`
- A recommended way to stop one thread from another is to have the first thread poll a `boolean` field that is initially `false` but can be set to `true` by the second thread to indicate that the first thread is to stop itself
- **Liveness failure** - the program fails to make progress
- Synchronization has no effect unless both read and write operations are synchronized
- Increment operator (++) is not atomic
- **Safety failure** - the program computes the wrong results
- Strive to assign mutable data to a single thread

### Item 67: Avoid excessive synchronization

- To avoid liveness and safety failures, never give control to the client within a synchronized method or block
- *Alien* methods may cause data corruption, deadlocks
- Java locks are reentrant
- `CopyOnWriteArrayList` is a variant of `ArrayList` in which all write operations are implemented by making a fresh copy of the entire underlying array

- Do as little work as possible inside synchronized regions
- When in doubt, do not synchronize your class, but document that it is not thread-safe

### Item 68: Prefer executors and tasks to threads

- Executor framework separates unit of work (task) and mechanism (thread creation)
- `Thread` is no longer a key abstraction, use `Runnable` or `Callable`
- `Executors.newCachedThreadPool` good choice for lightly-loaded server, if no threads available for submitted task, new one will be created
- `Executors.newFixedThreadPool(n)` good choice for heavily-loaded server, Only `n` threads will be created

### Item 69: Prefer concurrency utilities to wait and notify

- Prefer higher-level concurrency utilities (Executor Framework, concurrent collections and synchronizers) to `wait` and `notify`
- `ConcurrentHashMap` is optimized for retrieval operations
- Use `ConcurrentHashMap` in preference to `Collections.synchronizedMap` or `Hashtable`
- `BlockingQueue` used for producer-consumer queues
- `CountdownLatch` is a single-use barrier that allow one or more threads to wait for one or more other threads to do something
- For interval timing, always use `System.nanoTime` in preference to `System.currentTimeMillis`. `System.nanoTime` is both more accurate and more precise, and it is not affected by adjustments to the system's real-time clock.
- `CyclicBarrier` could be used if you need multiple `CountDownLatch` objects
- Always use the *wait loop* idiom to invoke the `wait` method; never invoke it outside of a loop

### Item 70: Document thread safety

- The presence of the synchronized modifier in a method declaration is an implementation detail, not a part of its exported API
- To enable safe concurrent use, a class must clearly document what level of thread safety it supports

  - **immutable** - Instances of this class appear constant. No external synchronization is necessary
  - **unconditionally thread-safe** - Instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization
  - **conditionally thread-safe** - Like unconditionally thread-safe, except that some methods require external synchronization for safe concurrent use. Examples include the collections returned by the Collections.synchronized wrappers, whose iterators require external synchronization
  - **not thread-safe** - Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients' choosing
  - **thread-hostile** - This class is not safe for concurrent use even if all method invocations are surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a thread-hostile class on purpose; such classes result from the failure to consider concurrency

- To prevent DOS attack, you can use a private lock object instead of using synchronized methods

### Item 71: Use lazy initialization judiciously

- Best advice for lazy initialization is "don't do it unless you need to"
- If you use lazy initialization to break an initialization circularity, use a synchronized accessor
- If you need to use lazy initialization for performance on a static field, use the *lazy initialization holder class* idiom
- If you need to use lazy initialization for performance on an instance field, use the *double-check* idiom

### Item 72: Don't depend on the thread scheduler

- Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.
- The best way to write a robust, responsive, portable program is to ensure that the average number of *runnable* threads is not significantly greater than the number of processors
- Do not rely on `Thread.yield` because it has no testable semantics
- Use `Thread.sleep(1)` instead of `Thread.yield()` for concurrency testing
- Thread priorities are among the least portable features of the Java platform

### Item 73: Avoid thread groups

- Thread groups do not provide any security functionality
- Thread API is weak
- Prior to release 1.5, there was one small piece of functionality that was available only with the ThreadGroup API the `ThreadGroup.uncaughtException` method was the only way to gain control when a thread threw an uncaught exception. As of release 1.5, however, the same functionality is available with `Thread.setUncaughtExceptionHandler` method.

### Item 74: Implement Serializable judiciously

- Implementing `Serializable` decreases the flexibility to change a class's implementation once it has been released

    - `private` and `package-private` fields become part of its exported API
    - incompatible changes after deserialization lead to failure

- Define `serialVersionUID` to avoid `InvalidClassException`
- Because there is no explicit constructor associated with deserialization, it is easy to forget that you must ensure that it guarantees all of the invariants established by the constructors
- Releasing new version of serialized class greatly improves number of test-cases need to be verified
- Classes designed for inheritance should rarely implement `Serializable`
- You should consider providing a parameterless constructor on nonserializable classes designed for inheritance
- Use *thread-safe state machine* pattern (atomic-reference to enum) to implement a serializable superclass
- Inner classes should not implement `Serializable`

### Item 75: Consider using a custom serialized form

- Do not accept the default serialized form without first considering whether it is appropriate
- The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content
- Even if you decide that the default serialized form is appropriate, you often must provide a readObject method to ensure invariants and security
- Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object
- Declare an explicit serial version UID in every serializable class you write

### Item 76: Write readObject methods defensively

- For classes with object reference fields that must remain private, defensively copy each object in such a field. Mutable components of immutable classes fall into this category.
- Check any invariants and throw an `InvalidObjectException` if a check fails. The checks should follow any defensive copying.
- If an entire object graph must be validated after it is deserialized, use the `ObjectInputValidation`interface
- Do not invoke any overridable methods in the class, directly or indirectly.

### Item 77: For instance control, prefer enum types to readResolve

- To satisfy singleton property for serializable object, implement `readResolve`
- If you depend on `readResolve` for instance control, all instance fields with object reference types must be declared `transient`
- `readResolve` is not obsolete. It is needed for writing a serializable instance-controlled class whose instances are not known at compile time
- Use enum types to enforce instance control invariants wherever possible

### Item 78: Consider serialization proxies instead of serialized instances

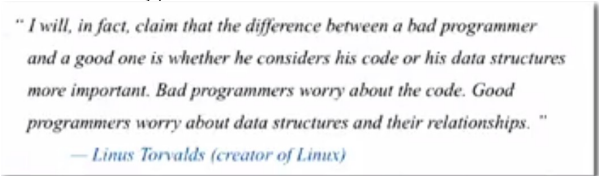
- Serialization proxy is a private static inner class implements Serializable and reflects serializable data for original object
- Add `writeReplace` method to proxy class. Serialization system emits a proxy instance instead of an instance of the enclosing class.
- Add `readObject` method to proxy class. Attacker wouldn't be able to violate class invariants.
- Add `readResolve` method to proxy class that returns logically equialent instance of the enclosing class.
- The serialization proxy pattern has some limitations:

    - It is not compatible with classes that are extendable by their clients
    - It is not compatible with some classes whose object graphs contain circularities
    - Serialization is slower than standard approach

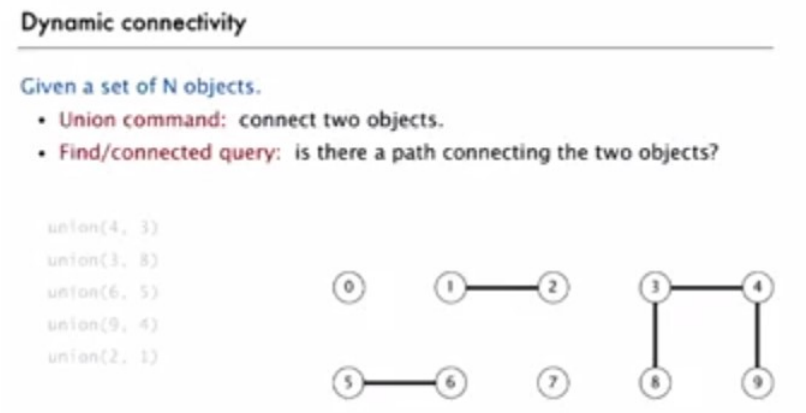
---

# Intro to algorithms

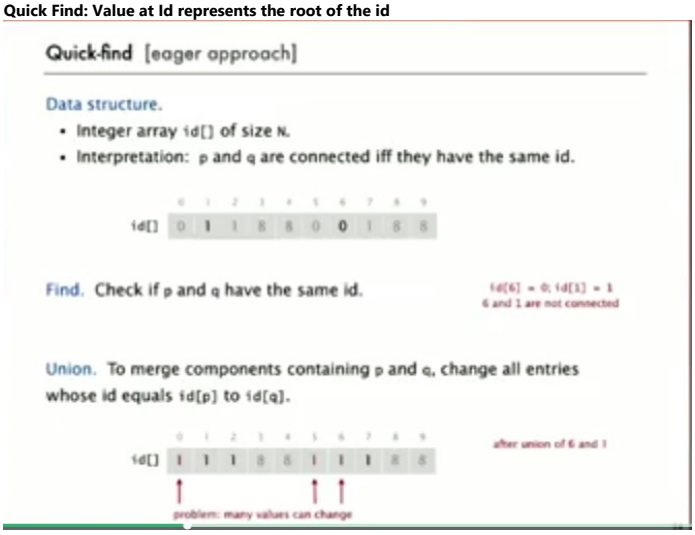
- Algorithms - methods for solving problems
- Data Structure - method to store information
- Algorithms were formalized in 1930 (by Church and Turing)
- Network Connectivity problem - in a network of nodes, find if two nodes are connected



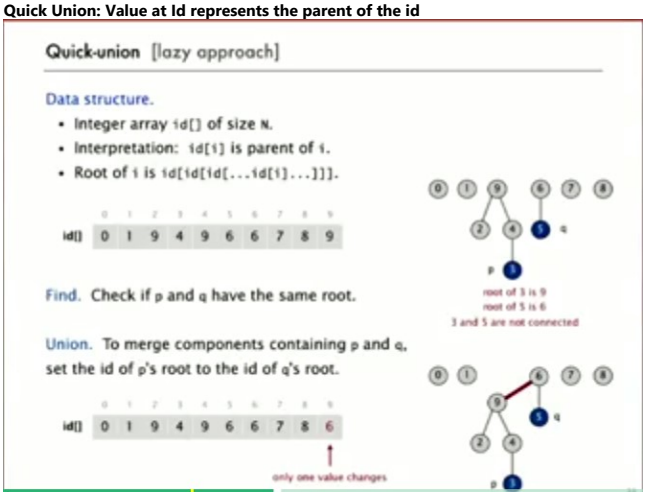

- **Dynamic Connectivity (Union/Find) Problem**




- Connection Properties (equivalence relation)
  - Reflexive: p is connect to p (relation with self)
  - Symmetric: p connected to q implies q connected to p
  - Transitive: p to q and q to r implies p is connected to r
- Connected Components: Maximal set of objects that are mutually connected
  - Any two object in the connected components are connected
  - Any object outside the set isn't connected
- Mostly our attempt will be to get an algorithm better than quadratic complexity as that won't scale
- **Quick Find: Value at Id represents the root of the id**

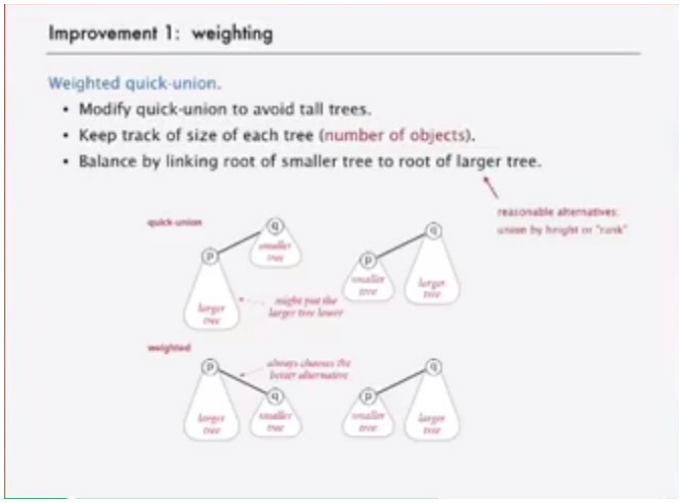



  - Union is expensive, find/union is O(n), find - O(1)
- **Quick Union: Value at Id represents the parent of the id**

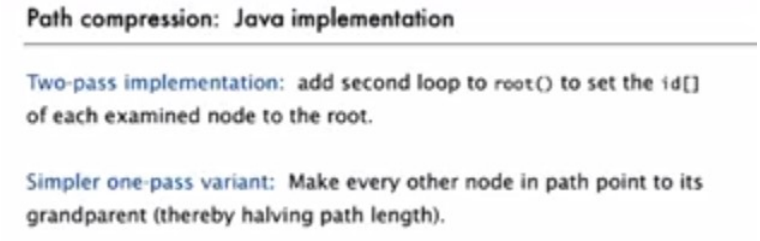



  - Issues: Trees can get tall
  - find can be too expensive

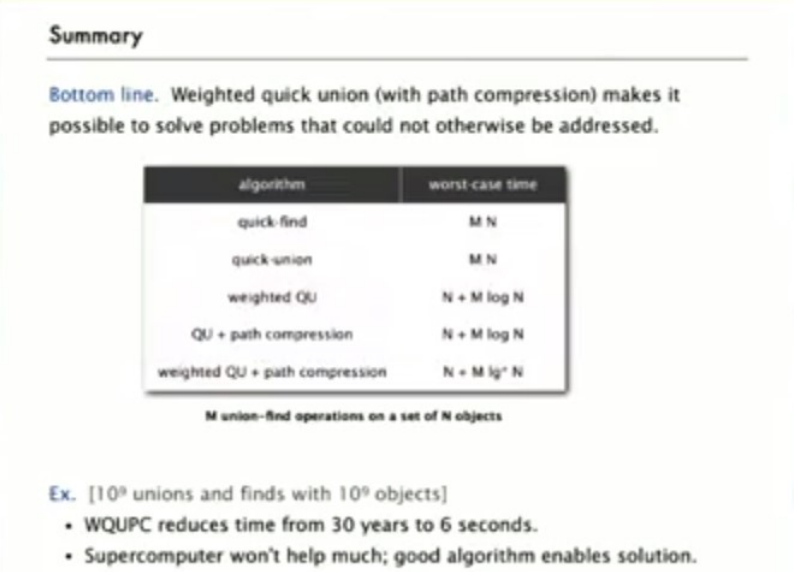
- **Weighted Quick Union: Quick union with ensuring that smaller tree becomes a child of the bigger tree**
  - Depth of any node is at most lgN




**Weighted Quick Union with Path compression:**




- lg* N -> # of times log needs to be done to get to 1 for a number, extremely small number even for large N
- There is no possible implementation that will give linear time for Dynamic connectivity problem. Weighted quick union with path compression is N + M*lg*N [M is the number of union/find operations]
- **Summary:**




- Application:
  - Percolation, electrical circuit, social network
  - Percolation phase transition - there is a small range of probability when percolation can go either way, below the range almost always no percolation, and above - it will always percolate

**Analysis of Algorithms**

- Scientific Method
  - Observer some features of the world
  - Hypothesize a model consistent with the observations
  - Predict events using Hypothesis
  - Verify predictions by making further observations
  - Validate by repeating until hypothesis and observations agree
  - Principles

        - Experiments must be reproducible
        - Hypothesis must be falsifiable

- Always a good idea to measure running time (java has a class Stopwatch)
- Plot running time against input size on a log log scale, slope of the line is what we are interested in (if slope is x, then it is N^x - power law)
- Doubling Hypothesis
    - double input size and then take the ratio T(2n)/T(n), log of that to base 2 will give you the power (N^power)
- if running time is represented by a*N^b. a depends on hardware, b will be decided by the algorithm
- Mathematical Models: Total running time = sum (cost of operation*frequency) across all operations, cost is hardware, freq is given by your algo
- To simplify -
    - instead of analyzing each and every operation, a representative set of operations are picked (most expensive, or most frequent)
    - throw away lower order terms, they are significant only when N is small (~) technical definition f(n) ~ (gn) means lim(n->inf) f(n)/g(n) = 1
        - ~ is an approximation technique, big O provides an upper bound
- Some estimates (discrete mathematics)
    - 1+2+...+N ~ (1/2)N^2 -> integration(x dx) 1->N
    - 1+1/2+1/3+...+1/N ~ ln N ~ int(i/x dx)
- Order of Growth Classifications
    - 1, log N, N log N, N^2, N^3, 2^N

| | | |
|---|---|---|
| 1 | Constant | statement (no loops) |
| log N | Logarithmic | loop but input gets divided in half |
| N | Linear | Loop, touch everything once |
| N log N | Linearithmic | Divide and Conquer |
| N^2, N^3 | Quadratic, Cubic | double loop, triple loop |
| 2^N | exponential | exhaustive search (combinatorial) |

- nlogn is where things remain useful, beyond that it is useless
- Analyze inputs too
    - best case based on data
    - worst case based on data
    - take an average/expected case
    - Real complexity lies somewhere between the two
- Model your input data per your needs (use worst case or randomly pick based on probability)
- Notations
    - Big theta - bounded by this function, so theta(N^2) means it lies between a*N^2 and b*N^2
    - Big O - upper bound ( O(N^2) -> < a*N^2)
    - Big Omega - lower bound
- For finding an optimal algorithm
    - find the lower bound (by analyzing)
    - find the upper bound (by analyzing and starting with brute force algo but improving it as better approaches come along)
    - Optimal algo will be when lower and upper bound merge, if not, the problem has no optimal solution and is considered open problem
- Open Problems - where there is no optimal algorithm (proved to be optimal)
- 1970s - golden age of algorithms

**Memory**

- Typical memory usage (in bytes)

| | |
|---|---|
| boolean | 1 |
| byte | 1 |
| char | 2 |
| int | 4 |
| float | 4 |
| long | 8 |
| double | 8 |

- Memory usage - Objects in Java (in bytes)
    - Object overhead - 16 bytes
    - Reference - 8 bytes
    - Padding - each object uses a multiple of 8 bytes

**Stacks, Queues**

- Collection of items - key requirements - adding, removing, iterating
- Key is to figure out which one to add/remove; Stacks (push and pop) - LIFO, queue - FIFO (enqueue, dequeue)
- Modular programming - separate interface and implementation

- LinkedList - contains the item and a reference to the next item, here is a stack using a linkedList

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

  - Every operation takes constant time
- Array Implementation of Stack

```
public class FixedCapacityStackOfStrings
{
   private String[] s;                          a cheat
   private int N = 0;                           (stay tuned)

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

**Considerations**

- Underflow - popping an empty stack
- Overflow - need to resize the array when the limit is reached
- Loitering - holding a reference to an object when it is no longer needed (set the popped item as null)
- Nulls - allow or disallow?

**Resizing Arrays**

- Fixed size array implementation needs client to provide the size ahead of time
- How to grow/shrink array size
- Repeated doubling
  - Growing the size - Double the size every time the array gets full
  - Shrinking the size - Wait till array is quarter full, then halve it
- LinkedList vs Resizing Arrays
  - worst case is constant time for LL
  - amortized time is constant for resizing array
  - less space required for res array
  - Some operations will be slower in res array though overall its ok

**Queues**

- API - enqueue, dequeue, isEmpty
- can have both linkedList and resizing Array Implementations

**Generics**

- Welcome compile time errors, avoid run-time errors
- Java doesn't allow generic array creation, so to create a generic implementation use Object and then cast to the generic parameter
- A good piece of code should have zero cast

**Iteration**

- Support iteration over stack items by client
- Make stack implement iterable interface
  - Iterable has a method that returns an iterator
  - An iterator is a class with methods hasNext() and next()
  - foreach statement works only if it is iterable

**Bag**

- API - add(), size(), iterator()
  - Stack w/o pop
  - Queue w/o Dequeue

**Applications**

- ArrayList uses resizing array
- java has java.util.Stack
- java has these DS but they are bloated and the performance might be slow
  - linkedList takes linear time to access an index
- Applications
  - Parsing in a compiler
  - JVM
  - Undo in a word processor
  - back butting web browser
  - Implementing function calls in a compiler
    - function call: push local environment and return address
    - Return: pop return address and local environment
  - Arithmetic Expression Evaluation
    - Evaluate infix expressions - Dijkstra's two stack algorithm (*Simply Amazing!!*)

      **Dijkstra's two-stack algorithm**

      Value: push onto the value stack.
      Operator: push onto the operator stack.
      Left parenthesis: ignore.
      Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.
- java.util.Stack - for each returns in FIFO

**Sorting**

- Rearrange an array of N items in ascending order by a key which is present in the record
- Callback - reference to an executable piece of code (java uses interfaces, c- function parameters)
  - A way of passing functions as parameters
- Comparable interface - compareTo method
  - Class implements comparable interface if it needs to be sorted
- Total Order is a binary relation <= that satisfies
  - Antisymmetry if v <= w and w <= v then v=w
  - Transitivity if v <=w and w <=x then v<=x
  - Totality either v<=w or w <=v or both
- Implement compareTo so that v.compareTo(w)
  - is a total order
  - returns -1 (v <w), 0 (v=w) and 1 (v>w)
  - throws an exception if either is null, or types are incompatible
- **Selection Sort**
  - In iteration i, find index min, which is the smallest remaining entry
  - swap a[i] and a[min]
  - Invariants
    - Entries to the left of the pointer are fixed and in ascending order
    - No entry to the right of the pointer is smaller than the entry to the left of the pointer
  - Analysis
    - Quadratic no matter what the input
    - Data movement is minimal (linear)
- **Insertion Sort**
  - In iteration, take the ith element and exchange it with each larger element on the left
  - Invariants - pointer scans from left to right
    - Entries to the left of the pointer are in ascending order
    - Entries to the right of the pointer are not seen yet
  - Analysis
    - For a randomly ordered array, roughly (.25)N^N compares and (.75)N^2 exchanges
    - Best case - ordered array - (N-1) compares, 0 exchanges
    - Worst case - ordered array in wrong direction - (.5)N^2 compares and exchanges
  - Great for partially sorted arrays, runs in linear time
    - An array is partially sorted if the number of inversion <= c*N
      - An Inversion is a pair of key out of order
    - Number of exchanges = # of inversions, and every element gets compared once
- **Shell Sort (non elementary sort algo)**
  - h-sorting an array = h interleaved sorted subsequences
  - h sort array for decreasing sequence of values of h
  - How to h-sort an array
    - Insertion sort with stride length h (going back h elements instead of 1)
  - Why insertion sort for h-sorting
    - When h is large the sub arrays will be small
    - when h is small, the arrays are partially sorted
  - A g-sorted array remains g sorted after h sorting it
  - What values of h to be used
    - research problem
    - power of 2 - 1
    - Knuth - 3x+1
    - Sedgewick - found some seq too
  - Analysis - open, no accurate model yet
    - Worst case number with 3x+1 increments is O(N^1.5)
    - number of compares with 3x+1 increments is at most c*N*(number of h), c is a small number
  - Practically very fast
- **Applications**
- **Shuffling -** produces a uniformly random permutation of the input array
  - Option 1
    - Generate a random real number for each card

        - sort
        - Expensive
    - Knuth Shuffle
        - Scan i from left to right
        - in iteration i, pick r between 0 and i uniformly at random
        - swap a[i] and a[r]
        - works in linear time
- **Convex Hull -** of a set of N points is the smallest polygon enclosing the points
    - Application
        - Robot motion planning to avoid an obstacle
        - Farthest Pair
    - Facts
        - Can be traversed only by taking counter-clock wise turns
        - Vertices appear in an order of polar angle wrt to the lowest point (y coord smallest)
    - Graham Scan
        - Start with point with the smallest y coordinate
        - sort points by polar angle
        - consider point in order, discard unless it creates a ccw turn
    - Implementing a ccw turn
        - signed area > 0, then a->->c is ccw, if <0 then cw, 0 = collinear
        - Signed area is determinant

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

$$(b - a) \times (c - a)$$

**Mergesort**

- Java uses mergesort for object sorting
- Divide array in 2 (Divide and conquer algorithm)
- Recursively sort each half
- Merge two halves
- Analysis
    - uses at most N lg N compares and 6NlogN array accesses for sorting array with size N
    - Following is a recurrence calculation
        - c(N) <= C([N/2])+ C([N/2]) + N
        - Accesses(N) <= A([N/2]) + A([N/2]) + 6N
    - if D(N) = 2* D(N/2) + N for N>1 with D(1) = 0, then D(N) = NlgN
- Uses extra space proportional to N
- A sorting algorithm is in-place if it uses <= cLogN extra memory
- Practical improvements
    - If input size is too small, use insertion sort
    - Stop if it's already sorted (compare the last element of the first half with the first element of second half)
    - Switch array and aux array (saves the copying/moving of items)
- **Bottom up mergesort**
    - Pass through array, merging subarrays of size 1
    - Repeat for subarrays of size 2, 4, 8,....
    - easy to code, no recursion

**Complexity of Sorting**
**Computational Complexity** - Framework to study efficiency of algorithms for solving a particular problem X
**Model of Computation -** Allowable Operations

- Cost Model - Operation Counts
- Upper Bound - Cost guarantee provided by some algorithm for X
- Lower Bound - Proven limit on cost guarantee of all algorithms for X
- Optimal Algorithm - best possible cost guarantee for X (lower bound = upper bound)

The optimal algorithm for sorting is NlgN (there can't be a better algorithm than that). Mergesort is an optimal algorithm for sorting based on number of compares, but uses extra space

However, if we know more about the distribution (instead of it being random), we may do better

**Assertions**

- Statements to test assumptions about your program
    - Helps detect logic bugs
    - Documents code
- Assert in java takes a boolean, throws exception if assertion is false
- Can be disabled in production by using a flag when run
- Should be used to check internal invariants assuming they won't be in production code

**Elementary Sorting Algorithms** - Insertion sort, Selection Sort, Shell Sort

**Classic Sorting Algorithms** - Mergesort and Quicksort (one of top 10 algos of 20th century)
- Critical components in the world's computational infrastructure

**Comparators**

- Comparable allows one way to compare objects of same type
    - It is tied to the type

- Comparator allows for multiple ways to compare this
  - Create an object of type comparator
  - Pass it as a parameter to sorting method etc to use it
- Implementing a comparator
  - Define a nested class that implements comparator interface
  - Implement the compare method

**Stability**

- A stable sort preserves the relative order of items with equal keys after sorting is done
- Insertion sort and mergesort are stable, selection sort and shell sort are not
- Mostly determined by what happens to items that are equal and some long distance exchanges that can make items jump over some equal items in the middle

**Quicksort**

- one of top 10 algorithms of 20th century in science and engineering
- Doesn't take any extra space
- Shuffling is needed for performance guarantee
- Equal keys - stop on keys equal to the partitioning element
- quicksort is much faster than mergesort
- Best case - compares is NlgN
- Worst case - 1/2N^N - if the array becomes sorted after shuffling
- Average Case - 1.39NlgN
- Worst case is less likely than lightening striking you are this moment
  - Random shuffle guarantees against worst case
- Some implementations end up running in quadratic time, also if too many duplicates then also quadratic
- Is not stable as partitioning does long range exchanges
- With extra space it can be made stable
- Practically fastest sorting algorithm
  - Implement insertion sort for smaller arrays
  - Or keep smaller arrays unsorted, and then do a pass to sort the overall array

**Selection Problem (Application) - Find kth largest**

- find top k etc
- Lower Bound - N, upper Bound - NlgN
- Modified quick sort gives the solution - Quick Select
- On an average it takes linear time
- Compare based selection algorithm - whose worst case running time is linear was found in 1973

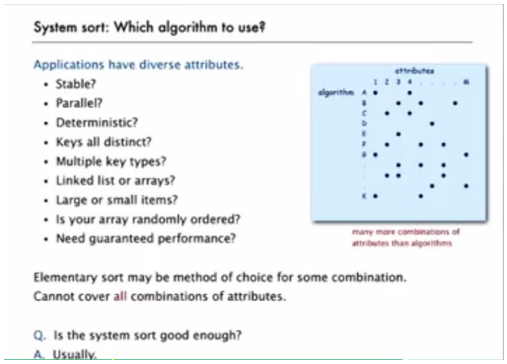
**Sorting when we have large number of duplicates**

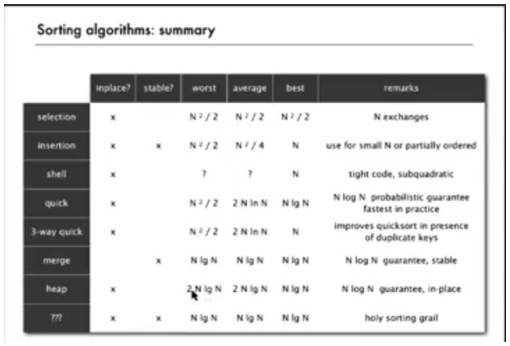
- Merge is always between 1/2 N log N and N log N
- Qsort
  - 1990s C user found defect in qsort (if partition doesn't stop on equal keys, the algo goes quadratic) - Bell Labs
  - Dijkstra 3-way partitioning is the best way where equal keys are all moved (for such cases)
    - Need to spend more time on this
    - Dutch National Flag problem
  - it is entropy optimal
  - In many practical cases, 3-way sorting is linear

Sorting Applications (indirect)

- find median
- duplicates
- compression
- in computer graphics
- Load balancing

Arrays.sort() --> uses tuned quicksort for primitives, tuned mergesort for objects
for finding the partitioning key - Tukey's ninther - median of the median of 3 samples each of 3 entries. The also avoids relying on system - depends on random shuffle that is dependent on the random number generator in the system



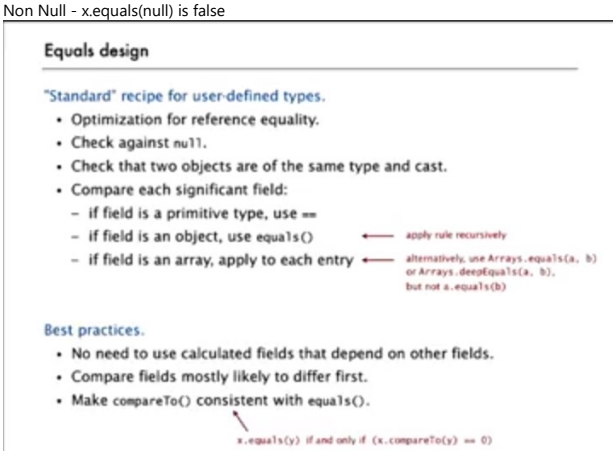

**Priority Queue**

- Remove largest or smallest item
- APIs -> insert(key), delMax, isEmpty(), max(), size()
- Applications
  - Numerical computation
  - Compression
  - Event driven simulation
  - AI
  - Spam Filtering and many more
- Sample application -> you want top M out of N, but don't have enough memory to store N items
- Binary Heaps
  - Complete Binary tree is completely balanced except for possibly the bottom
  - Height of complete binary tree is LogN
  - Heap ordered binary tree - Parent's keys no smaller than children keys
    - Array implementation, no links needed, arithmetic on indices will give us parents/children
    - swim operation to move nodes in the right place if there is an anomaly
      - child is bigger than a parent
    - insert becomes, add at the end and make it swim
    - sink is the opposite of swim
      - parent is smaller than children
    - Keys should be immutable
  - 
- Immutability
  - Can't change values once created, keyword final
  - Safe to use as key in priority queue or symbol tables
    - If they are mutable, heap order will change w/o you knowing
- Classes should be immutable unless there is a very good reason to make it mutable, and you should limit the mutability as much as you can
- Heapsort
  - Create max heap with all N keys
  - Repeatedly remove the maximum key
  - Construction uses <= 2N exchanges and sort uses <=2NLgN compares exchanges
  - Significance/Pros
    - in-place sorting algo, no extra space (mergesort needs extra space)
    - no quadratic worst case (qsort has that)
  - Cons
    - Inner loop is longer than qsort
    - Makes poor use of cache memory (as there are longer distance exchanges, not optimal for computers)
    - Not stable

**Symbol Tables**

- Key Value Pair abstraction
- Insert a value with specified key
- Given a key, search the corresponding value
- Applications
  - Dictionary
  - book index
  - file search

    - financial account
    - web search
    - DNS, reverse DNS
    - compiler
    - genomics
- API
    - Associative Array Abstraction
    - put, get, delete, contains, size, isEmpty, iterate over keys
    - Conventions (make the implementation much easier)
        - Values are not null
        - get returns null if key is not present
        - put overrides the old value with new
    - Natural assumptions about keys
        - Keys are Comparable (implement compareTo) - Keys should come from a total order
        - Key are any generic types
        - Use hashcode to scramble keys
    - Best practice also includes using immutable datatype
    - More about assumptions
        - Equality (equals in Java) is an equivalence relation
            - Reflexive: x.equals(x) is true
            - Symmetric: x.equals(y) iff y.equals(x)
            - Transitive: x.equals(y) is true and y.equals(z) is true implies x.equals(z) is true
        - Non Null - x.equals(null) is false




    - Binary Search Implementation
        - provides constant or logN performance for most of the operations except insert and delete which is linear
    - Binary Search Tree
        - A BST is binary tree in symmetric order
        - A binary tree is either
            - empty OR
            - two disjointed binary trees (left and right)
        - Symmetric Order: Each node has a key and every node's key is
            - Larger than all the keys in its left subtree
            - Smaller than all the keys in its right subtree
    - BST Representation in Java
        - A BST is a reference to a root node (only one instance variable)
        - A node has a key, value and a reference to left and right node (subtree)
    - A BST can have a worst case based on the order of insertion of keys (for keys coming in ascending order, it is linear and acts as a linked list)
    - A BST is similar to partitioning in Qsort
- If N distinct keys in a BST come in random order, expected number of compares in search/insert is ~2LgN, expected height of the tree is 4.3LgN

**Geometric Applications of BSTs**

1d Range Search (example B queries)

- Range Search - find all keys between k1 and k2
- Range count - number of keys between k1 and k2

Orthogonal line intersection search: Sweep line algorithms

- find all intersecting lines from a group of lines (only horizontal or vertical)
- Sweep a vertical line across the plane, when it hits the left coordinate of a horizontal line put it in a BST, when we hit the right, take it out. If we hit a vertical line, do a range search on the end points to find the y coordinates of the existing stuff on BST, those are intersecting lines

**Kd Trees**
2d Range Search

- Range Search - find all keys within a 2d range
- Range count - number of keys within a 2d range

Grid Implementation

- divide space in mxm grid
- assign a square to each point
- Doesn't work well when there is clustering - very common in geometrical data

Space Partitioning Trees
<geometric applications very common>

**2d tree**
Recursively partition a plane in two half planes
- Use a vertical line first
- For next use a horizontal line
- keep alternating

Range search can make use of the property and compare only a few points to see if they lie in the target rectangle

Nearest Neighbor - Find closest point to a given point

Flocking Boids - 3 simple rules lead to complex emergent flocking behavior
- Collision Avoidance - point away from k nearest boids
- Flock Centering - points towards the center of mass of k nearest boids
- Velocity Matching - update velocity to the average of k nearest boids

kd Tree - k dimensions

Application - N Body Simulation

Interval Search Trees
Create BST, each node stores an interval
Use left point as key
also, store the max end point in a subtree at the each node of the tree

Orthogonal Rectangle Intersection Problem - All rectangles axis aligned, find all intersecting ones
Microprocessors make use of this - design of computer became a geometric problem as machines started assembling it. Lot of rule around what can or cannot intersect and how close certain things can be.

Applications
1d -> range search, interval search
2d -> line segment intersection
2d -> orthogonal rectangle intersection

**Hashing**
Red black BST gives us all the operation in logN, can we do better
Convert a symbol table to an array using hash function
Hash function takes a key and converts to an integer, use that index to store the value
Issues
- Computing hash functions
- Equality Test - use equals
- Collision Resolution - if two keys hash to same array index

Implementation is space/time trade off
- if no space limitation - store every key as an index
- if no time limitation - hash everything to one key and then do sequential search

Computing the hash function
- Efficiently computable
- Each table index should be equally likely

Every object in Java implements hashcode that returns a 32 bit int
- Requirement if (object a).equals(b) then hashcode(a) has to be equa to hashcode(b)
- highly desirable if a is not equal to b then it's better if their hashcodes differ
- Default implementation is using address of the object

Integer - hashcode returns itself
String - Horner's Method (adds a string has a base 32 number)

General Recipe
- 31x + y rule to combine all the significant fields
- for primitive use hashcode of wrapper type
- for reference types use their hashcodes
- for null put 0
- for arrays apply this to every entry

Universal Hash Functions

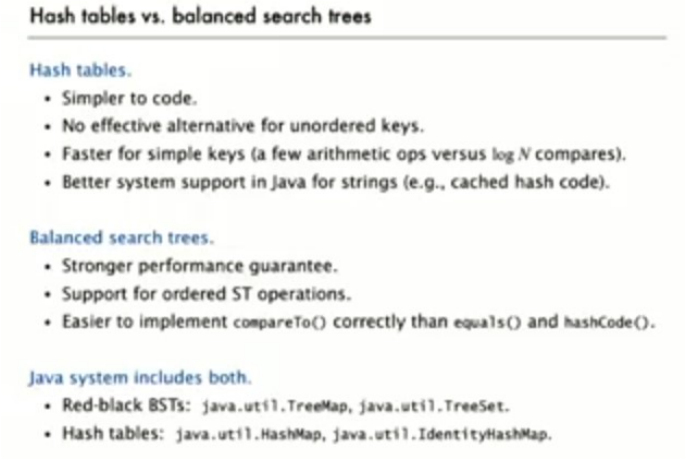
Modular Hashing
- hash code is an integer between -2^31 and 2^31-1 (32 bit int)
- Hash Function -> to store the value in an array of M, use mod M, If M is prime, we will be able to distribute stuff to all the bits and each position will be mostly utilized

Uniform Hashing Functions
- Bins and Balls
- Birthday Problem - how many elements will it take to get to the first duplicate hash sqrt(PI*M/2)
- Coupon Collector - every bin has at least one ball - MLnM
- Load Balancing - After M tosses, most loaded bin will have (log M/log log M) balls

Collision Resolution
- Separate chaining
  - Create a linked list for each hash location
- Linear Probing - Array with significantly bigger size than the number of keys expected
  - Put the key at the hash location if not occupied, if it is occupied, check the next until you find a slot

Cost of computing a hash has to be managable for hashtables to be of any use
Denial of Service attack -> if attackers know your hash function, they can send all the data that will hash to a particular value

One way hash function - hard to find a key that will hash to a particular value or two keys that hash to the same value




**Symbol Table Applications**
- Set: Collection of distinct keys
  - Exception Filters
- Dictionary
- File Indexing (search for a string in a list of files) - Build an index with strings as keys and set of all files containing it as value
- Matrix Multiplications - typical implementation is N^2, for Sparse we can do better. We can store the non zero values as key-values which represent index and the value
  - Sparse vector representation and using it we can have sparse matrix representation. dot product will become linear

- Symbol tables can't do ordered iterations or any operations that need keys to be in order

Concordance query - find all occurrences of a word with context (stuff around it)

---

https://medium.com/omarelgabrys-blog/diving-into-data-structures-6bc71b2e8f92

---

# Sprng in Action (Book, 4th Edition)

**General**

- At the heart of Spring is Dependency Injection and Aspect Oriented Programming
- Blots on Java - applets, EJBs
- Created as a lighter alternatives to heavy J2EE technologies particularly EJBs
- J2EE eventually caught up to a lot of things that Spring had
- It's open source, created by Rod Johnson, to address complexities of enterprise application development (Fundamental mission - Simplify Java Development)
- Enabled POJOs to achieve things that only EJBs could
- Four key strategies
  - Lightweight and minimally invasive development with POJOs
  - Loose coupling through DI and interface orientation
  - Declarative programming through aspects and common conventions
  - Eliminating boilerplate code with aspects and templates
- Objects collaborate with each other, the objects that an object collaborates with are called dependencies

**Dependency Injection**

- Dependency Injection
  - With DI, objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren't expected to create or obtain their dependencies
  - Objects should know their dependencies only via their interface, the dependencies can be swapped out with a different implementation of the interface
- The act of creating associations between application components is commonly referred to as *wiring*
- The Spring application context is fully responsible for the creation of and wiring of the objects that make up the application
  - Several implementations, primarily based on how the configuration is loaded
    - Example: For XML -> `ClassPathXmlApplicationContext, looks for xmls in classpath in its main method`
    - `Java Based -> AnnotationConfigApplicationContext`

**Aspect Oriented programming (AOP)**

- Enables you to capture functionality that's used throughout your application in reusable components - enabling separation of concerns
- System services such as logging, transaction management, and security often find their way into components whose core responsibilities is something else. These system services are commonly referred to as *cross-cutting concerns* because they tend to cut across multiple components in a system.
- AOP Declaration
  - Define bean in your configuraton
  - define point cuts - places where the aspect applies - defined using a point-cut expression language to identify classes/methods with specific names etc

**Templates**

- Spring seeks to eliminate boilerplate code by encapsulating it in templates
  - JDBC, Rest, JNDI, JMS all are examples of places where you need a lot of boilerplate code

**Spring Container**

- In a Spring-based application, your application objects live in the Spring *container*
- Two implementations - Bean Factories and ApplicationContexts, ApplicationContext build upon Bean Factory
- Different flavors of ApplicationContext
  - `AnnotationConfigApplicationContext`—Loads a Spring application context from one or more Java-based configuration classes
  - `AnnotationConfigWebApplicationContext`—Loads a Spring web application context from one or more Java-based configuration classes
  - `ClassPathXmlApplicationContext`—Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
  - `FileSystemXmlApplicationContext`—Loads a context definition from one or more XML files in the filesystem
  - `XmlWebApplicationContext`—Loads context definitions from one or more XML files contained in a web application
- Spring Bean's Lifecycle - Each step is declarable and helps you customize how you manage your bean

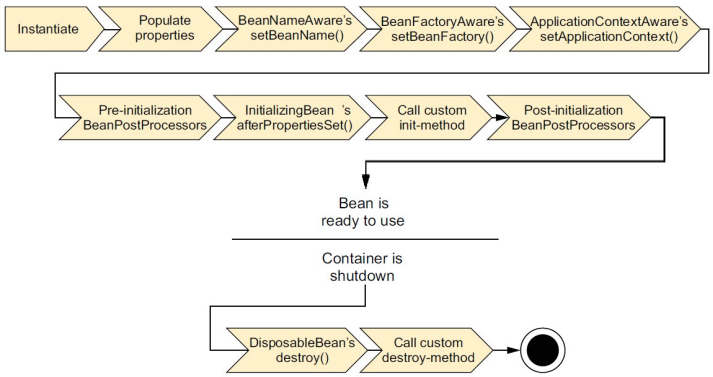



**Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.**

**Spring Core/Modules**

- As of Spring 4, it has 20 modules with spring distribution having 3 jars each (bin, source, javadoc)

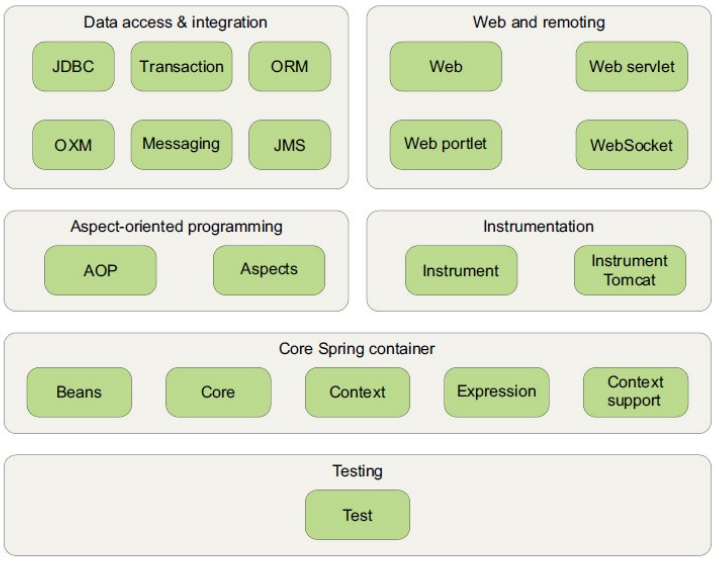



Figure 1.7 **The Spring Framework is made up of six well-defined module categories.**

- 
- Spring Modules
  - Core Spring Container - Bean management via Bean factory, appContext and jndi access, ejb support, email
  - AOP - Aspects
  - Data Access and Integration - JDBCTemplate, hooks to ORMs, JMS
  - Web and Remoting - Spring MVC, RMI, create/consume Restful APIs
  - Instruments - adds agents to JVM
  - Testing
- Spring Boot takes an opinionated view of developing with Spring to simplify Spring itself

---

# Introduction to MongoDB

- Mongo - fast, scalable, tunable consistency, no tables, no schema
- Impedance Mismatch: Gap between how object oriented programming treats objects vs how they are represented in Relational DBs
- Most people are object oriented developers, tables and rows are not objects
- There is no schema in Mongo
- There are no relations
- RDBMS (Scalability issues)
  - Locking - Rigid consistency model
  - Replication and sharding makes it worse
  - Possible fixes
    - De-normalizing helps, but we lose redundancy benefits
    - Relax consistency
- MongoDB (Scalability ideas)
  - No schema: Schema enforcement responsibility goes to the application
  - Single Document write scope
    - A document lives in a collection
    - A lock is applicable only to one document
  - Eventual consistency
  - Capped Collection: Fixed size collection
- Durability: Whether data was persisted to durable media before control is returned
- Consistency: Choice of your use case
  - Fire and forget, send the update and go back. Worst case - primary fails before record gets written, lost forever
  - Majority: Makes the update durable to primary and a fix number of secondaries
  - Consistent and Durable - wait the data to get durable in all secondaries
- **Summary of Mongo features - fast, scalable, tunable consistency, no schema, no tables**
- Installed components
  - mongod.exe - deamon (server)
  - mongo.exe- shell
  - Lots of command line options to start the server, best to have a custom path for data and logs
- Run mongo all the time - install mongo as a service in windows
  - mongod <commandline arguments> --install
  - net start mongod
- Useful mongo shell commands
  - show dbs
  - db
  - use dbname
- Replica set: Primary, Secondary(ies) and one Arbiter DB
  - Writes happen only on primary db
  - Secondaries allow only read access
  - Secondaries will get data from Primary (eventually)
  - Primary fails - one of the secondary will become primary

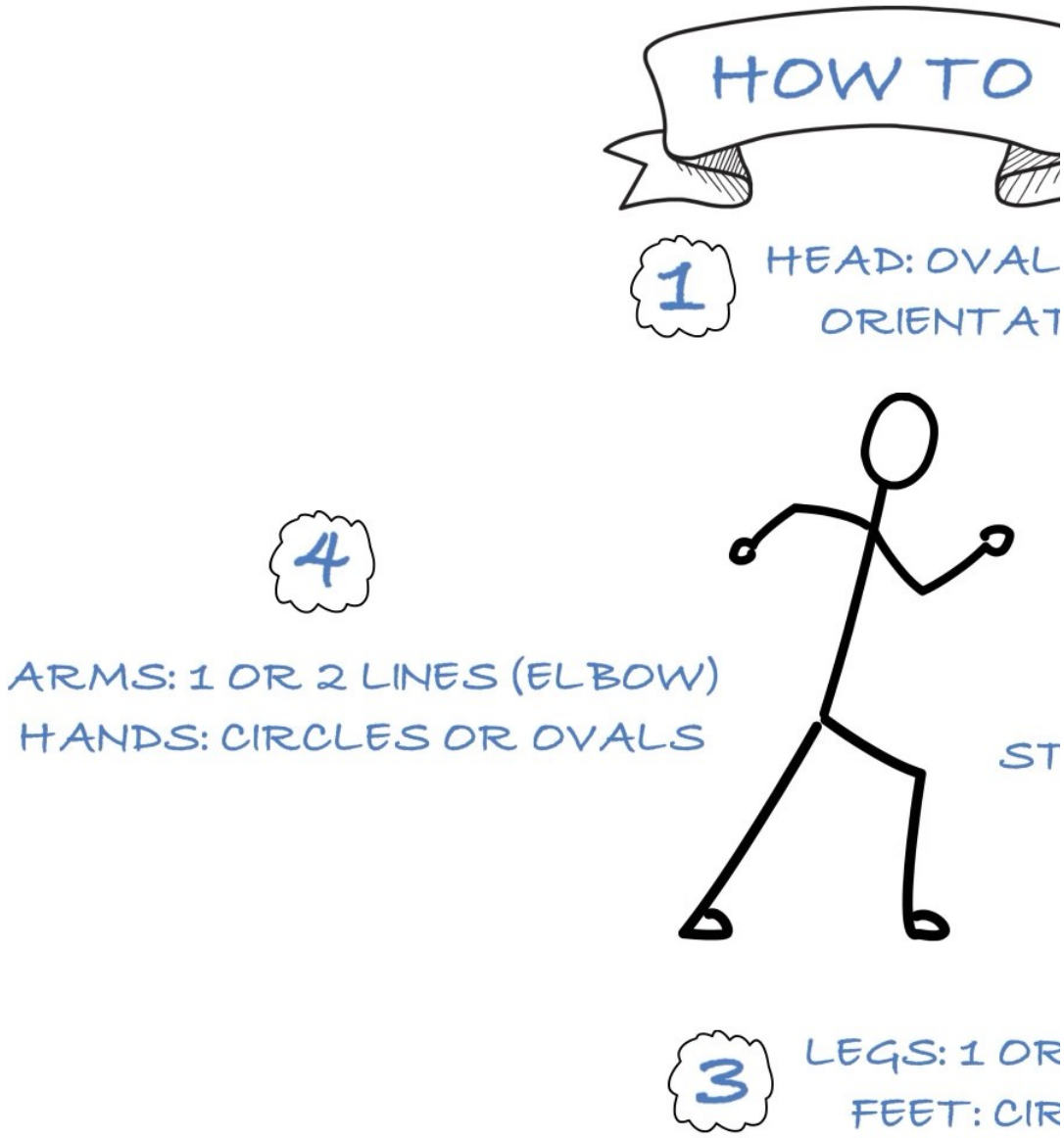
        - Selection Process:
            - Election, a machine needs simple majority (>50% votes)
            - if you have only one primary and a secondary, and primary goes down, you get only 50% votes and selection will not happen
            - Arbiter DB exists for the sole purpose of breaking a tie in election process - does not store any data
            - Arbiter always votes for someone else
    - Minimal Replica set possible will have 3 DBs - primary, Secondary, Arbiter
- Commands for reference to start a replica set
    - start "a" mongod --dbpath ./db1 --port 30000 --replSet "demo"
      start "b" mongod --dbpath ./db2 --port 40000 --replSet "demo"
      start "c" mongod --dbpath ./db3 --port 50000 --replSet "demo"
    - rs.initialize(configName) brings the shell for the replicaSet
    - db.foo.save(json string)
    - rs.help()

**Mongo Shell**

- Shell is a JavaScript interpretor
- Blind Shell - run through a command line and not open the interactive shell
    - mongo server1 --eval "mongo command"
- You can create js scripts too to run as batch on mongo shell
- mongo server1 scriptName.js --shell -> will remain in shell
- printjson(takes a command, returns result as json)
- Editing in a shell
    - ^A - start of line
    - ^k - end of line
    - ^L - clear screen
    - ^back arrow (<-)
- multi-line continuation is allowed
- You can set an editor to open your editor (set EDITOR="editor path.exe"
    - edit myFunc
- You can run some commands at the start of every shell session
    - the file where you can enter it in ".mongorc.js"
    - mongo --norc will disable mongor
-

---

# Drwaing stick figures




---

## Spring Fundamentals (Pluralsight)

- Why Spring?

1. improves maintainability, testability, scalability
2. Puts focus back on business (reduces development complexity)

- Everything in Spring is POJO
- Spring is a glorified Hashmap
- WORA - write once run anywhere
- Spring helps remove configuration from code
  - configuration in code makes it brittle - needs to be rebuilt for deploy to a different environment
  - Testability suffers with config in code
- Wherever we have code calling concrete implementations, spring comes in
- spring-context is the artifact that provides core functionality
- XML was the first way to provide configuration
- Configuration
  - applicationContext.xml holds the configuration - name doesn't matter, but this file is the starting point (default is this name)
  - Spring Context is a hashmap of objects and they are stored in the xml file
  - Can simply be a registry

    - Namespaces  aid in configuration/validation - auto suggest starts popping up, they are defined at the start of xml (w/ xmlns:.....)
- Beans are pojos defined in applicationContext.xml
    - Replace new keyword, wherever you use new, think of beans
    - Define class, but use the interface
- Separation of concerns:
    - we separate business logic and configuration,
    - just need to use a different configuration file for a different environment, w/o the need to recompile your code
- Setter Injection
    - name based
- Constructor Injection
    - Guaranteed contract
    - All possible constructors need to be defined
    - index based
- Autowire: Auto wire beans together (dynamically discovered instead of explicit declaration)
    - ByType - if only one bean of the type exists in the container
    - ByName - name fixes the problem of byType if more than one objects exist for a type
    - Constructor - looks for an object of that type to pass to the constructor
    - no - cannot be auto-wired
- Autowire may appear to hamper performance but it doesn't
- ByName expects things to be named (and linked by names) in specific way
- Component Scanner - Annotation Scanner
    - Needs a few configurations in applicationContext.xml
        - Context Specification
    - Goes through the whole project and wires things together
- Annotations
    - 3 main core spring annotations to find beans /components inside our applications - known as Stereotype annotations (all semantically same)
        - @Component, @Service, @Repository
        - Bean - @component, business logic layer - @Service, Data Access Layer - @Repository
- Autowired - implies it is defined to be instantiated with spring fw
    - can be done at method level, setter level, constructor level
- JSR-330 " Dependency Injection of Java
- We can do all the configuration for spring in Java
- ApplicatioinContext is replaced by ApplicationConfig - place to bootstrap our application - @Configuration at class level
- Beans are defined by @Bean
- Setter injection with Java
- Using Java config for autowired
    - Put this @ComponentScan({"package"})
- Autowiring can completely eliminate stuff from applicationContext or ApplicationConfig class
- Bean Scopes (5 Scopes) - @Scope("appropriate scope")
    - Singleton (default) - one instance per spring container
    - Prototype - every time you request an object, you get a new one (opposite of singleton)
    - Web Only scopes
        - Request
        - Session
        - Global
- When there is a conflict in the way you can get a bean a message shows up and you can figure out what has gone wrong!
- web scopes ignored for now - should get covered when i do spring mvc
- Properties Files - used to abstract out values that can change with each environment (passwords, urls, connections)
    - Declare a property placeholder in applicationiContext.xml
    - in Java way - @Value("${propertyName}")
- When using Java config add @PropertySource to your config class and write a boilerplate method to get the object (PeropertySourcesPlacholderConfigurer)

# Enumerations in Java

- Exact listing of all the elements of a set
- Why?
  - good way of naming things
  - Type sage
  - Limits input
  - Groups things in a set
  - Iterable
- When to use
  - As much as possible (if you can do it, do it)
  - Constraining input is beneficial
  - In place of string constants
  - In place of integers used as types
-

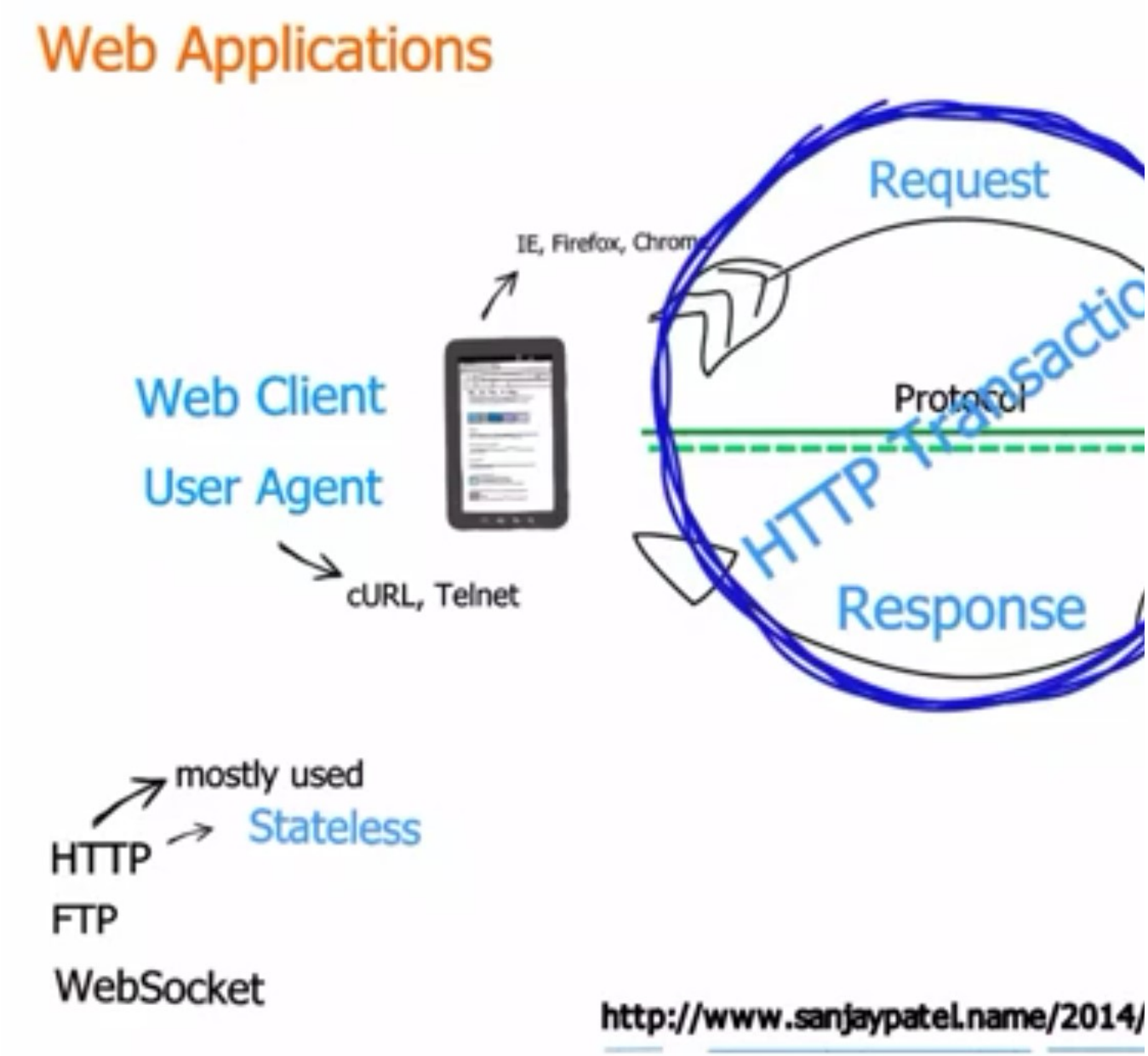
---

# Annotations

- Annotation is metadata (for code)
  - Will not directly get executed
    - example - a little "sign here" note on a contract
- It's a data holding class that describes your code
- Applies to
  - Classes
  - fields
  - methods
- it's put before the line/code that it applies to
- Uses
  - Compilers use it to compile the code correctly
    - @SuppressWarnings
    - @Depracated, @Override
  - Tools
    - Javadocs
    - Testing Frameworks
    - Generate code
  - Runtime
    - Serialization
    - IoC
    - ORM
- public **@**interface <Annotation Name>
- Reflection
  - Code that can examine itself and even change during runtime
  - Meta-programming (programming outside the normal layer of programming)
  - Code treating code like data
- Reflection is not efficient

---

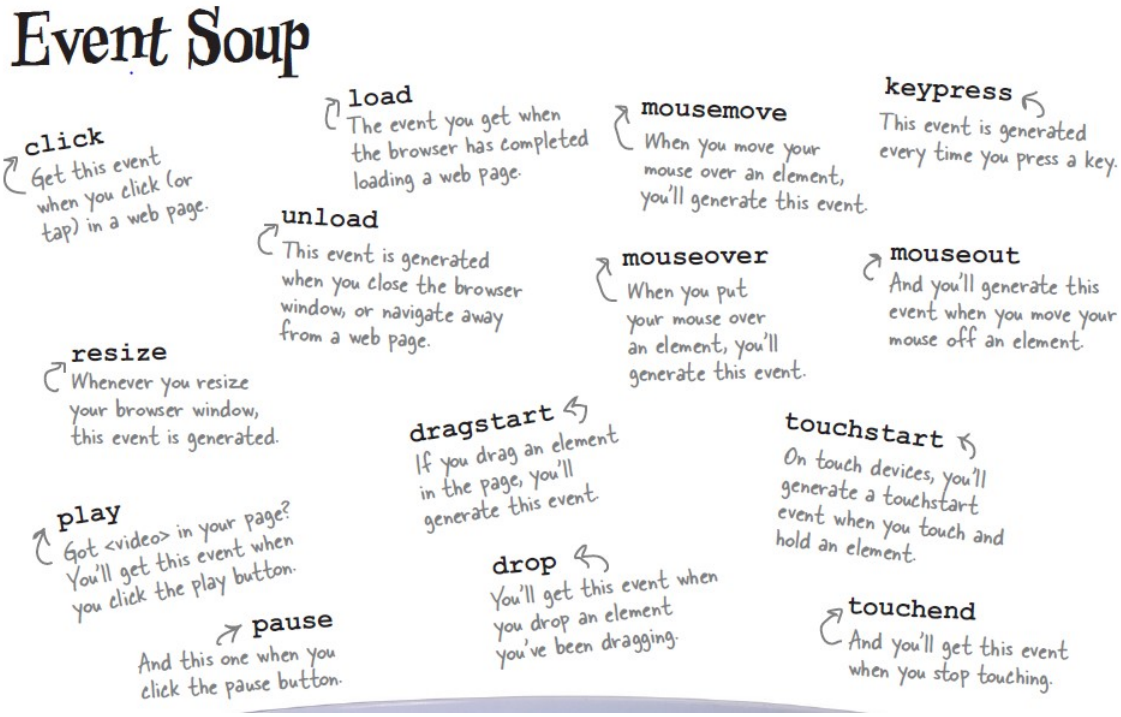
# Web Applications basics diagram




## Javascript Programming (Head First Notes)

- HTML - content, css -presentation, javascript - behavior
- When a browser parses a web page, it creates an object model of your page
- EcmaScript defines the standards for Javascript implementations
- Variable Naming
  - Start with letter, _ or $
  - After that use letters, digits, _, $
  - avoid reserved keywords
- JS is case sensitive
- Variable naming convention followed is camelCase
- JS output
  - Alerts (browser)
  - Console logs (console.log)
  - document.write -> writes arbitrary html to the page
  - manipulate the document (DOM)
- JS code can go in head/body as file/inline
- JS is pass by value
- Global variable life - until page is there or not reloaded
- A variable used without declaration (var) will always be global no matter where it happens
- Hoisting: JS declares all variables needed at the start of the execution, even the ones that are not declared. but hoisting doesn't initialize them

- Code hygiene
  - Groups variable declarations at the top of the scope they are in
  - Group functions together to locate them easily
- Objects: var obj = {key1: val1, key2:val2,...};
- variables representing objects hold reference to the objects (exactly like Java)
- "for in" iterates through all the properties of an object
- Accessing a property obj.prop or obj[prop]. latter allows for expressions to be written there
- DOM
  - when browser loads a page, along with displaying it, it creates an object model of the page stored in DOM (~tree data-structure)
  - JS on the page has access to DOM and can manipulate it at run-time, the resulting changes are reflected in browser immediately
  - any value that is not defined is set as undefined, when the value is expected to be an object a null is used (undefined for objects)
  - for numerical calculations where computation is impossible, NAN is returned. Also NAN != NAN !. You have to use isNAN for that.
- Automatic Datatype conversions with operands of different types
  - Comparison:String converted to number; boolean converted to number;undefined is equal to null. ""->0
  - +: number converted to string
  - Other arithmetic operators: all try converting operands to numbers
  - in general this is a tricky area so watch out as there are more cases than you can think
- === strict equality. same type, same value. The opposite operator is !==
- Typecast string to number: Number(string)
- Considered false: 0, undefined, null, "", NAN
- A string can become a string object as needed and you can call many useful methods on them
- Events
  - After loading browser keeps informing /reporting things - called events
  - Event handlers - code to respond to events (other names - callback and listeners)
  - When an event is generated it is associated with an object and holds information about it which can be passed on the handler
  - Some event types - dom events, network events, timer events
  - Browser has an event queue and events get handled one at a time, so if a handler for a particular even is running slow, it blocks the page
  - timer event - setTimeout(time in milliseconds, handler), setinterval(time in milliseconds, func) - keeps repeating after the interval defined




- Functions
  -

# Cloudera CDH (Pluralsight)

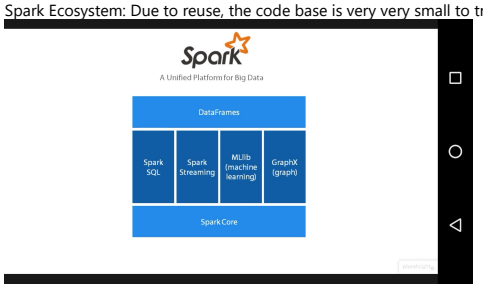
- 3Vs: Volume, Velocity , Variability/Varierty, (extra V - Value)
- unstructured data, heterogeneous sources, schema less, horizontal scaling
- Scaling up is no longer an option, so you get many machines and make them act like one big machine, it's called cluster
- Google invented the big data model, Yahoo created Hadoop
- Hadoop
  - part of apache software foundation
  - new type of data platform
  - Cost effective
  - Massively parallel processing
- Setting up hadoop can be complicated, cloudera steps in there
- Cloudera
  - Core is open source

    - CDH - Cloudera's distribution including Hadoop
    - widely deployed and tested
- Cloudera Navigator = big data + governance
- Virtual Machines
    - Docker
    - Kvm
    - vmware
    - virtual box (Oracle) - It's multiplatform, good performance, open source, totally free
- Cloudera Director - part of paid (enterprise) version - good for production deployments
- Cloudera Manager - software on cluster -- works only on linux, jdk 7&8
- IO Bound/CPU Bound (Type of workload)
- IO Bound
    - Reading a lot from disk/network
    - sorting
    - indexing
    - grouping
    - data import/export
    - data movement/transformation
- CPU Bound
    - Complex operations on input data
    - classification
    - clustering
    - ML etc
- Cluster - connected machines that work together as a single system

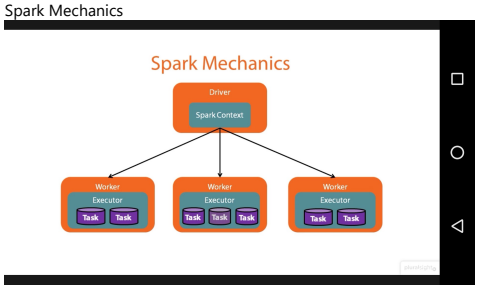
---

# Apache Spark Fundamentals (Pluralsight)

- Benefits
    - Readability
    - Expressiveness
    - Fast
    - Testability - distribution worry is only deploy time
    - Interactive
    - Fault tolerant
    - Unifies big data
- Hadoop ecosystem exploded with tools for different needs
- Spark Staack
    - Core
        - Spark SQL
        - Spark Streaming
        - MLlib
        - GraphX
- Spark Ecosystem: Due to reuse, the code base is very very small to traditional hadoop ecosystem




- Google MapReduce Paper(2004), Yahoo created Hadoop
- Spark creators founded Databricks (2013)
- Backward compatibility is also taken care of all Spark versions (any release not marked alpha takes care of that). 1 release every 3 months for consistency.
- SparkContext and SqlContext - spark-shell gives these two by default
- Spark Mechanics




- Spark context is the starting point for any spark program. You can instantiate it with configuration
- There are two sets of Hadoop APIs (old and new)
- Lambda (Anonymous functions) - functional code block w/o a name, can be passed as an argument to another function that accepts code blocks/functions as inputs
- Spark RDD has two types of functions
    - Transformation: returns another RDD, evaluated lazily (a nested graph is built but not evaluated)
    - Actions: force calculations of the transformation graph. doesn't return an RDD usually
        - Associative property: no matter how you divide a computation, result is same
        - collect - returns an array to the driver, should be called when you are sure driver alone can handle the volume

    - take - takes a few elements (based on the argument)
    - takeSample, takeOrdered etc.
    - max, min, countApproxDistinct
    - Commutative: order of operands doesn't matter (a+b = b + a)
- MapPartitions - can run a function applied on all items at once, very helpful when calling function on each item separately will be very expensive, may be function has a big fixed start up cost
- RDD Combiners:
    - union
    - intersection (only distinct values returned from the intersection)
    - subtractRDD
    - cartesian (double for loop)
    - zip
-

---

# Behavioral Interview Questions

---

# Catalog of Patterns of Enterprise Application Architecture

Catalog of Patterns of Enterprise Application Architecture

Last Significant Update: January 2003

*A short summary of the patterns in Patterns of Enterprise Application Architecture (P of EAA).*

[ Japanese | Russian ]

These pages are a brief overview of each of the patterns in P of EAA. They aren't intended to stand alone, but merely as a quick aide-memoire for those familiar with them, and a handy link if you want to refer to one online. In the future I may add some post-publication comments into the material here, but we'll see how that works out.

Many of these diagrams demonstrate the rather poor GIF output of Visio. The nice diagrams were redrawn for me by David Heinemeier Hansson

**Domain Logic Patterns:** Transaction Script (110), Domain Model ( 116), Table Module (125), Service Layer (133).

**Data Source Architectural Patterns:** Table Data Gateway (144), Row Data Gateway (152), Active Record (160 ), Data Mapper (165).

**Object-Relational Behavioral Patterns:** Unit of Work (184), Identity Map (195 ), Lazy Load (200)

**Object-Relational Structural Patterns:** Identity Field (216), Foreign Key Mapping (236 ), Association Table Mapping (248 ), Dependent Mapping (262), Embedded Value (268), Serialized LOB (272), Single Table Inheritance (278), Class Table Inheritance (285 ), Concrete Table Inheritance (293 ), Inheritance Mappers (302 ).

**Object-Relational Metadata Mapping Patterns:** Metadata Mapping (306), Query Object (316), Repository (322).

**Web Presentation Patterns:** Model View Controller (330 ), Page Controller (333 ), Front Controller (344 ), Template View (350 ), Transform View (361 ), Two-Step View (365), Application Controller (379).

**Distribution Patterns:** Remote Facade (388 ), Data Transfer Object (401 )

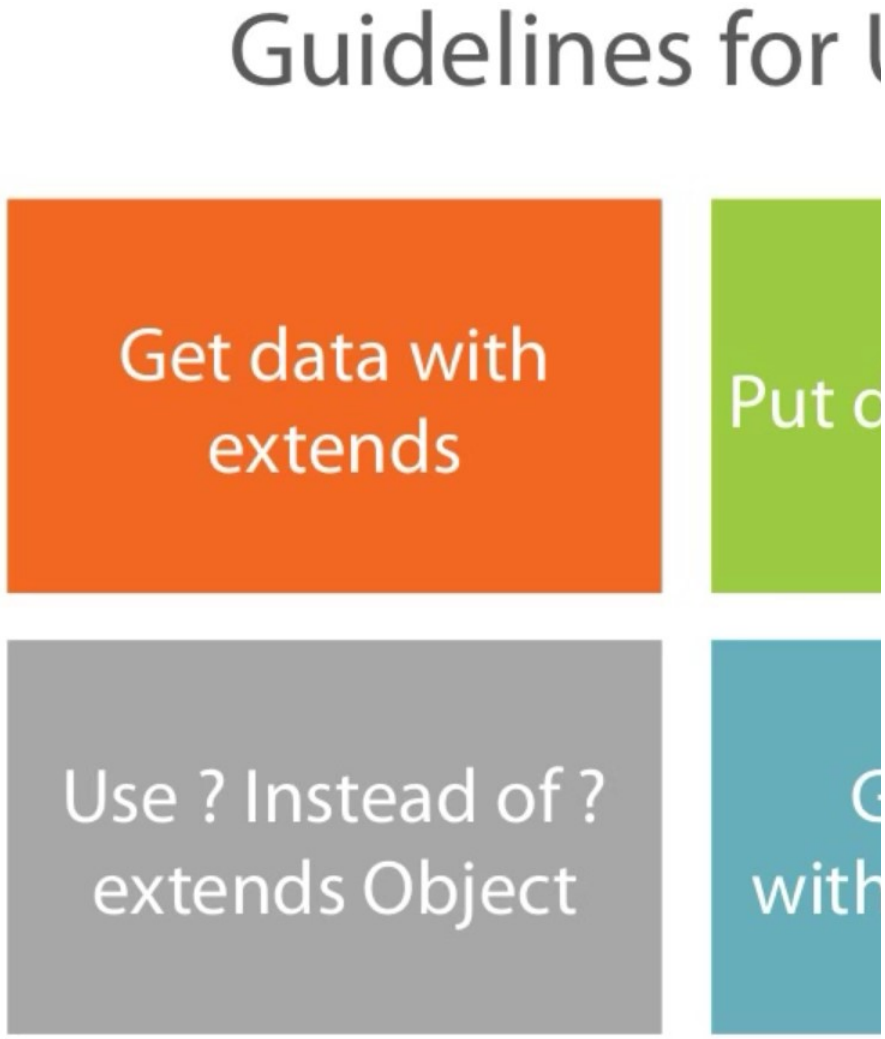
**Offline Concurrency Patterns:** Optimistic Offline Lock (416), Pessimistic Offline Lock (426), Coarse Grained Lock (438 ), Implicit Lock (449).

**Session State Patterns:** Client Session State (456), Server Session State (458), Database Session State (462).

---

# Generics in Java

- Generics introduce type safety (without copy paste) and convert runtime errors into compile time errors
- Most common use case is with collections
- Generic class can extend another generic class and passes the type parameter up
- Wildcards make generics more flexible. super -> data in; extends -> data out
- Generics are compile time construct and byte code converts all of them to raw lists without types
    - this helps in backward compatibility and hence ability to deal with legacy code
- Reflection - not all the information can be taken out from generic types directly
- Lambdas have their types inferred -> confusing errors with generics, look at the target type of the entire type as well as body of lamda
- Intersection types T extends A & B - can fake missing classes/interfaces and binary backwards compatibility




## The Algorithm Design Manual

- 3 desirable properties of algorithms: correct, efficient, easy to implement
- Reasoning correctness
  - Statement to prove (S)
  - Set of assumptions (A)
  - Reasoning that goes from A -> S
- Problem has two components
  - Acceptable input set
  - Output that satisfies specific properties
- Mathematical induction is usually the right way to verify the correctness of a recursive or incremental insertion algorithm.
- Modeling the problem: Break down the problem in precisely described and understood component problems
- Common Problem domains with their associated keywords
  - Permutations-> arrangements and ordering
    - arrangement, touring, ordering, sequence
  - Subsets -> selection of items from a set
    - cluster, committee, collection, packaging, selection, group
  - Trees -> hierarchy
    - hierarchy, descendants, taxonomy, dominance relationship
  - Graph -> arbitrary relation between objects
    - network, web, relationships, circuit

    - Points -> location in a geometric space
        - positions, records, sites, locations
    - Polygons -> region in a geometric space
        - shape, region, boundaries
    - Strings -> sequence of chars
        - text, labels, patterns
- Modeling your application in terms of well-defined structures and algorithms is the most important single step towards a solution.
- All the problem domains mentioned above work recursively, you just delete some part and see they still remain the same class just smaller

**BigO benchmarks**
- n! is useless for n >= 20
- 2^n for n >= 40
- n^2 for n >= 10000
- nlogn for n>= 1billion
- log n, practically never

| $n$ $f(n)$ | $\lg n$ |
|---|---|
| 10 | 0.003 $\mu$s |
| 20 | 0.004 $\mu$s |
| 30 | 0.005 $\mu$s |
| 40 | 0.005 $\mu$s |
| 50 | |

Class Dominance (increasing order)

- Constant functions
- Log Function (logn)
- Linear (n)
- Superlinear (nlgn)
- Quadratic (n^2)
- Cubic (n^3)
- exponential c^n (c > 1)
- factorial (n!)

Big O Algebra

- adding: keep only the dominant term
- Multiplication: keep both

f(n) dominates g(n) => Lim(n->infinity) f(n)/g(n) = 0

Full dominance function list

n! >> c^n >> n^3 >> n^2 >> n^(1+eps) >> nLogn >> n >> sqrt(n) >> log^2n >> logn >> logn/loglogn >> log log n >> alpha(n) >> 1
f(n) = o(g(n) iff g(n) dominates f(n) [ little Oh notation ]

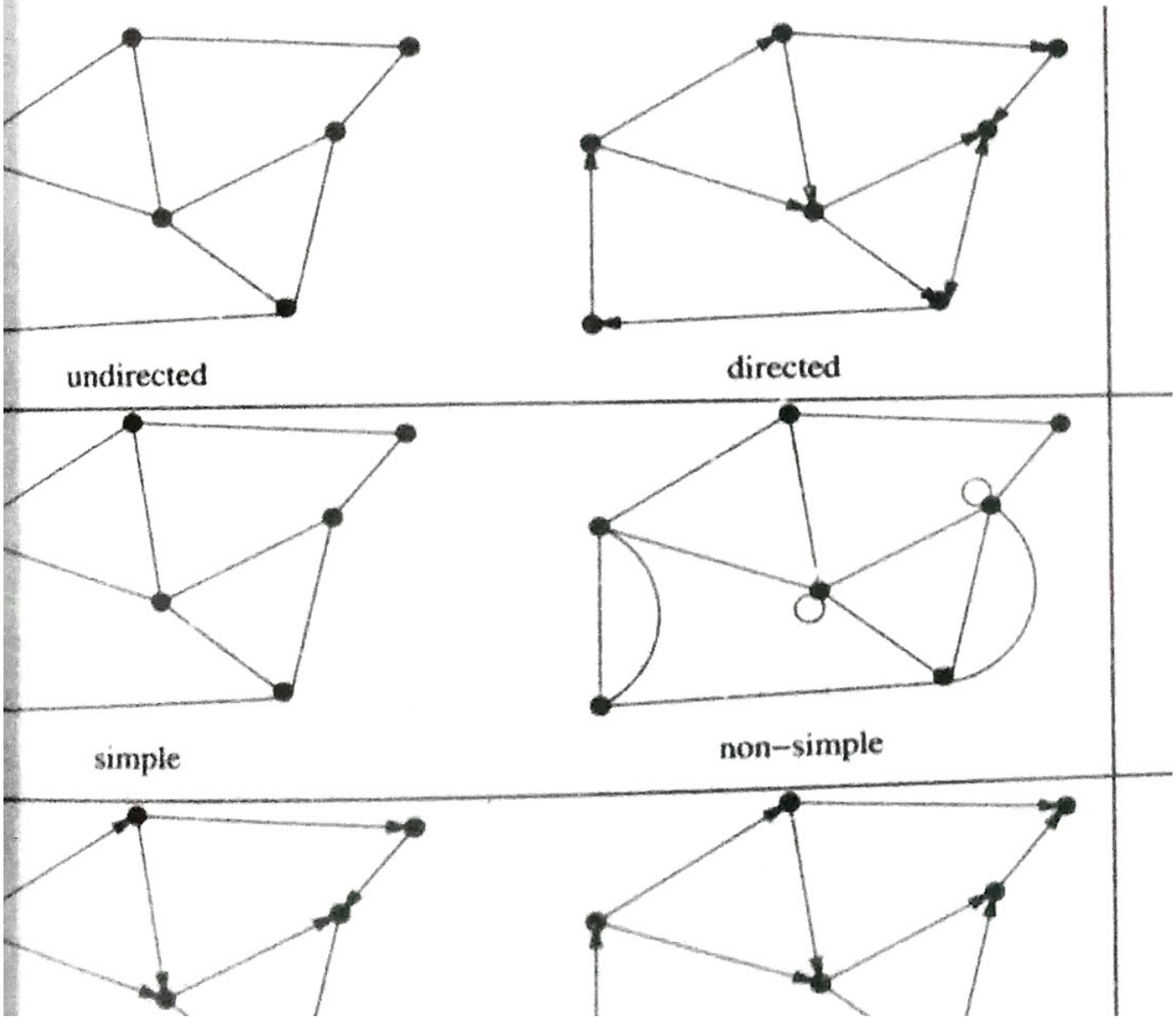
**Data Structures**

- Data structures can be neatly divided into two categories: contiguous and linked
  - Contiguous: arrays, matrices, heaps, hash tables
  - Linked: trees, lists, graph adjacency lists
- Arrays are structures of fixed sized data records such that they can be efficiently located by its index
- Container is a data structure that permits storage and retrieval of data items independently of content
  - Stacks and Queues
- Dictionaries allow storage of data items, but retrieval is based on key/content
- Stacks: Push and Pop
- Queues: Enqueue, Dequeue
- Dictionaries: search, Insert, Delete, Max, Min, Predecessor, Successors
- For most data structures, usual trade off is in maintenance vs retrievals
- Binary Trees
  - Binary Search Tree: Insertion, Retrieval, Search, Traversal
  - Binary Search tree is a binary tree when left side keys of the tree are smaller than root and right bigger
  - Search is O(h) where h is the height of the tree
  - Traversal types: pre-order (element processed first and then sub-trees), post-order (element is processed last)
  - Shape of a binary search tree depends on the order of insertion of elements. An average tree height will be O(logn). worst case is n
- Priority Queue
  - Basic operations are: Insert, find min/max, Delete
- Hashing
  - mapping a key to an integer to benefit from search using an index
  - The number of slots for hashing should be a prime number - it helps in more uniform distribution across the range
  - Collisions are always possible in hashing. Possible solutions
    - Have an array assigned to each key/idx
    - Open addressing - go to hash, if empty use, else check the next stop. Deletion is a nightmare as it needs re-inserts
- String: a sequence of characters where the order matters
- Rabin Karp - Pattern Matching Algorithm - expected near linear algorithm for string matching
  - makes use of hashing
  - hash of substring is compared with hashes of contiguous blocks of strings of same length
    - Equality -> do actual comparison
    - Inequality implies move on
  - Hashing of a string with m chars take O(m) time but given we have the hash value of previous window, we can use simple algebra to compute hash rather than looping again
- For reasonable large datasets we should always look for algorithms that are near linear complexity else they will get meaningless more often than not
- Sorting
  - Sorting should be seen as a building block of algorithms
  - Examples:
    - Search: Sort and perform bin search using O(lg n)
    - Closest pair: Sort and compare elements next to each other (linear time)
    - Duplicates: Special case of closest pair where dist = 0
    - Frequency Distribution: Sort and same items will be next to each other
    - Selection: kth largest - sort and get the kth elements
    - Convex Hulls: polygon w/ smallest area that covers a given set of n points
  - Sorting can be thought as a first step towards a performant algorithm
  - stable sorting: for equals keys, retains the original order. Most fast algos are not stable
  - Some algos get into trouble if there are too many values that are same
- Heap Sort
  - Selection sort using the right data structure ()
  - Selection sort - repeatedly extract smallest element from the remaining unsorted set
  - Heap
    - A data structure that supports the priority queue operations - insert and extract min
    - Maintains partial order, weaker than sorted order but better than random order
    - Heap Labeled tree: a node dominates its children (dominate is whatever comparator you define)
      - min heap - dominate = smaller
      - max heap - dominate = larger
    - Heaps can be constructed by inserting a new element to left most open slot
    -

- Recurrence Relations: An equation that is defined in terms of itself

**Graphs**

- A graph G = (V, E) is defined on a set of vertices V and contains a set of edges E of ordered or unordered pairs of vertices from V
- Graph Types
  - Undirected vs Directed
  - Weighted vs unweighted
  - Simple vs non simple (for non simple, multiedge, self loop are some of the possible features)
  - Sparse vs Dense
  - Cyclic vs Acyclic
    - A tree is an acyclic unidirected graph
  - Embedded vs Topological
  - Implicit vs Explicit
  - Labeled vs unlabeled
- Degree of a vertex - number of edges adjacent to it
  - A graph where each vertex has same degree is called regular graph
- Two possible data structures to represent graphs
  - Adjacency matrix and adjacency list (better most of the times)
-


undirected
directed
simple
non–simple

# Head First Java Notes (Self)

Serialization:
- If any superclass of the current class is serializable, then the current class is serializable as well even if it explicitly doesn't implement serializable (that's how interfaces work)
- Serializable is a marker interface
- Serialization saves the entire object graph (objects referenced within objects and so on). The process fails if any of the objects in the graph is not serializable
- if an instance variable needs to be omitted from the serialization process it should be marked as transient
- Usually stuff that is linked with the runtime environment is a good candidate to omit serialization
- If the object has a non serializable class somewhere up its inheritance tree, the constructor for that non serializable class will run along with any constructors above that (even if they are serializable)
- static variables not serialized
- Connection streams and chain streams: connection streams connect to a source or target, chain streams cannot do that directly.they need to be chained to connection streams
- during deserialization, all the classes of the objects need to be in the JVM
- objects are read in the same order during deserialization that they were written in during serialization
- to write a text file user FileWriter
- File object is useful and represents a file but doesn't have access to the contents of the file it represents
- a buffer is a temporary holding place
- if a class changes after serialization and before de-ser, it's not ok in cases where de-ser will see problems, but in other cases (like a new instance var added will get default value) it may be ok
- serialVersionId: by default computed based on class definition, (you can use serialver on command line for a class to get its version id). You can hardcode this ID, and then change the class without changing the id (static variable serialVersionUID)

Network Programming
- TCP port is a 16 bit unsigned number assigned to a specific server application.
- Port numbers from 0 to 1023 are reserved for "well known services" (http, ftp, smtp)

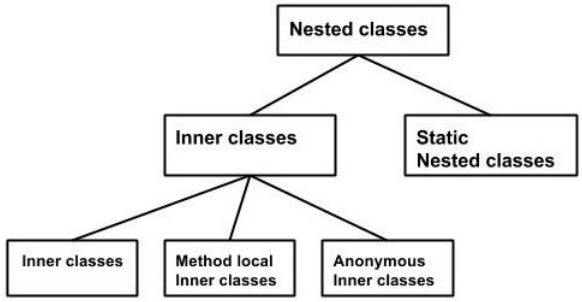
Threads
- A new thread is a different call stack
- JVM switches between the call stacks rapidly and it appears as if it's doing things in parallel
- A job that needs to run in a thread(worker) needs to implement Runnable and define run method
- States of a thread: New, Runnable, Running, Blocked, dead. Most of the time it will keep moving between Runnable and Running
- A thread scheduler is completely managed by JVM and there are no methods we can call on it
- Do not base your program's correctness on scheduler acting a certain way
    - Sleep method on thread guarantees that the thread doesn't run for the duration of sleep
- Thread safety: add synchronize to the method. It means a thread needs a key to use this method.
- If an object has two synchronized methods, it means you can't have more than one thread at a time entering any of the synchronized methods
- Synchronization has perf cost
- Synchronization can be done at a level more granular than a method: synchronize(this) { your code }
- There is a lock for the class too, it is used when we are talking about static variables
- Threads can be assigned priorities. They may influence scheduler, but nothing is guaranteed
- Synchronizing access method is generally a good idea

Generics
- Whole idea is to have type safe collections
- Three main things to know
    - create an instance of generified class
    - declare a variable of generic types
    - declaring and invoking methods that take generic types
    - creating your own generic class (not much needed)
- Naming convention:
    - E: Element that the collection should hold
    - Should be a single character (just convention again)
    - T: is used when it's not a collection
- For methods
    - if class is generified you can use the letter E/T whatever is defined in the method
    - If an undefined type needs to be used, define it before return type and use it in the method parameters
- Difference b/w ArrayList<Animal> and ArrayList<T extends Animal>:First can take _only_ Animal, second can take Animal and subclasses
- extends applies to interfaces too in case of generics
- Adding new keywords is tricky in a language as you might have used those as variable names in past and the code will break
- Sort can use list's element's comparator if not passed an explicit comparator to sort's overloaded version
- Some key usage patterns
    - List: when sequence matters
    - Set: uniqueness matters (based on equals method)
    - Map: finding something by key matters (no dupe keys, typically keys are strings, but can be any object)
- Equality: a == b (Reference equality) when a and b are references implies, they are equal if they point to the same object (i.e. store the same address in them)
    - a.equals(b) (Object Equality) uses the content stored in the references a and b to decide equality
- HashSet - when inserting an object first checks for hashCode equality, if they are diff, it assumes them to be different. If hashCodes are equal, it thinks it is 'possible' they are equal and the does equals test
- when you override equals, ensure the hashCode of equal objects come out to be same, because of the use of hashCode when populating HashSet etc
- HashCode and equality rules
    - if two objs are equal, they must return matching hashCodes. so if you override equals, override hashcode too
    - a.equals(b) should be same as b.equals(a)
    - hashcode being equal doesn't guarantee equal-ness
    - hashCodes generate unique integers
    - default equals is reference equality
- TreeSet -> uniqueness and sorted
- ? extends SomeClass - will not allow any add operations to the list. You can use T extends SomeClass too if you want to use T at some point later.

Packaging and Deployment
- Use a standard directory structure. Have source, classes as two folders. use javac -d to specify where to put the classes
- Jar - java archive. To make a jar executable, put a manifest file manifest.txt in the JAR (in classes directory) and specify the main class. (Main-Class:myApp)
- Packages help avoid naming collisions
- Sun recommends a package structure that starts with your reverse domain name and then add your project's organizational

- Jnlp - java network launch protocol

Inner Classes
- An inner class is a class defined inside another class
- it can access all the stuff from outer class (incl private stuff) and vice versa
- Usually you will instantiate an inner class only from within the outer class but if you want to instantiate it outside the outer class, here is the syntax
  MyOuter outerObj = new MyOuter();
  MyOuter.MyInner innerObj = outerObj.new MyInner();
- A class cannot be marked private unless it is inside another class



-
-

Last comments
- Strings are immutable and so are wrappers

---

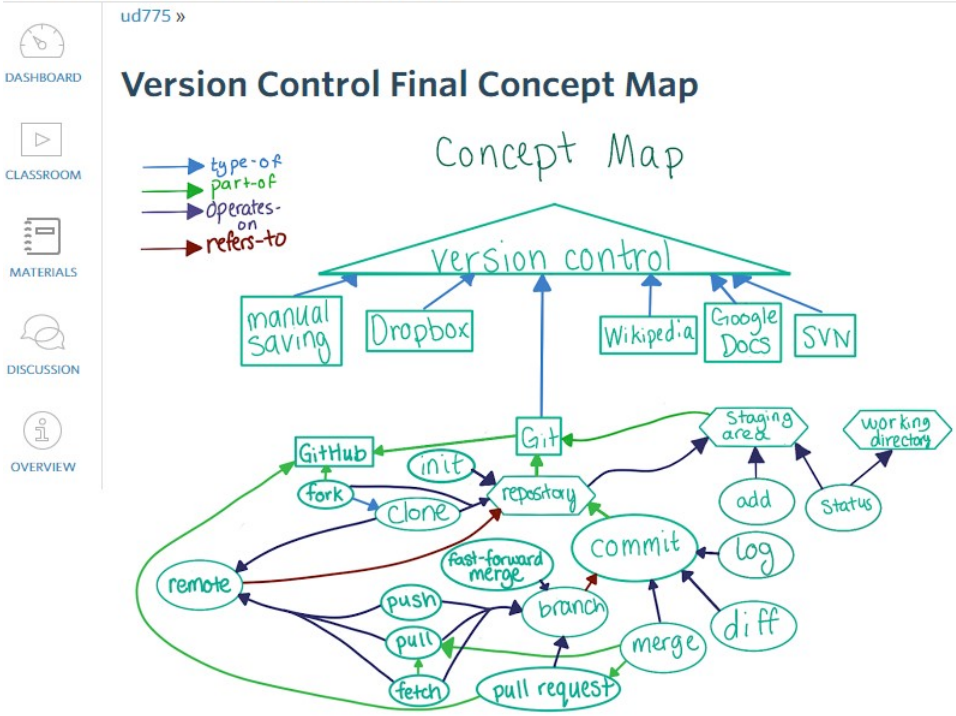
# HFJ - Github notes - Head First Java

Head First Java Notes from GIT.docx
25-11-2017 10:19, 37.2 KB

---

# Git/Github (Udacity Course)

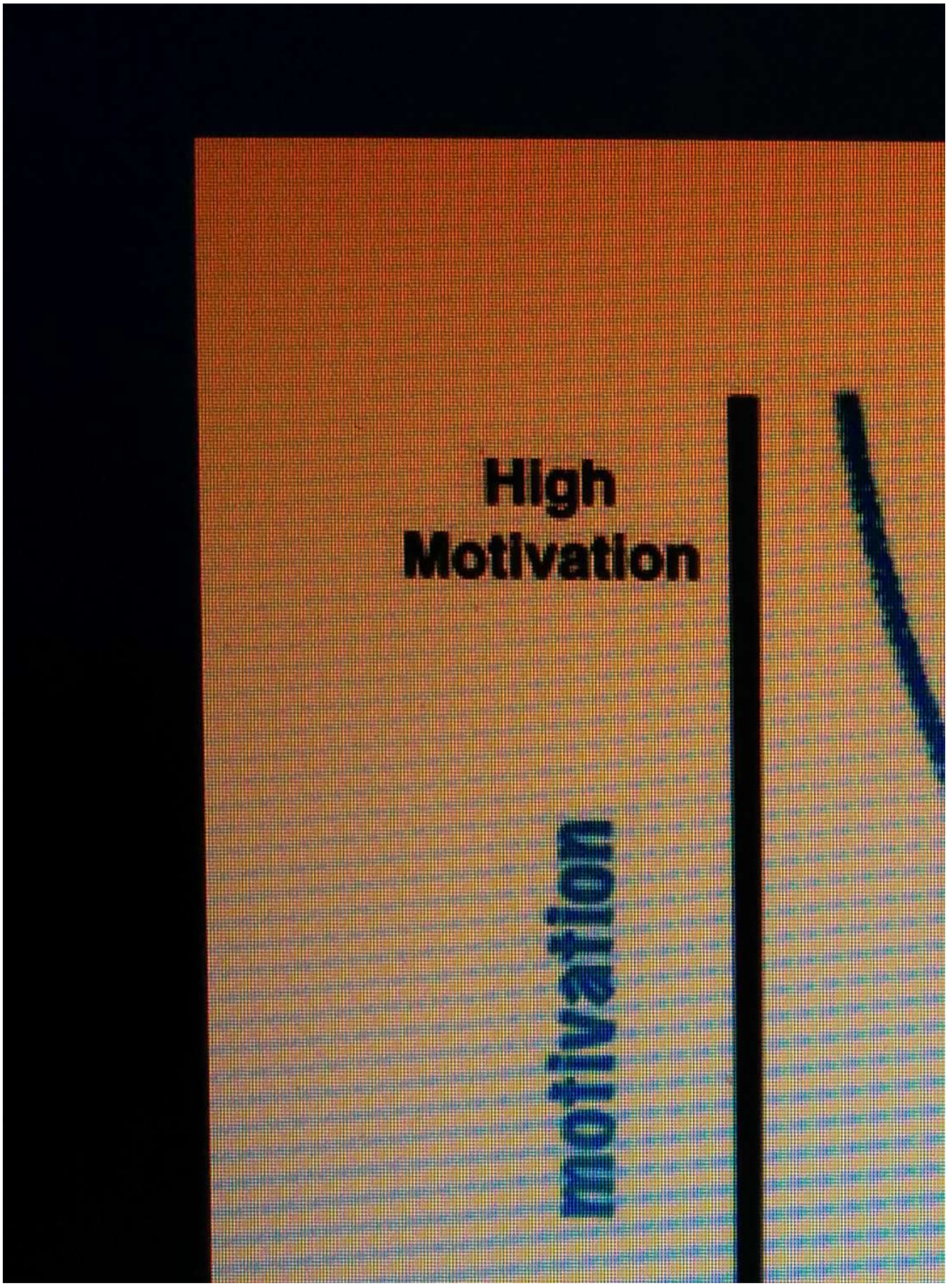
## How to Use Git and GitHub




**Gitflow**
https://nvie.com/posts/a-successful-git-branching-model/

---

## Tiny Habits


High
Motivation
Motivation

# Head First HTML and CSS

- <q> is for quotes
- As much as possible label and tag your elements in HTML, to be referenced by css and js later
- <blockquote> - same as q except its for longer quotes that need to be displayed on their own
- <br> - line break
- Elements that don't need closing tags are called void elements (br, img)

# Introduction to Web

HTTP has
- Verbs
  - Get
  - Post
- Status Codes
  - 200 (ok)
  - 404 (not found)
  - 500 (server error)
  - 100s - informational
  - 200s - successes
  - 300s - redirectoins
  - 400s - client errors
  - 500s - server errors

Front End
- Whatever runs on browser
- HTML, CSS, Javascript
  - HTML - content
  - CSS - look and feel
  - JS - Page behavior

HTML
- Anything in head tag never goes on the page itself
- It has some style though (h1 etc)
- head: metadata, not displayed in body
- body: all the content for page
- html: wrapper around everything in doc
- <p>: paragraph
- : Division, container
- <ul> - unordered list
- <ol> - ordered list
- <li> - an element in a list
- MDN: best documentation of anything web
- Semantic HTML: Use the right tag for the right situation even if you can achieve it otherwise. Example: use ol if the list is actually ordered. Another one is article, its like general content a far as the display goes, but labels it properly. More hooks for css too.
- Good code: Service to your future self
- Codepen: good site to try out web pages
- span: is like div but smaller, the smallest container possible
- span can go around only text, div can have any element within it
- self closing or void tags: the ones that dont need a closing tag
- tags can have attributes to provide more info
- Grouping, classes, Ids
  - Group by Div
  - name them using class =""
  - classes can be reused in the page
  - ids are unique
- All the labeling i.e. semantic html stuff is relevant for adding styling (css) and behavior(js)
- you can add multiple classes in the same div
- HTML tags naming - do not take it lightly, give functional name not representational so even with changes in style they remain relevant
- In any naming shorten things as much as it makes sense to (Google's rule)

# Functional Programming

Principles
- Functions as parameters
- Composition (function on top of other functions)
- Type: set of values (but no behavior). Types can be composed too
- Totality: for every input, there is an output
- Use static types for domain modeling and documentation
- Partial Application:
- Continuation: what happens next? exception handling etc..

Guidelines
- Parameterize everything

- Function type:

---

# Cryptography: The Big Picture (Pluralsight)

- Cryptography - science of encipherment and decipherment of information in order to ensure it is only available to those who need it
- Examples
  - SSL, TLS
  - Virtual Private Network
  - Hash on a downloaded file
  - Digital Signatures
- Plain text, Cipher text
- Key Concepts
  - Cipher - "system" used to create an encoded or secret message
  - key - gives us the information to apply cipher
  - CIA Triad - A good security plan incorporates a balance of all three (one impacts the other)
    - Confidentiality - keeping information private
      - Link/Message/File encryption
    - Availability - available to use - gets impacted by how strong the encryption is
    - Integrity - information is complete and unaltered
      - Digital Signatures
      - Hashes
- Symmetric Key Cryptography
  - Uses same key to encrypt and decrypt the message
  - Generally very fast
  - Strength is determined by the key
  - Stream Cipher: Encryption is performed on each bit as they come
  - Block Cipher: first breaks the message into blocks, then blocks go via the encryption system
  - DES
    - Data Encryption Standard (DES) - Invented by IBM (called Lucifer)
    - Uses 64 bit key and 16 rounds encryption
    - Cracked in 1998 using Brute Force Attack
  - Triple DES (3DES)
    - came as a quick fix for DES
    - key length - 168 bits, 48 rounds of encryption
    - commonly used by payment cards
  - AES - Advanced Encryption Standard
    - Strategic fix for DES
    - came via a competition
    - can use 128, 192, or 256 bit key
    - uses fewer rounds than DES but vary by key length and block size
    - faster and more secure than DES and 3DES
    - Other finalists were - RC4, 5, 6 (RC - (Ron's code)
      - variable key size
  - Blowfish
    - key size between 32 to 448 bits, 16 rounds, is free for all
  - Twofish - one of the contenders for AES
    - either 128,192, 256,
    - 16 rounds of encryption
  - Weakness of symmetric key cryptography
    - Need to find a way to send the key securely
    - Key management can be difficult, each person needs a different key or possible each session should have a different key
    - Non repudiation - no method to ensure that sender/receiver are the only people with the key
  - Due to its weaknesses it's not a complete solution
- Modes: Modes of operation of encryption algorithm
  - Electronic Code Book Mode
    - Break into blocks, run cipher individually on the blocks and combine
  - Cipher Block Chaining Mode
    - the first encrypted block is appended to second and then the cumulative message is encrypted and so on
    - This eliminates duplicate/recurring encrypted characters
- Rounds: # of times encryption/decryption applied
- Asymmetric Key Cryptography
  - Uses two different keys, public and private key
  - Exchange public keys between sender and receiver
  - Diffie- Hellman Algorithm
    - uses a public and private key to create a symmetric key
    - vulnerable to man in the middle attack
  - RSA - Most popular Asymmetric encryption algorithm
    - support digital signatures
    - uses one way function
  - ECC - Elliptic Curve Cryptosystem
    - support digital signatures
    - used in devices with less resources
  - El Gamal
    - extension of Diffie Hellman
- Weakness of Asymmetric cryptography
  - slower
- Caesar Cipher - Shift cipher
- Man in the middle attack

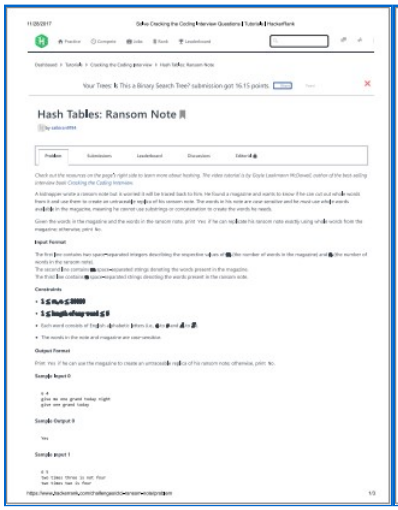
    - can spoof the keys when they are sent
- Hashing Algorithms
    - Hashing is one way to ensure that the content sent is same as content received
        - run hashing on both the contents, and compare
    - Salting
        - uses one way hash on password and a unique additional value (called salt) - generates a hash that can be stored in the database
        - you cannot reverse engineer the original value easily (for one way hash functions)
    - MAC - Message authentication Code
        - instead of salt, encrypts with symmetric algo and then run MAC on it
        - Hash MAC - add a symmetric key, key and message are hashed, other end uses the symmetric key and MAC to ensure the content is correct
        - CBC - MAC -> MAC and content are sent.. details not that relevant
    - MD2,4,5 - Message Digest Algorithms
        - MD5 is newer version of MD2 and 4, but even that is found to be vulnerable to collision attacks
        - SHA was the answer after MD5 was found inadequate
    - SHA-1 (result is 160 bit), SHA256 (SHA-256)
        - SHA-1 was broken in 2017
        - SHA-256 is the improved version
    - Haval
        - Modification of MD5, result is of variable length, still found vulnerable to collision attacks
    - Tiger
        - 192 bits hash, designed be faster than SHA-1 and MD5
- Digital Signatures
    - Cryptographically proving identity
    - Public/private keys are sued - provides non repudiation
        - Non Repudiation - ensuring that a person cannot deny that they were responsible for an action
- Standards that combine solutions for **confidentiality, integrity and Authentication**
    - S/MIME
    - PGP - Pretty Good Privacy
    - PKI - Public Key Infrastructure

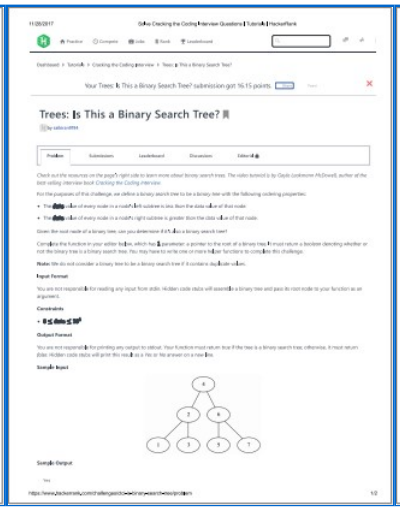
---

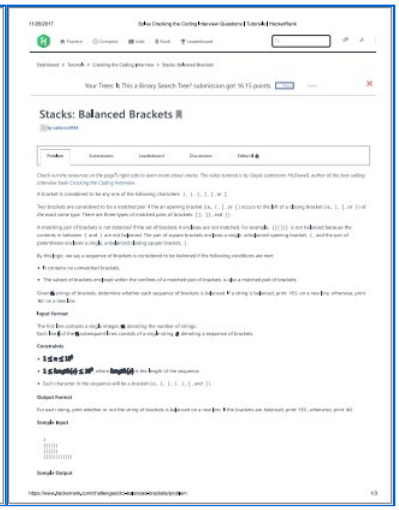
# CTCI Notes

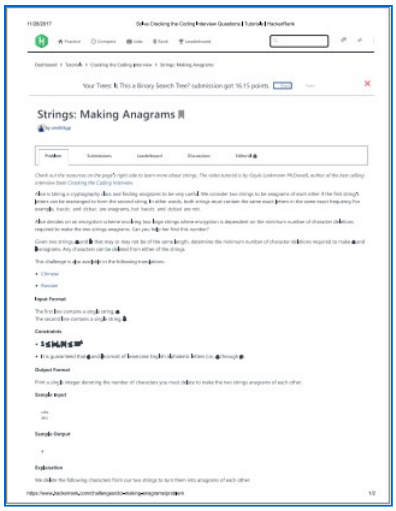
---

# CTCI Problems






# History of Internet

- Link Layer
  - Only worries about physically connecting two computers and transfer data
  - works with an actual physical address
  - ethernet/wireless are solutions for this
- IP Layer
  - Decides where to route a packet
  - Router table - keeps tab on some information useful for routing
  - IP address - two parts network address and then the host address
  - IP protocol only deals with the network transfer
  - The network takes over for the remaining communication
  - DHCP - Dynamic host configuration protocol - you can acquire an IP as you become part of a network
  - At home we all get 192.168.... this is not a real address, it's a non routable address and works within the network, router when talks to the outside world  translates it to a real one
  - Network address translation: takes care of doing the conversion (@ access point)
  - routers will have an IP of 192.168.0.1
  - traceroute - for debugging network
    - sends a packet with ttl = 1, so first guys itself fails it and sends a note back
    - then sends with ttl = 2 and so on
  - Routers can get into cycle issues where you keep going round and round infinitely in a cycle w/o going anywhere
    - Solved using TTL - time to live - every time you hit a router the number goes down and you hit zero, the message gets dropped (30-40 hops is what is given)
  - IP sends stuff really fast across the world - determined by speed of light and the content size (usually a round trip across the world is 15 (5-20)C hops and 0.25s, in closer areas things can reach in 1/100th of a second or 1/10th of a second
- Transport Layer: Handle errors and make best use of resources
  - keeps a message until it receives an acknowledgment from the destination
  - the moment it knows a packet is received, it throws away that packet
  - this is the genius of the Internet, the intermediate nodes don't need to store anything and are allowed to fail so they can be extremely fast
  - this storage is held by the sender (user essentially), so our computers and devices are responsible for storing it until you know it has reached
  - (Van Jacobson) slow start algorithm is what saved us in 1987 when everybody thought Internet will fail because the backend slowed down badly
    - Van co-authored traceroute
- Application Layer
  - WWW
  - Mail
  - Videos streaming
  - FTP
  - Many others
  - assumes a reliable connection via TCP layer and protocols define the rules of communication
  - port number - defines a service within a host/machine, so same IP address can serve multiple services, it's almost like a telephone extension
  - telnet command opens up a tcp connection, you can directly make requests to get web pages following the specifications of HTTP (RFC 1945)
  - different services have stand port numbers

| Port | Service name | Transport protocol |
|---|---|---|
| 20, 21 | File Transfer Protocol (FTP) | TCP |
| 22 | Secure Shell (SSH) | TCP and UDP |
| 23 | Telnet | TCP |
| 25 | Simple Mail Transfer Protocol (SMTP) | TCP |
| 50, 51 | IPSec | |
| 53 | Domain Name Server (DNS) | TCP and UDP |
| 67, 68 | Dynamic Host Configuration Protocol (DHCP) | UDP |
| 69 | Trivial File Transfer Protocol (TFTP) | UDP |
| 80 | HyperText Transfer Protocol (HTTP) | TCP |
| 110 | Post Office Protocol (POP3) | TCP |
| 119 | Network News Transport Protocol (NNTP) | TCP |
| 123 | Network Time Protocol (NTP) | UDP |
| 135-139 | NetBIOS | TCP and UDP |
| 143 | Internet Message Access Protocol (IMAP4) | TCP and UDP |
| 161, 162 | Simple Network Management Protocol (SNMP) | TCP and UDP |
| 389 | Lightweight Directory Access Protocol | TCP and UDP |
| 443 | HTTP with Secure Sockets Layer (SSL) | TCP and UDP |

    - Internet is never 100% right, but is never down, most of the architecture is still the one created in 1970s
    - Internet is not built for perfection but for healing itself

- Domain Naming System (sits somewhere between tcp and Internet or between ip and link)
    - converts names to network addresses

Public Key Encryption

- Diffie and Hellman
- Depends on two keys, mathematically related to each other
    - Public key and Private key - public is open, private key has to be kept secret
    - Message encrypted with public key can be decrypted with the private key
    - It is possible to break this encryption w/ sufficient computation, but that kind of computing power is not available (so it's impractical)
- Public key math - multiple two numbers (really fast)
- Private key math - prime factorize a large number (horribly slow)
- Receiver sends a public key - sender uses it to encrypt the message, receiver has the private key (related to the second prime number) which can quickly decrypt it
    - Computing the private key will need huge computation
- SSL sits on top of TCP
- TCP connection b/w two hosts is called socket
- SSL ~ TLS (Transport Layer Security) ~ HTTPS

Digital Certificates

- public key certificate - is an electronic document which uses a digital signature to bind a public key with an identity

Certificate Authority

- entities that issue digital certificates
- trusted third parties, trusted by both - the owner of the certificate and the people relying on the certificate
    - example - verisign
- Signed certificates are provided after truly validating the identity of the entity
- your OS has a list of certificate authorities as part of the OS
- Key creation process
    - CA creates a public key and private key - hands over the public key to guys building OSes
    - So when you have a machine with OS installed, you have the public key from the vendor
    - Amazon comes in - generates a public key - private key pair
    - Amazon sends its public key to verisign
    - Verisign creates a message digest using its private key

---

# Finance training by kamlesh jain

owner of Resources → ensures utilization of resources

FISCAL AUTHORITY – POLITICALLY A

MONETARY AUTHORITY – PROFESSIONALLY

owner of money

supposed to be indep

→ fixes – amount of money in the s

[ MPC – monetary policy committ

China – political system – communist

economic — " — – capitalis

Macro Economics – Country level

RBI → Bank of all banks &
bankers to the Govt.
does not take deposits

Key Economic Indicators

(1) GDP – measure of utilization resource

(2) Unemployment – a resource that is (resp
consuming but not producing

(3) Inflation – purchasing power of money

Fiscal deficit / Surplus —

Budget is a cash flow statem

INFLOW — OUTFLOW = FISCAL DEF

PIIGS — Portugal, Ireland, , Gre
(bankrupt)

Germany — voting rights are given o
pass an economics test

Balance sheet — statement of status

Profit & Loss → Statement for a period — r

# SOURCES OF MONEY/CAPITAL

Date :

MON TUE WED THU FRI

- SAVINGS FROM PAST } Equity - owner's
- CURRENT INCOME }
- Borrowed } Debt

Investment
Return
(There is a

Equity markets are the smallest Fin markets, debt
(Fixed

× de

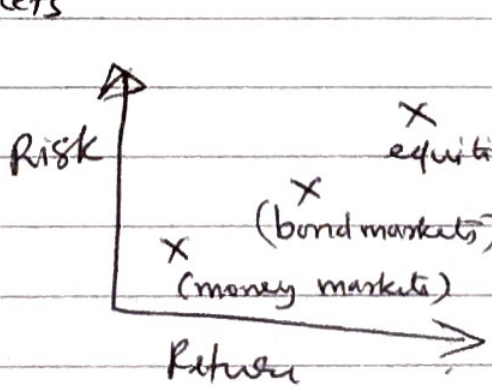
4 types of markets

Eq

Fixed Inc

Commodities

Currencies




Eq

Debt

Ed Fi

# Bonds

- everything is driven by interest
  - → Real Rate
  - → Inflation
  - → Risk Pre

Interest Rate Graph ( Yield curve

Govt is risk free

RR + IP = Risk Free Re

RP = credit spread

nominal Interest Rate =

PRIMARY MARKET

↳ ISSUER & INVESTOR

home loan

fixed deposit

EPF

SECONDARY MARKET

↳ investors transact

issuer n

Excitement is here

A Fin market can be create

Players

Process

Products

Date :
MON TUE WED

BANK
↳ is in the business of managing trust

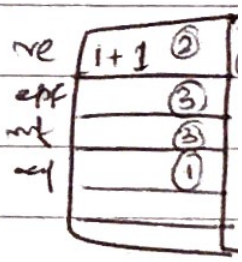
NIM- Net interest margin

ALM- asset liability management

↓

IB (NIM → fees)

IB capabilities
1. advising / fees / commission
2. Execution Ability
3. Product innovation
4. Research




| Buy Side | Sell Side |
|---|---|
| (taking money) | (only advice) |
| (deciding for client) | (facilitate execution) |
| (managing their money) | (NO DECISION FOR CLIENTS) |
| (Asset mgmt) | |

trading a/c = your a/c w/ broker for trad

Demat a/c – electronic storage of securities

---

Custodian – guarantees a transaction

broker – has a license to an exchange
(takes a commission for access t

prime brokers – brokers for Hedge funds

Regulators – creates the rules of the gam
Banks – RBI
securities – SEBI
insurance – IRDA
if noone owns it – RBI

Exchange – platforms to do transactions

Rating Agencies – provide ratings

Market Maker – create a market
(ex: a large order that can crash
makers can take it & do it o

**Places**

OTC – dark market (bigger m
Exchg – lit market

Equity Markets

Partnership / Proprietorship

(1) ↳ unlimited
(company/

(2)

Shareholders have residual rights

dilution by x% → x% new sh
Divestment

DRHP – draft red herring Prospectu
( it is like the n

✓ Economic Machine by Ray Dalio

→ BSE - tech stack ??

→ Rupay ??

money market funds / ~~Rentals~~

CP/CD - fixed inc. products

nseindia.com

→ too big to fail

→ Meltdown (Al-jazeera)

→ Margin call

→ Bill Ackman - everything you need to know

{ kh

---

Common Shares

Preffered Shas — no voting rigl

Depository Notes (Receipts)

(ADR) - american Dep ~~Rates~~ Re

GDR - Global Dep Re - —

IDR = BDR = Indian/ Bhar

FIXED INCOME

MONEY MARKET

↳ market for money
(short term)
< 12 months

OTC
only w/ high denomina
w/ interest
highly liquid
takes care of short term cash flow imbala
b/w big corporates / in
non-zero but smaller i
no collateral
also known as liquid or

~~BOND MARKET (12 MONTHS)~~

PRODUCTS

(1) T-bills — Treasury Bills
(2) CD — Certificate of De
(rate, tenu etc
customized by F
a deposit can only

# BONDS (>12 MONTHS)

$140 trillion = total of fin. instr. of

$80 trillion = bonds

bond → "I owe you"

indemnity → guarantee

doc (FD) – called indenture

debenture = debt + indenture

whatever is in doc is called c

Terms (Bonds)

PV / FV – Present / Face value

coupon rate – rate committed during
(remains fixed thru the

Term – time remaining

Frequency –

Yield – effective return

FV – future value

→ spread

Clean Price = price of the commodity

Dirty Price = includes other stuff

market price + int. acc
(clean)

DAY COUNT CONVENTION

① [ 30 = en
[ 360 = e
→ AMERICAN.

② India = actual d / actual

③ 30/365 (used whe

5 | SDL

Masala Bonds (investors outside India denominated

Co

Fl

# FOREX MARKET

Date :

MON TUE WED THU FRI SAT SUN

- almost entirely OTC
- no holiday, open 24x7x365
- no standardized rates
- no commission, money is made on spread
- 4th decimal is significant (called pips)
- daily traded volume is 6 trillion dollars
- 80-90% trades in 4 pairs USD/JPY, USD/EUR, USD/ & (GBP/USD=??)
- buy rate & sell rates are diff; diff b/w the two is sp
- Regulator is RBI

Base curr / Counter currency ⇒ 1 Base currency / cou

USD/INR = 1/63.44

90% cases USD is base currency except GBP, EUR

Quotation 63.10/90 = USD/INR

63.10 — BC — buy rate
63.90 — BC — sell rate

Read from dealer's perspective

# Big Data on AWS

Typical Hadoop distribution

- Hadoop consists of Hadoop (HDFC + MapReduce) along with HBASE
- HBASE tables are HDFS files
- Hive - SQL abstraction layer over Hadoop
- Pig - scripting - abstraction over MapReduce (Pig Latin)
- Sqoop - Move data between Hadoop and other types of storages
- Mahout - ML
- Flume - Log File integration tool

AWS Hadoop distribution

- MapReduce, HDFS
- Hive QL, PigLatin over MapReduce
- Impala - like Hive - sql abstraction over Hadoop
    - Hive works as a batch system
    - Impala works as an interactive system

MapReduce

- Divide and conquer
- Map - divide the work and do its work
- Reduce - take map output and reduce it to single value

Job Flow

- is a request to spin up and ephemeral hadoop cluster with configuration specifics
    - Data center/AZ
    - Number of nods
    - VM types used
    - Security, logging, other settings
- Pin point the right distribution with required components
    - hbase, impala, pig, hive

Default user name for EMR is hadoop
HDFS commands

# Design Patterns in Java: Behavioral

**Chain of Responsibility**

- Examples
    - java.util.Logging
    - servlet.Filter
    - Spring Security Filter Chain

Design

- Chain of receiver objects
- Handler is interface based
    - has handle method
    - has a scuccessor
- ConcreteHandler for each impl
- Each handler has a ref to next
- Each handler tries to handle something given to it, if it can't it passes it to the successor

The chain can be changed without changing the individual classes

Pitfalls

- No guarantee that the chain handles everything, something might not get handled at all
- chains can be too long

**Command Pattern**

- Encapsulates request as an object
- Object Oriented callback
- sender is decoupled from processor
- Often used for undo functionality
- Example
    - java.lang.Runnable
    - javax.swing.Action

**Design**

- Command Interface
    - contains an execute method
    - sometimes it can have undo/unexecute method
- ConcreteCommand
    - gets the receiver
- Invoker
- Receiver

Pitfalls

- Dependence on other patterns
- May lead to duplicated code

Possibly the second most used pattern after singleton

**Interpreter Pattern**

- Represents grammar
- Interpret a sentence
- Map a domain
- Examples
    - java.util.Pattern
    - java.text.Format

Design

- Abstract Expression
    - Terminal Expression
    - Non Terminal Expression

Pitfalls

- Complexity
- Class per rule
- Specific Case

Summary
- Pattern when we are defining a grammar, rules or validations use case

**Mediator Pattern**

- Loose coupling (of actions and objects they act upon)
- Hub Router
- Examples
    - Java.util.timer
    - java.lang.reflect#invoke

**Design**

- Interface based
- Concrete class
- minimizes inheritance
- mediator knows about colleagues instead of objects knowing about each other
- two key components are Mediator and Colleague

**Pitfalls**

- Avoid creating a deity object thats everything to everyone
- Limits sub-classing
- Over or with command - generally works well with command pattern

**Memento Pattern**

- Externalize an objects state to make it possible (usually) for rollback
- Examples

    - java.util.Date
    - java.io.Serializable

**Design**

- Class Based
- Key components
    - Originator
    - Caretaker
    - Memento
        - Magic Cookie

Basically keep pushing a saved value to a stack in caretaker and implement revert and save methods in memento

Pitfalls

- can be expensive
- delete/history

Summary

- captures state
- used to recreate the state
- identical to command??? not clear how

**Observer Patter**

- Decoupling pattern
- a subject has one to many observers
- common use case - event handling
- also called pub/sub
- also used in MVC where view is event driven
- Example
    - java.util.Observer
    - java.util.EventListener
    - javax.jms.topic

Key Components

- Subjects
- Objects will register with the subject
- Observable
- Views are Observers

Pitfalls

- unexpected things can happen given their are unknown observers
- large sized consequences as you don't control who is observing
- what changed?? debugging can get difficult

Summary

- decoupled communication between objects if they attach to same subject

**State Pattern**

- Storing the app state in an object instead of many variables
- Reduces cyclomatic complexity
- Examples in Java
    - None
- Design
    - Abs class/interface
    - Each State is class based
    - Context unaware
- Design elements
    - Context
    - State
        - Concrete State A
        - Concrete State B

Lot of times states are represented by variables (int, boolean etc)

Fan example - on, off, low, medium each has a Class representing it, action is pulling the switch
It's a state machine where the object knows about all its states, individual states know about where they have to go to when some action is done
get wary if you are writing a lot of if-else statements

**Pitfalls**

- Knowing all states
- Too many classes (1 class per state)

It is identical to strategy pattern except the purpose being different

**Strategy Pattern**

- Remove conditional statements
- Behavior is encapsulated in classes
- Client aware of strategies
- Difficult to add new strategies
- Examples
    - Comparator

Design

- Abstract Base Class
- Concrete Class per strategy
- Remove if/else conditions
- Strategies are independent
- Context, Strategy, ConcreteStrategy

Context receives the call, decides which strategy to call (all extends the same base class)

Pitfalls

- Client needs to know what all algorithms are available
- increased number of classes

Summary

- externalizes algorithsm
- class per strategy
- client knows different strategies
- reduce conditonals

---

# Ocean Study Notes

URI/URN

> A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.
>
> URI can be classified as name, or locater or both
>
> One can classify URIs as locators (URLs), or as names (URNs), or as both. A Uniform Resource Name (URN) functions like a person's name, while a Uniform Resource Locator (URL) resembles that person's street address. In other words: the URN defines an item's identity, while the URL provides a method for finding it.

URL = URI + access mechanism (http://, ftp://)

---

# Introduction to Bitcoin  and Decentralized Technology

Bookkeepers

- get some money periodically for doing their job
- Money creation slows down

A transaction can reach to different people at different times. Majority decides what is the right order of transaction

Proof of Work - 1 vote per CPU
new vote once in 10 minutes

Entire ledger can be downloaded and verified

Big ideas
- Privacy and Pseudo-anonymity
- Openness
- Cryptocurrency
- Decentralization
- Trustless
- Programmable Money

You lose your private key, the money is effectively destroyed

Decentralization Downsides
- less efficient - high value transactions need an hour
- Proof of work - burns energy by design
- Changing more difficult, widespread agreement needed

Acquiring Bitcoin
- exchanges - ex: coinbase.com
- buying btc using fiat currency needs you to reveal your identity

Bitcoin Wallets
- Apps or websites
- Full Node -> full ledger
- Private keys control funds - once lost there is nothing you can do, nobody else in the world will have it
- wallet is just a collection of private keys
- Code storage - send the private key to a computer not on network
- hardware wallets - devices
- Always backup your wallet
- brain wallet - a sequence of words that are enough to generate your private key

Mining - no longer cost effective
Miners = bookkeepers

bitcoin.org - compares wallets

---

# Tomcat/Web Server

- The HTTP 1.1 supports seven types of request: GET, POST, HEAD, OPTIONS, PUT, DELETE, and TRACE
- A socket is an endpoint of a network connection (client side)
    - A point in time connection when client wants something
- ServerSocket - waits for incoming request. Once the server socket gets a connection request, it creates a Socket instance to handle the communication with the client.
    - Another important property of a server socket is its backlog, which is the maximum queue length of incoming connection requests before the server socket starts to refuse the incoming requests.

serverCocket.accept is where it waits (await method of HttpServer class) for a client to contact, then all you have to do is parse Http request and present an output compliant with Http response specifications

---

# Spring MVC4

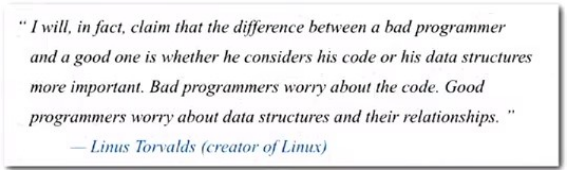
- Web application framework
- Frameworks
    - Struts:
    - Struts 2: (very diff from struts)
    - tapestry - governed by 1 person (pojo based)
    - Wicket
    - GWT (google, rich UI, hard to learn)
    - Oracle
    - JSF
    - Stripes - lightweight - similar to Spring MVC (mostly view)
    - Spring MVC - unobtrusive, takes a lot of nice components from other frameworks
- web f/w built around principles of spring
    - POJO based and interface driven
    - Based on dispatcher servlet/ Front Controller Pattern
        - MVC - Model view controller
    - lightweight, unobtrusive
    - meant to improve struts, so fixed the shortcomings
    - Annotation based configuration

    - restful
    - i18n
    - seamless integration with other spring services/beans
    - stable releases - backward compatible
    - active community
    - jee based
- History
    - released in 2003 (spring) - interface driven development
    - 2004 - 1.0, mvc got added
    - 2006 - won jolt award
    - 2006 - 2.0, 2.5
    - 2007 - annotations
    - 3.0 - 2009
- Architecture
    - built on top of java servlet API
    - uses Servlet front controller
- Request/Response Lifecycle
    - Incoming requests hit front controller
    - delegates to one of our controller
    - delegates to backend (DB, services etc)
    - returns a model to controller
    - rendering of model is done by someone - goes to front controller
    - front controller sends it to a view template
    - returns control to front controller
- Key Terms
    - DispatcherServlet - entry/config point for application
    - Controller - command pattern object that handles the request and determines which view to route to
    - RequestMapping - url and request type that a method is tied to
    - ViewResolver - locates JSP pages or whatever view tech you need
    - servlet-config - config file per dispatcher servlet (center for wiring everything)
    - POJO
    - Bean
- Standards
    - put views under WEB-INF
        - security
        - control user experience
    - Controllers
        - POJOs with annotation (@Controller)
        - Path set using @RequestMapping
- MVC 4 - new additions
    - Java COnfig
    - Java 8
    - JEE 6/Servlet 3+
    - REST Support
- maven - explicitly defined dependencies override transitive dependencies
-

---

# Algorithms, Part II

- Algorithms: method for solving a problem
- Data Structure: method to store information
- First algorithm existed during the time of Euclid
- Turing formalized it in 1930s




**General Rules to Reinforce for solving hard problems**

- Build API First
- Build base functions
- build logic

**Graphs**

**Undirected Graphs**

- Set of vertices connected pairwise by edges
- Thousands of practical applications
    - hundreds of algorithms on graphs are out there
- a very challenging branch of CS

## Graph applications

| graph | vertex | edge |
|---|---|---|
| communication | telephone, computer | fiber optic cable |
| circuit | gate, register, processor | wire |
| mechanical | joint | rod, beam, spring |
| financial | stock, currency | transactions |
| transportation | street intersection, airport | highway, airway route |
| internet | class C network | connection |
| game | board position | legal move |
| social relationship | person, actor | friendship, movie cast |
| neural network | neuron | synapse |
| protein network | protein | protein-protein interaction |
| molecule | atom | bond |

Terminology

- Path: sequence of vertices connected by edges
- Cycle: Path whose first and last vertices are the same
- Connected: Two vertices are connected if there is a path between them

Graph Processing Problems

## Some graph-processing problems

Path. Is there a path between $s$ and $t$?
Shortest path. What is the shortest path between $s$ and $t$?

Cycle. Is there a cycle in the graph?
Euler tour. Is there a cycle that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once.

Connectivity. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges
Graph isomorphism. Do two adjacency lists represent the same graph?

Some problems are easy, some hard, some intractable so far. It is a really hard area
Anomalies - self loop, parallel edge

## Graph API

| public class Graph | |
|---|---|
| Graph(int V) | *create an empty graph with V vertices* |
| Graph(In in) | *create a graph from input stream* |
| void addEdge(int v, int w) | *add an edge v-w* |
| Iterable<Integer> adj(int v) | *vertices adjacent to v* |
| int V() | *number of vertices* |
| int E() | *number of edges* |
| String toString() | *string representation* |

```
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
```

read graph from input stream

print out each edge (twice)

## Typical graph-processing code

*compute the degree of v*

```
public static int degree(Graph G, int v)
{
   int degree = 0;
   for (int w : G.adj(v)) degree++;
   return degree;
}
```

*compute maximum degree*

```
public static int maxDegree(Graph G)
{
   int max = 0;
   for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
         max = degree(G, v);
   return max;
}
```

*compute average degree*

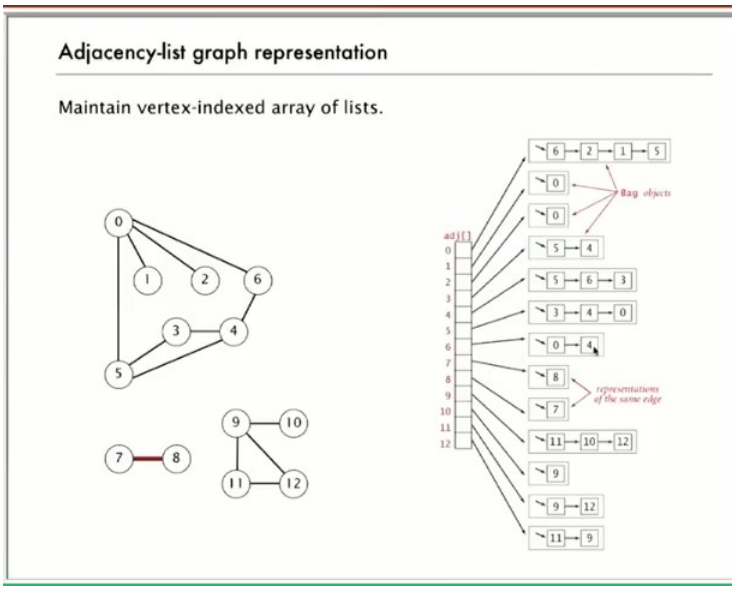
```
public static double averageDegree(Graph G)
{  return 2.0 * G.E() / G.V();  }
```

*count self-loops*

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;   // each edge counted twice
}
```

**Graph Representation**

- Set of edges representation (not used that much as it leads to inefficient implementations of the APIs)
- Adjacency Matrix (not widely used as it needs to much storage)
- Adjacency List: Traversal is linear, and in real life graphs are generally sparse




| representation | space | add edge | edge between v and w? | iterate over vertices adjacent to v? |
|---|---|---|---|---|
| list of edges | E | 1 | E | E |
| adjacency matrix | $V^2$ | 1 * | 1 | V |
| adjacency lists | E + V | 1 | degree(v) | degree(v) |

**Depth First Search**

- Tremaux Maze Exploration
  - Unroll a string behind you
  - Mark each visited intersection and each visited passage
  - Retrace steps when no unvisited options
    - The idea is to not go to the same place
- Systematically search through a graph
- Idea is to mimic Maze exploration

**DFS (to visit a vertex v)**

**Mark v as visited.**
**Recursively visit all unmarked vertices w adjacent to v.**

- Design Pattern for graph processing
  - Decouple graph type with graph processing
    - Create a graph object
    - Pass the graph to a graph processing routine
    - Query the graph processing routine for information

```
public class Paths

                    Paths(Graph G, int s)      find paths in G from source s
                  boolean hasPathTo(int v)     is there a path from s to v?
      Iterable<Integer> pathTo(int v)          path from s to v; null if no such path
```

### Depth-first search

Goal. Find all vertices connected to $s$ (and a corresponding path).

Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.
  `(edgeTo[w] == v)` means that edge `v-w` taken to visit `w` for first time

- DFS marks all the vertices connected to so in time time proportional to sum of their degrees
- After DFS is done, can find vertices connected to s in constant time, can find a path to s in time proportional to its length
- Not always optimal
- One Application - Photoshop flood fill

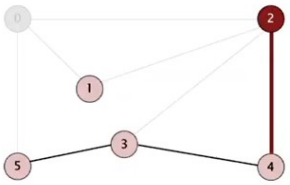
**Breadth First Search**

- Uses a queue

### Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.




| queue | v | edgeTo[] | distTo[] |
|---|---|---|---|
| | 0 | – | 0 |
| | 1 | 0 | 1 |
| 4 | 2 | 0 | 1 |
| 3 | 3 | 2 | 2 |
| 5 | 4 | 2 | 2 |
| 1 | 5 | 0 | 1 |

- BFS calculates the shortest path
  - Use-case: routing in a communications network
  - Kevin Bacon numbers
  - Erdos numbers

**Connected Components**

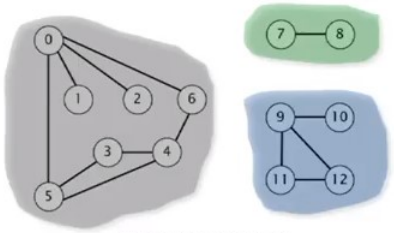
- Vertices are connected if there is a path between them
- Connectivity Queries - is v connect w, how to answer that in constant time
  - Adjacency Matrix - can do this (very sparse and very large need of space)
  -

## Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: $v$ is connected to $v$.
- Symmetric: if $v$ is connected to $w$, then $w$ is connected to $v$.
- Transitive: if $v$ connected to $w$ and $w$ connected to $x$, then $v$ connected to $x$.

Def. A connected component is a maximal set of connected vertices.




3 connected components

| v | id[v] |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 1 |
| 8 | 1 |
| 9 | 2 |
| 10 | 2 |
| 11 | 2 |
| 12 | 2 |

## Finding connected components with DFS

```
public class CC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

   public int count()
   public int id(int v)
   private void dfs(Graph G, int v)

}
```

DFS code

```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}
```

number of components

id of component containing v

all vertices discovered in same call of dfs have same id

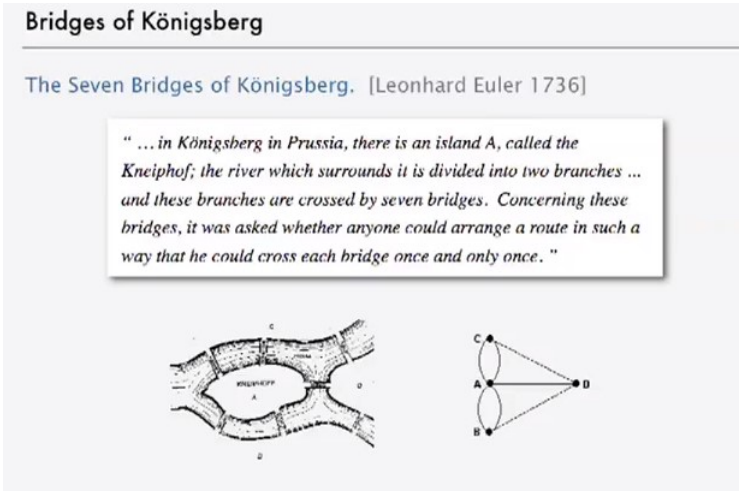
**Applications**

- STDs - all folks in a connected component will likely be infected
- Particle Detection in an image - image is a pixel graph and similar colored stuff can be connected components in turn making up the images

**Challenges**

- Is graph Bipartite: you can divide the vertices in two subsets such that each edge connects elements across the groups

    - DFS can solve it - little tricky
- Finding a cycle in a graph
    - DFS can solve it - fairly easy
- Bridges of Konigsberg - can you walk so that you go over each bridge only once
    -




- Euler Tour
    - is there a cycle that uses an edge exactly once
    - can be done if all the vertices have even degree
- Hamiltonian Cycle problem
    - Find a cycle that visits every vertex exactly once
    - Traveling salesperson problem
    - NP Complete Problem
    - Intractable - nobody knows an efficient way to do this
- Are two graphs identical (except for the vertex names) - Graph isomorphism problem
    - factorial complexity - try naming all the combinations
    - Open problem - noone knows the nature of difficulty
- Laying out a graph in the plane without corssing edges
    - linear time algorithm based on DFS
    - very complicated for most people (Tarjan 1970)

**Directed Graphs**

- Edges have direction (order of pair matters)
- InDegree (number of arrows coming in), OutDegree (number of arrows going out)
- Examples: Road Network
    - One way streets
    - Political Blogosphere
    - Interbank loan graph
    - Implication Graph
    - Combinatorial Circuit
    - WordNet Graph
- Digraph Processing Problems
    - Is there a directed path from s to t
    - Shortest path from s to t
    - Topological sort - draw a digraph so that all edges point upwards
    - String Connectivity - Is there a directed between all pairs of vertices
    - Transitive Closure: for which vertices v and w, is there a path from v to w
    - PageRank - importance of a web page
- API

## Digraph API

| public class Digraph | |
|---|---|
| Digraph(int V) | *create an empty digraph with V vertices* |
| Digraph(In in) | *create a digraph from input stream* |
| void addEdge(int v, int w) | *add a directed edge v→w* |
| Iterable<Integer> adj(int v) | *vertices pointing from v* |
| int V() | *number of vertices* |
| int E() | *number of edges* |
| Digraph reverse() | *reverse of this digraph* |
| String toString() | *string representation* |

```
In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "->" + w);
```

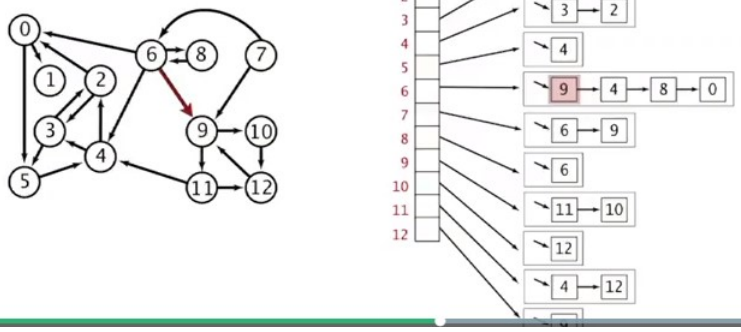
read digraph from input stream

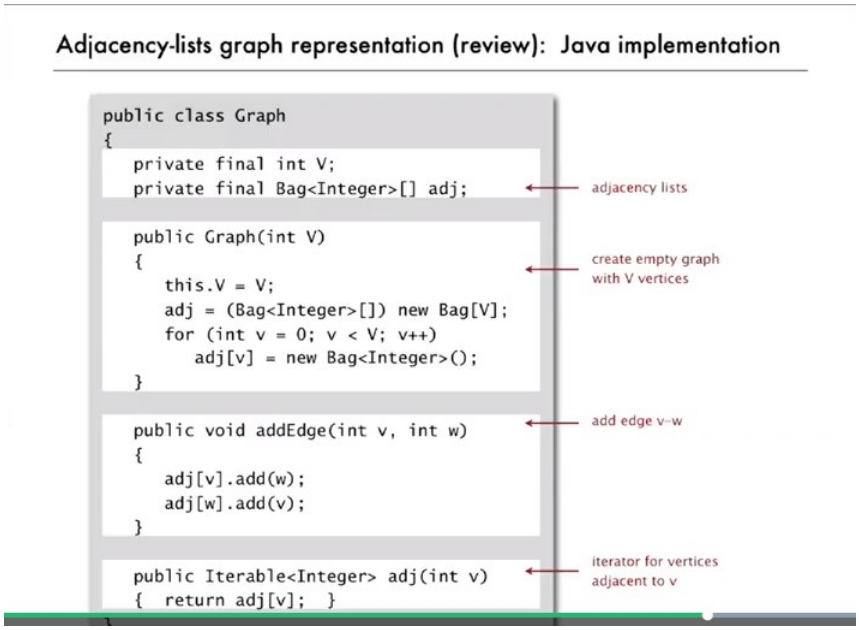
print out each edge (once)

- Representations

## Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.



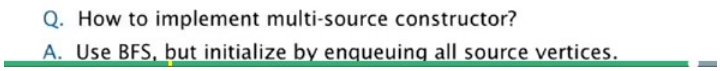



**Reachability**
**Depth First Search**: Exactly the same (because when we implemented graph, it treated each edge as two, going out and coming in) - so the two algorithms are identical
**Applications**

- Program Control Flow Analysis - Every program can be seen as a digraph
  - Dead Code elimination
  - Infinite Loop Detection
  - Garbage Collection (Mark-Sweep Algorithm)
    - Every data structure is a digraph
    - Objects are vertices, edges are references
    - uses 1 extra bit per object
    - Unreachable objects get collected
- Robert Tarjan showed that DFS can solve a lot of complex Graph problems in linear time

**Breadth First Search**

- Computes the shortest path
- Multiple source shortest paths: Given a digraph and a set of source vertices, find the shortest path to each other vertex from any of the source vertices




- Web Crawler: Each page is a vertex, link is an edge
  - Why not DFS?
  - Bare-bones web crawler

**Bare-bones web crawler: Java implementation**

```
Queue<String> queue = new Queue<String>();
SET<String> discovered = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
discovered.add(root);

while (!queue.isEmpty())
{
   String v = queue.dequeue();
   StdOut.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!discovered.contains(w))
      {
         discovered.add(w);
         queue.enqueue(w);
      }
   }
}
```

queue of websites to crawl
set of discovered websites

start crawling from root website

read in raw html from next website in queue

use regular expression to find all URLs in website of form http://xxx.yyy.zzz [crude pattern misses relative URLs]

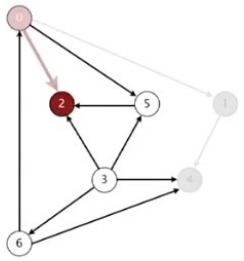
if undiscovered, mark it as discovered and put on queue

**Topological Sort**

- Precedence Scheduling: order in which tasks need to be scheduled when there are precedence constraints

**Topological sort demo**

- Run depth-first search.
- Return vertices in reverse postorder.




postorder
4 1

- DAG - Directed Acyclic Graph -- task scheduling graphs cannot have cycles
- if no directed cycle, DFS based algorithm finds a topological order
  - if it has a cycle, you can't sort topologically
- Java compiler uses this to detect cycles (A extends B, B extends C, C extends A)
- MS Excel Formula

**Strong Components**

- Strongly connected - if there is a directed path v -> w and w-> v
- strongly connected to is an equivalence relation (refl, symm, trans)
- Application
  - Food Web Graph: A strong component is species that eat each other
  - Software Modules: Vertices are modules, edges are dependencies. A strong component is a collection that is mutually using each other and should be packaged together

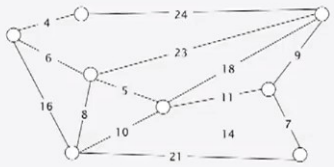
1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

**Minimum Spanning Tree**

## Minimum spanning tree

Given. Undirected graph $G$ with positive edge weights (connected).
Def. A spanning tree of $G$ is a subgraph $T$ that is both a tree (connected and acyclic) and spanning (includes all of the vertices).
Goal. Find a min weight spanning tree.




graph G

## Applications

MST is fundamental problem with diverse applications.

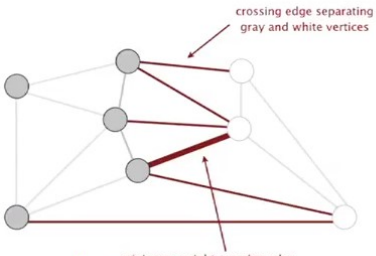
- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

**Greedy Algorithm**

## Cut property

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.




## Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

## Greedy MST algorithm: efficient implementations

Proposition. The greedy algorithm computes the MST.

Efficient implementations. Choose cut? Find min-weight edge?
Ex 1. Kruskal's algorithm. [stay tuned]
Ex 2. Prim's algorithm. [stay tuned]
Ex 3. Borüvka's algorithm.

## Weighted edge API

Edge abstraction needed for weighted edges.

| public class Edge implements Comparable<Edge> | |
|---|---|
| Edge(int v, int w, double weight) | *create a weighted edge v-w* |
| int either() | *either endpoint* |
| int other(int v) | *the endpoint that's not v* |
| int compareTo(Edge that) | *compare this edge to that edge* |
| double weight() | *the weight* |
| String toString() | *string representation* |

## Edge-weighted graph API

| public class EdgeWeightedGraph | |
|---|---|
| EdgeWeightedGraph(int V) | *create an empty graph with V vertices* |
| EdgeWeightedGraph(In in) | *create a graph from input stream* |
| void addEdge(Edge e) | *add weighted edge e to this graph* |
| Iterable<Edge> adj(int v) | *edges incident to v* |
| Iterable<Edge> edges() | *all edges in this graph* |
| int V() | *number of vertices* |
| int E() | *number of edges* |
| String toString() | *string representation* |

## Edge-weighted graph: adjacency-lists implementation

```
public class EdgeWeightedGraph
{
   private final int V;
   private final Bag<Edge>[] adj;

   public EdgeWeightedGraph(int V)
   {
      this.V = V;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj(int v)
   { return adj[v]; }
}
```

same as Graph, but adjacency lists of Edges instead of integers

constructor

add edge to both adjacency lists

### Minimum spanning tree API

Q. How to represent the MST?

```
public class MST

                  MST(EdgeWeightedGraph G)      constructor
Iterable<Edge>    edges()                       edges in MST
        double    weight()                      weight of MST
```

**Kruskal's Algorithm**

### Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree $T$ unless doing so would create a cycle.

Use Union Find to check whether a cycle gets created by adding an edge to the tree. If the edge is connecting vertices in the same connected component, it would create a cycle

### Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST(EdgeWeightedGraph G)
   {
      MinPQ<Edge> pq = new MinPQ<Edge>();
      for (Edge e : G.edges())
         pq.insert(e);

      UF uf = new UF(G.V());
      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();
         int v = e.either(), w = e.other(v);
         if (!uf.connected(v, w))
         {
            uf.union(v, w);
            mst.enqueue(e);
         }
      }
   }

   public Iterable<Edge> edges()
   {  return mst;  }
}
```

build priority queue

greedily add edges to MST

edge v–w does not create cycle

merge sets

add edge to MST

Complexity: ELogE

### Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

| operation | frequency | time per op |
|---|---|---|
| build pq | 1 | E |
| delete-min | E | log E |
| union | V | $\log^* V$ † |
| connected | E | $\log^* V$ † |

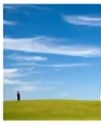
### Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

MST - Does it have an algorithm that gives linear time performance - open
not proven that it doesn't exist
best theoretical is really close to linear (E)

**Shortest Path Problems**

### Shortest path applications



- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

### Shortest path variants

Which vertices?

- Source-sink: from one vertex to another.
- Single source: from one vertex to every other.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

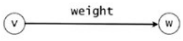
- Nonnegative weights.
- Arbitrary weights.
- Euclidean weights.

Cycles?

- No directed cycles.
- No "negative cycles."

### Weighted directed edge API

| public class DirectedEdge | |
|---|---|
| DirectedEdge(int v, int w, double weight) | *weighted edge v→w* |
| int from() | *vertex v* |
| int to() | *vertex w* |
| double weight() | *weight of this edge* |
| String toString() | *string representation* |




Idiom for processing an edge e: `int v = e.from(), w = e.to();`

**Edge Relaxation - take an edge into account**

- Replace an old edge if you get a better one, if not, don't change anything

### Shortest-paths optimality conditions

Proposition. Let $G$ be an edge-weighted digraph.
Then `distTo[]` are the shortest path distances from `s` iff:

- `distTo[s]` = 0.
- For each vertex `v`, `distTo[v]` is the length of some path from `s` to `v`.
- For each edge `e` = `v→w`, `distTo[w]` ≤ `distTo[v]` + `e.weight()`.

### Generic shortest-paths algorithm

**Generic algorithm (to compute SPT from s)**

**Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.**

**Repeat until optimality conditions are satisfied:**

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?
Ex 1. Dijkstra's algorithm (nonnegative weights).
Ex 2. Topological sort algorithm (no directed cycles).
Ex 3. Bellman-Ford algorithm (no negative cycles).

**Dijkstra's Algorithm**

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: $V$ insert, $V$ delete-min, $E$ decrease-key.

| PQ implementation | insert | delete-min | decrease-key | total |
|---|---|---|---|---|
| unordered array | 1 | V | 1 | $V^2$ |
| binary heap | log V | log V | log V | E log V |
| d-way heap (Johnson 1975) | $d \log_d V$ | $d \log_d V$ | $\log_d V$ | $E \log_{E/V} V$ |
| Fibonacci heap (Fredman–Tarjan 1984) | 1 † | log V † | 1 † | E + V log V |

† amortized

**Dijkstra's algorithm can work with cycles but not with negative weights**

**Longest path in a DAG**: Run Topological sort algorithm with weights negated
-> Useful in **parallel job scheduling:**

- Bunch of jobs with precedence and duration
- Critical Path Method - find the longest path from the source to schedule each job (
  - precedence is represented by 0 weighted edges
  - duration is an edge from start to finish
  - source and sink are connected to start and end of each job and the weights for those are 0

**Negative Weights**

- Dijkstra algorithm doesn't work
  - You can hit the negative cycle as many times as you want and keep making your path shorter
- A SPT exists if no negative cycles and vice versa

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.

Bellman-Ford algorithm

**Bellman–Ford algorithm**

**Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.**

**Repeat V times:**
**– Relax each edge.**

**Bellman ford can work with negative edges but not negative cycles**

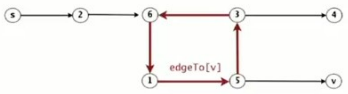
### Single source shortest-paths implementation: cost summary

| algorithm | restriction | typical case | worst case | extra space |
|---|---|---|---|---|
| topological sort | no directed cycles | E + V | E + V | V |
| Dijkstra (binary heap) | no negative weights | E log V | E log V | V |
| Bellman-Ford | no negative cycles | E V | E V | V |
| Bellman-Ford (queue-based) | | E + V | E V | V |

Remark 1. Directed cycles make the problem harder.
Remark 2. Negative weights make the problem harder.
Remark 3. Negative cycles makes the problem intractable.

### Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.




Proposition. If any vertex v is updated in phase V, there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

### Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

| | USD | EUR | GBP | CHF | CAD |
|---|---|---|---|---|---|
| USD | 1 | 0.741 | 0.657 | 1.061 | 1.011 |
| EUR | 1.350 | 1 | 0.888 | 1.433 | 1.366 |
| GBP | 1.521 | 1.126 | 1 | 1.614 | 1.538 |
| CHF | 0.943 | 0.698 | 0.620 | 1 | 0.953 |
| CAD | 0.995 | 0.732 | 0.650 | 1.049 | 1 |

### Shortest paths summary

Dijkstra's algorithm.
- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.
- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.
- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

**MinCut Graph (weighted digraph, source - s, target - t) - Didn't really maxflow, mincut stuff**

- An st cut is a partition of vertices in two disjoint sets, with s in one set A and t in the other set B
  - s is designated as source and t as target in this graph
  - capacity = sum of edges from A -> B
  - Min Cut - Find a cut with minimum capacity

**Maxflow Problem (weighted digraph, source - s, target - t)**

- An st-flow is an assignment of values to the edges such that
  - capacity constraint: edge's flow <= edge's capacity
  - Local equilibrium: inflow = outflow at every vertex
  - source - everything has only outflow, target has only inflow

They have the same solution (such problems are called dual problems)

**Ford Fulkerson Algorithm**

## Ford-Fulkerson algorithm

**Ford–Fulkerson algorithm**

**Start with 0 flow.**
**While there exists an augmenting path:**
- **find an augmenting path**
- **compute bottleneck capacity**
- **increase flow on that path by bottleneck capacity**

## Relationship between flows and cuts

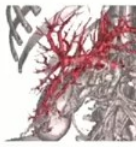
Def. The net flow across a cut $(A, B)$ is the sum of the flows on its edges from $A$ to $B$ minus the sum of the flows on its edges from from $B$ to $A$.

Flow-value lemma. Let $f$ be any flow and let $(A, B)$ be any cut. Then, the net flow across $(A, B)$ equals the value of $f$.

## Maxflow and mincut applications

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Many, many, more.



liver and hepatic vascularizi

---

**Strings**

- Java char is a 16 bit unsigned integer (supports original 16 bit unicode)
- charAt, substring n Java are constant time, concat is linear

### The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).
Underlying implementation. Resizing `char[]` array and length.

| | String | | StringBuilder | |
|---|---|---|---|---|
| operation | guarantee | extra space | guarantee | extra space |
| length() | 1 | 1 | 1 | 1 |
| charAt() | 1 | 1 | 1 | 1 |
| substring() | 1 | 1 | N | N |
| concat() | N | N | 1 * | 1 * |

* amortized

- 
- Radix = number of possible values in a string
- ascii - raidix is 128 (7 bits)
- extended ascii - 256, unicode - 65526 (16 bit)

#### Key Index Counting

- Sorting is optimal at NlgN if depends on compares
- Should have low cardinality
- first pass to count frequencies with key as index
- go through the count array and make cumulates by adding the freq from prev key
- use cumulates array to place the values in the right place in an aux array
- assign aux to the original array
- Complexity: N+R (practically Linear time)
- It is stable

#### LSD (Least Significant Digit) Radix Sort

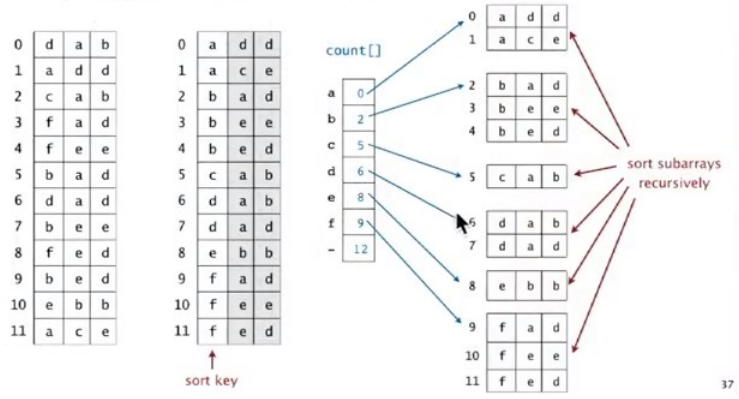
- Strings are all of same length
- Sort using the least significant char (using key index counting) and then move to more significant characters

#### MSD (Most Significant Digit) Radix Sort

### Most-significant-digit-first string sort

MSD string (radix) sort.

- Partition array into $R$ pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).



## MSD string sort: Java implementation

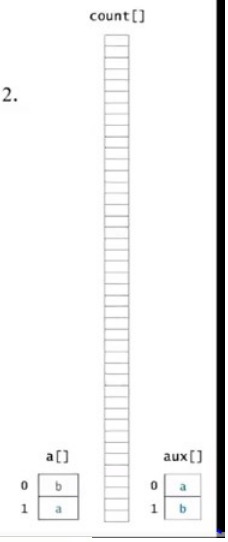
```
public static void sort(String[] a)
{
   aux = new String[a.length];
   sort(a, aux, 0, a.length-1, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int[] count = new int[R+2];                          key-indexed counting
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   for (int r = 0; r < R; r++)                          sort R subarrays recursively
      sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

## MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

Observation 2. Huge number of small subarrays because of recursion.




## Cutoff to insertion sort

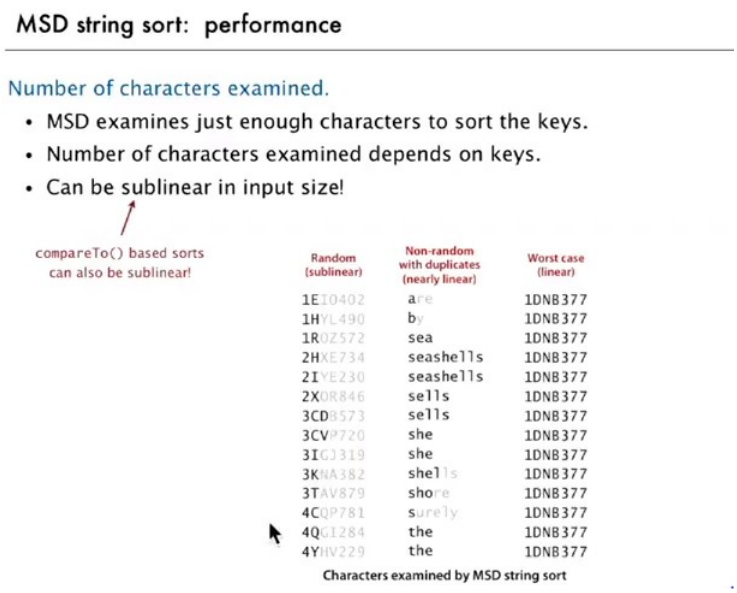
Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at $d^{th}$ character.
- Implement `less()` so that it compares starting at $d^{th}$ character.

```
public static void sort(String[] a, int lo, int hi, int d)
{
   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

in Java, forming and comparing substrings is faster than directly comparing chars with charAt()

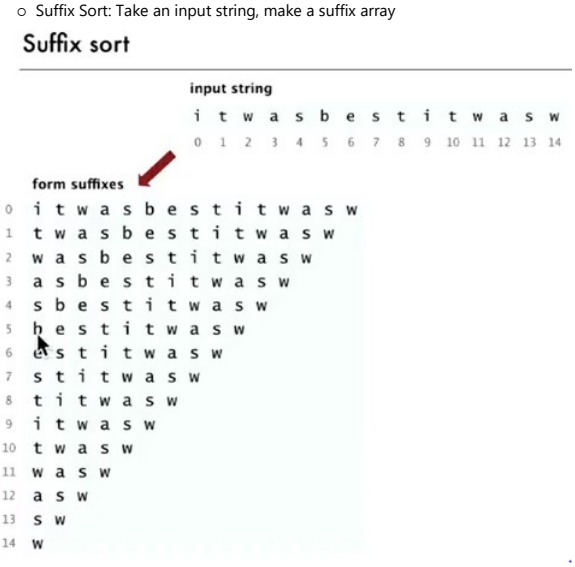
42




**3-way Radix Quicksort**

**Suffix Arrays**

- Keyword-in-context Search: Given a huge text, pre-process it to enable fast sub-string search
  - Suffix Sort: Take an input string, make a suffix array




  - Sort suffixes
  - Then use binary search
- Longest repeating sub-string
  - Bioinformatics, cryptanalysis, Data Compression, visualize repetition in music
  - Brute Force
    - DN^2: Try all pairs of i, j and see how far they match
  - Suffix Sort
    - Form suffixes, sort them
    - Check adjacent substring
    - If we have too many repetitions (like copy the same text twice) - worst case - becomes quadratic
  - Suffix Sorting in Linearithmic time - Manber Myers Algorithm
    - Kind of like MSD -- doubles the number of characters you look at in each iteration
    - Gave up on explanation...

**R-way Tries**

## Tries

Tries. [from retrieval, but pronounced "try"]

- For now, store *characters* in nodes (not keys).
- Each node has $R$ children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

for now, we do not draw null links

## R-way trie: Java implementation

```
public class TrieST<Value>
{
   private static final int R = 256;     ← extended ASCII
   private Node root = new Node();

   private static class Node
   {  /* see previous slide */  }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);  }

   private Node put(Node x, String key, Value val, int d)
   {
      if (x == null) x = new Node();
      if (d == key.length()) { x.val = val; return x; }
      char c = key.charAt(d);
      x.next[c] = put(x.next[c], key, val, d+1);
      return x;
   }
```

## R-way trie: Java implementation (continued)

```
   ⋮
   public boolean contains(String key)
   {  return get(key) != null;  }

   public Value get(String key)
   {
      Node x = get(root, key, 0);
      if (x == null) return null;
      return (Value) x.val;     ← cast needed
   }

   private Node get(Node x, String key, int d)
   {
      if (x == null) return null;
      if (d == key.length()) return x;
      char c = key.charAt(d);
      return get(x.next[c], key, d+1);
   }

}
```

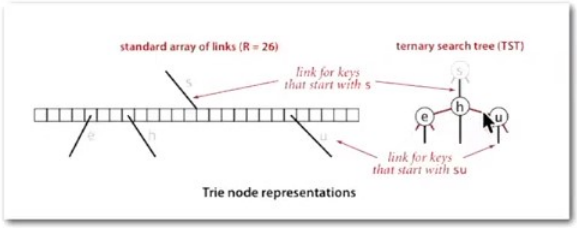
Takes up too much space

## TST representation in Java

A TST node is five fields:

- A value.
- A character $c$.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```
private class Node
{
   private Value val;
   private char c;
   private Node left, mid, right;
}
```




**Substring Search**

- MxN - brute force
- KMP - Knuth-Morris-Pratt
  - Deterministic Finite State Automaton
  - went above the head for the non trivial scenario
- Boyer Moore
  - Practically N/M
  - Worst case can be M*N
- Rabin Karp
  - fingerprint search
  - modular hashing
    - take remainder using a big prime
    - Horner's method to calculate a polynomial in linear time
    - Hash for t is know, how can we use it to get hash for t+1
  - Monte Carlo Version: if hash matches, return
    - Probability of collision is 1/Q (Q is the prime number)
    - Always runs in linear
    - extremely likely to get the right answer
  - Las Vegas version: if hashes match, check for equality
    - Always gets the right answer
    - Worst case can be M*N

## Substring search cost summary

Cost of searching for an $M$-character pattern in an $N$-character text.

| algorithm | version | operation count: guarantee | operation count: typical | backup in input? | correct? | extra space |
|---|---|---|---|---|---|---|
| brute force | — | $MN$ | $1.1\,N$ | *yes* | *yes* | 1 |
| Knuth-Morris-Pratt | *full DFA (Algorithm 5.6)* | $2N$ | $1.1\,N$ | *no* | *yes* | $MR$ |
| | *mismatch transitions only* | $3N$ | $1.1\,N$ | *no* | *yes* | $M$ |
| Boyer-Moore | *full algorithm* | $3N$ | $N / M$ | *yes* | *yes* | $R$ |
| | *mismatched char heuristic only (Algorithm 5.7)* | $MN$ | $N / M$ | *yes* | *yes* | $R$ |
| Rabin-Karp† | *Monte Carlo (Algorithm 5.8)* | $7\,N$ | $7\,N$ | *no* | *yes*† | 1 |
| | *Las Vegas* | $7\,N$† | $7\,N$ | *yes* | *yes* | 1 |

† probabilistic guarantee, with uniform hash function

# Design of Data Intensive Applications

**Definition**
We call an application data-intensive if data is its primary challenge—the quantity of data, the complexity of data, or the speed at which it is changing—asopposed to compute-intensive, where CPU cycles are the bottleneck.

**Common system requirements (data intensive apps)**

Store data so that they, or another application, can find it again later (databases)
Remember the result of an expensive operation, to speed up reads (caches)
Allow users to search data by keyword or filter it in various ways (search indexes)
Send a message to another process, to be handled asynchronously (stream processing)
Periodically crunch a large amount of accumulated data (batch processing)

**Reliability**

Reliable: Good behavior in face of adversity (hardware/software faults, human error)
Continuing to work correctly, even when things go wrong. Things that go wrong - faults. systems that can cope with that are fault tolerant/resilient. Goal should be to avoid faults leading to failure

Fault vs Failure - Fault is deviation from expected behavior, failure is stopped service

*Hardware Faults*
- Traditionally components had redundancy, main component fails, the backup takes over
- Now trend is towards software fault tolerance as hardware tolerance increases cost and doesn't align with the model of elastic capacity.

*Software Errors*
Systematic errors due to bugs. Usually applications make assumptions and at some point some of them get broken, perhaps due to unusual circumstances that weren't thought through.

*Human Errors* - humans are known to be unreliable, they are the leading cause of outages in large internet services followed by hardware faults

**Scalability**
Scalability: Ability to cope up with increased load
Define a parameter to measure load (load parameter - can be request per second, ratio of reads to writes, concurrent users, hit rate on a cache, essentially dependent on your system)

Maintainable: Maintaining current behavior of the system over time and adjusting for new requirements

**LATENCY AND RESPONSE TIME**

*Latency* and *response time* are often used synonymously, but they are not the same. The response time is what the client sees: besides the actual time to process the request (the *service time*), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled—during which it is *latent*, awaiting service [17].

Usually it is better to use *percentiles*. If you take your list of response times and sort it from fastest to slowest, then the *median* is the halfway point: for example, if your median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

Amazon has also observed that a 100 ms increase in response time reduces sales by 1% [20], and others report that a 1-second slowdown reduces a customer satisfaction metric by 16%
Distributing load across multiple machines is also known as a *shared-nothing* architecture.

**Maintainability**
Three sub principles

*Operability*

Make it easy for operations teams to keep the system running smoothly.

*Simplicity*

Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system. (Note this is not the same as simplicity of the user interface.)
Making a system simpler does not necessarily mean reducing its functionality; it can also mean removing *accidental* complexity. Moseley and Marks [32] define complexity as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.

> One of the best tools we have for removing accidental complexity is *abstraction*. A good abstraction can hide a great deal of implementation detail behind a clean, simple-to-understand façade

*Evolvability*

Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as *extensibility*, *modifiability*, or *plasticity*.

Data Models and Query Language

Data models are perhaps the most important part of developing software, because they have such a profound effect: not only on how the software is written, but also on how we *think about the problem* that we are solving.

Most applications are built by layering one data model on top of another. For each layer, the key question is: how is it *represented* in terms of the next-lower layer

Relational Model

The best-known data model today is probably that of SQL, based on the relational model proposed by Edgar Codd in 1970 [1]: data is organized into *relations* (called *tables* in SQL), where each relation is an unordered collection of *tuples* (*rows* in SQL).

NoSQL

There are several driving forces behind the adoption of NoSQL databases, including:

A need for greater scalability than relational databases can easily achieve, including very large datasets or very high write throughput

A widespread preference for free and open source software over commercial database products

Specialized query operations that are not well supported by the relational model

Frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model [5]

For a data structure like a résumé, which is mostly a self-contained *document*, a JSON representation can be quite appropriate

The advantage of using an ID is that because it has no meaning to humans, it never needs to change: the ID can remain the same, even if the information it identifies changes. Anything that is meaningful to humans may need to change sometime in the future—and if that information is duplicated, all the redundant copies need to be updated. That incurs write overheads, and risks inconsistencies (where some copies of the information are updated but others aren't). Removing such duplication is the key idea behind *normalization* in databases.

Document Databases: Store data as json documents

Document databases reverted back to the hierarchical model in one aspect: storing nested records (one-to-many relationships, like `positions`, `education`, and `contact_info` in Figure 2-1) within their parent record rather than in a separate table.
The main arguments in favor of the document data model are schema flexibility, better performance due to locality, and that for some applications it is closer to the data structures used by the application. The relational model counters by providing better support for joins, and many-to-one and many-to-many relationships

For highly interconnected data, the document model is awkward, the relational model is acceptable, and graph models (see "Graph-Like Data Models") are the most natural

A more accurate term is *schema-on-read* (the structure of the data is implicit, and only interpreted when the data is read), in contrast with *schema-on-write* (the traditional approach of relational databases, where the schema is explicit and the database ensures all written data conforms to it)
Schema-on-read is similar to dynamic (runtime) type checking in programming languages, whereas schema-on-write is similar to static (compile-time) type checking

The schema-on-read approach is advantageous if the items in the collection don't all have the same structure for some reason (i.e., the data is heterogeneous)—for example, because:

There are many different types of objects, and it is not practical to put each type of object in its own table.

The structure of the data is determined by external systems over which you have no control and which may change at any time.

Data Locality: A document is usually stored as a single continuous string, encoded as JSON, XML, or a binary variant thereof (such as MongoDB's BSON). If your application often needs to access the entire document (for example, to render it on a web page), there is a performance advantage to this *storage locality*. If data is split across multiple tables, like in Figure 2-1, multiple index lookups are required to retrieve it all, which may require more disk seeks and take more time.

In general document dbs are good if the documents built are generally queried and updated as a whole most of the time

An imperative language tells the computer to perform certain operations in a certain order.
In a declarative query language, like SQL or relational algebra, you just specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed (e.g., sorted, grouped, and aggregated)—but not *how* to achieve that goal. It is up to the database system's query optimizer to decide which indexes and which join methods to use, and in which order to execute various parts of the query.

Imperative code is very hard to parallelize across multiple cores and multiple machines, because it specifies instructions that must be performed in a particular order. Declarative languages have a better chance of getting faster in parallel execution because they specify only the pattern of the results, not the algorithm that is used to determine the results.

*Pure* functions: means they only use the data that is passed to them as input, they cannot perform additional database queries, and they must not have any side effects

as the connections within your data become more complex, it becomes more natural to start modeling your data as a graph

*recursive common table expressions - SQL equivalent for graph like queries - explore*

Semantic Web: Idea that asks websites to publish information as machine-readable data for computers to read

**Storage and Retrieval**

*log* in more general sense is an append-only sequence of records.
An index is an *additional* structure that is derived from the primary data to help make search fast based on certain keys
well-chosen indexes speed up read queries, but every index slows down writes.
Hash index - stores the key along with its byte offset or some info to get to it faster
deletion record is called tombstone

An append-only log seems wasteful at first glance: why don't you update the file in place, overwriting the old value with the new value? But an append-only design turns out to be good for several reasons:

Appending and segment merging are sequential write operations, which are generally much faster than random writes, especially on magnetic spinning-disk hard drives

Concurrency and crash recovery are much simpler if segment files are append-only or immutable. For example, you don't have to worry about the case where a crash happened while a value was being overwritten, leaving you with a file containing part of the old and part of the new value spliced together.

SSTable - SortedStringTable
Log-Structured-Merge Tree
The most widely used indexing structure is quite different: the *B-tree*.
You can think of overwriting a page on disk as an actual hardware operation. On a magnetic hard drive, this means moving the disk head to the right place, waiting for the right position on the spinning platter to come around, and then overwriting the appropriate sector with new data

In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a *write-ahead log*

One write to the database resulting in multiple writes to the disk over thecourse of the database's lifetime—is known as write amplification. It is of particular concernon SSDs, which can only overwrite blocks a limited number of times before wearing out.

In some situations, the extra hop from the index to the heap file is too much of a performance penalty for reads, so it can be desirable to store the indexed row directly within an index. This is known as a *clustered index*.
A compromise between a clustered index (storing all row data within the index) and a nonclusteredindex (storing only references to the data within the index) is known as a covering index orindex with included columns, which stores some of a table's columns within the index

With both magnetic disks and SSDs,data on disk needs to be laid out carefully if you want good performance on reads and writes.However, we tolerate this awkwardness because disks have two significant advantages: they aredurable (their contents are not lost if the power is turned off), and they have a lower cost pergigabyte than RAM.

Transaction processing just means allowing clients to make low-latency reads andwrites—as opposed to batch processing jobs, which only run periodically (for example, once perday).
An application typically looks up a small number of records bysome key, using an index. Records are inserted or updated based on the user's input. Because theseapplications are interactive, the access pattern became known as online transaction processing(OLTP).

*Table 3-1. Comparing characteristics of transaction processing versus analytic systems*

| Property | Transaction processing systems (OLTP) | Analytic systems (OLAP) |
|---|---|---|
| Main read pattern | Small number of records per query, fetched by key | Aggregate over large number of records |
| Main write pattern | Random-access, low-latency writes from user input | Bulk import (ETL) or event stream |
| Primarily used by | End user/customer, via web application | Internal analyst, for decision support |
| What data represents | Latest state of data (current point in time) | History of events that happened over time |
| Dataset size | Gigabytes to terabytes | Terabytes to petabytes |

A data warehouse, by contrast, is a separate database that analysts can query to their hearts'content, without affecting OLTP operations[48].The data warehouse contains a read-only copy of the data in all the various OLTP systems in thecompany.
Data is extracted from OLTP databases (using either a periodic data dump or a continuousstream of updates), transformed into an analysis-friendly schema, cleaned up, and then loaded intothe data warehouse. This process of getting data into the warehouse is known asExtract–Transform–Load (ETL)
On the surface, a data warehouse and a relational OLTP database look similar, because they both have a SQL query interface. However, the internals of the systems can look quite different, because they are optimized for very different query patterns.
Many data warehouses are used in a fairly formulaic style, known as a *star schema* (also known as *dimensional modeling. A variation of this template is known as the snowflake schema, where dimensions are further brokendown into subdimensions*

*In most OLTP databases, storage is laid out in a row-oriented fashion: all the values from one rowof a table are stored next to each other. Document databases are similar: an entire document istypically stored as one contiguous sequence of bytes*

*The idea behind column-oriented storage is simple: don't store all the values from one rowtogether, but store all the values from each column together instead. column-oriented storage often lends itself very well to compression.*
*One technique that is particularly effective in data warehousesis bitmap encoding*

*Bitmap indexes work with columns wiht low cardinality. For each distinct column value they can create a bit for every row if that row has that particular value or not. Where conditions can be executed very fast by using bit operations on them*
*Different queries benefit from different sort orders, so why not store the same data sorted inseveral different ways? Data needs to be replicated to multiple machines anyway, so that you don'tlose data if one machine fails. You might as well store that redundant data sorted in differentways so that when you're processing a query, you can use the version that best fits the querypattern.*

*The difference is that a materialized view is an actual copy of the query results, written to disk, whereas a virtual view is just a shortcut for writing queries.*

**Encoding and Evolution**

The translation from the in-memory representation to a byte sequence is called encoding (also known as serialization or marshalling), and the reverse is called decoding (parsing, deserialization,unmarshalling)

Many programming languages come with built-in support for encoding in-memory objects into bytesequences. Often many disadvantages like tied up with the language, poor support for versioning, performance is bad

Standardized encodings - json, xml
JSON's popularity is mainly due to itsbuilt-in support in web browsers. Many limitations there too

- JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision.

Binary Encodings - mainly used internally as you don't need other organizations to agree to it

JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a profusion of binary encodings for JSON (MessagePack, BSON,BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example).

**Modes of Dataflow**
***data outlives code***

***Databases***

- processes reading to/writing from DBs

***RPC/Rest Calls***
This approach is often used to decompose a large application into smaller services by area of functionality, such that one service makes a request to another when it requires some functionality or data from that other service. This way of building applications has traditionally been called a service-oriented architecture (SOA), more recently refined and rebranded as microservices architecture

A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*.
There are two popular approaches to web services: REST and SOAP. They are almost diametricallyopposed in terms of philosophy, and often the subject of heated debate among their respectiveproponents

REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP [34, 35]. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. REST has been gaining popularity compared to SOAP, at least in the context of cross-organizational service integration [36], and is often associated with microservices

The API of a SOAP web service is described using an XML-based language called the Web Services Description Language, or WSDL

A definition format such as OpenAPI, also known as Swagger[40], can be used to describe RESTful APIs and producedocumentation.

The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called *location transparency*).

RPCs are never as reliable as local calls, so you need to learn and design the failure scenarios, possible retires, but then retries need to be safe if part of your request was actually done successfully though you received a failure

***Message Passing Dataflow***

********** **Distributed Data** **********

---

Why?

- Scalability: increased load can be handled with more machines
- High Availability: one machine goes down, others can work
- Latency: geographically placing data closer to users

Different Options:

- Shared Memory Architecture: Scaling up, vertical scaling)
  - Buy a more powerful machine (one CPU and RAM sits on top of the storage), prices increase more than linearly with capacity
  - They may offer some level of fault tolerance as disks and CPUs are allowed to hot swapped
  - Definitely no geographical latency capability
- Shared Disk Architecture
  - CPU and RAMs are independent but share the same array of disks
  - Contention and Overhead of locking limits its scalability
- Shared Nothing Architecture: (Scaling out, horizontal scaling)
  - many machines (called nodes) with their own CPU, RAM and disks
  - Coordination is done via software
  - these need most attention from developers as there are many constraints and trade offs introduced

**Distributing Data Across Nodes**

- Replication: Keeping a copy of data in different nodes (potentially in different locations)
  - Can improve performance
  - Redundancy
  - Fault tolerance
- Partitioning:
  - Splitting data into smaller subsets called partitions and different partitions can be assigned to different nodes (called sharding)

**Replication**
**Why**: Fault tolerance, scalability (higher number of read nodes), latency (geographical co-location).
Each node that has a copy is called replica
No issues if data is static, but handling changes is the tricky part. there are 3 algorithms for that

Single Leader Replication (Active/Passive or Master/slave):

- One replica is designated as leader (master/primary), when clients write, they must send their request to the leader who writes the new data to its local storage
- Leader publishes the data change to replication log/change stream
- Followers(slaves, secondaries) - write data to their local storage in the same order as they were processed by the leader
- Reads are sent to all, writes only to leader

*Synchronous vs Asynchronous Replication*
Leader waits, leader proceeds without waiting

Synchronous:

- Any one node going down can bring the system to a halt
    - Semi - synchronous - when leader can proceed if one of the follower is good

Asynchronous:

- Fully asynchronous system means that writes will be lost even if it was confirmed to the client in a scenario when it couldn't replicate in time and the master node goes down
- Advantage is it is always available, w/o waiting for any other nodes

Setting up a new follower - take a copy of master for a particular snapshot, then request all changes after that and process. Also each follower holds a log of changes received from master as well as the last one processed.

Leader failure - Failover means electing a new leader and route all client traffic to the new leader

**Replication Methods**

- Statement based replication: maintain all the statements run on the leader
    - non deterministic parts will pose challenges, like now()
    - sequence needs to be maintained
    - this method can have complications so not preferred these days
- Write Ahead Logs
    - log is an append only sequence of bytes containing all writes to the database
    - Because this deals with really low level (bytes) it may differ based on versions, so upgrades can cause issues
- Logical Log Based Replication
    - logs describing DB action at the level of a row
        - inserted row - all the data
        - deleted row - primary key of deleted data
        - updated - key and relevant columns
- Trigger/SP based
    - A trigger lets you attach app code with data changes

**Issues caused by Replication Lag**

- Reading your own writes: you create a new item on a webpage and then webpage needs to show you the newly created item, writes go to master, read can go anywhere and you may not find your data there. Possible fixes
    - For stuff that user changed, route reads also to leader. May not work if everything is editable by the user, then everything will be routed to the leader
    - Have a timestamp of the most recent write, ensure the node for read has caught up till that timestamp, else use another one
- Monotonic Reads: Guarantee that a user will not go back in time (saw some data, then a second later, it goes off, can happen with different levels of catch up for different nodes
    - Possible fix is to tie a user to a node, but in case of node failure, rerouting will spoil it (but mostly momentarily)
- Consistent Prefix Reads: Causality should not be reversed (linked events should appear in right order)

**Multi Leader Replication:**

- Natural extension of Single Leader approach, more than one node is allowed to accept writes
- More useful if you deal with multiple data centers, then you can have one leader per datacenter. Within the data center, you can replicate like single leader and then leaders give each other their changes
- **Offline Operations:** Your device accepts writes even when you are offline, the local storage acts as the leader
- Will need to deal with conflict resolution (single leader will block everyone else, so no such issue)
    - One option is to route changes to one record to one leader only or some other mechanism of affinity, it will still break down when the key attribute deciding affinity changes
    - many possible ways to deal with it
        - assign an Id to all the writes and higher id wins
        - report conflict and store all the required information so that we can use it to decide what is right (may be using app code)

**Leaderless Replication:**

- All nodes get read/write requests. Stale data is identified based on the values from other nodes and the stale guy is brought up to speed (called Read Repair)
- Anti Entropy - background process that keeps fixing staleness
- A quorum is configured in such systems, if there are n nodes, we are good if we received w confirmations, and good for reads when we get r responses, where w + r > n. Requests for reads/writes are sent to everyone but waiting is done until quorum is achieved

---

# Interview Questions: Core Java

https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-1
https://howtodoinjava.com/core-java/string-class/interview-stuff-about-string-class-in-java/
https://www.edureka.co/blog/interview-questions/java-interview-questions/
https://www.journaldev.com/java-interview-questions
https://www.journaldev.com/2366/core-java-interview-questions-and-answers
https://beginnersbook.com/2013/05/java-interview-questions/

---

# Interview Questions: Microservices

https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/

---

# Interview Questions: Web Servers

https://howtodoinjava.com/server/tomcat/a-birds-eye-view-on-how-web-servers-work/

---

# Spark Classroom Training - Day 1/4

Password -- hdp or 123

The Shared location:- \\192.168.1.101\d\Spark Training

3 modules - JP participants

1) Hadoop Foundation Module
2) Spark Storm Kafka Module
3) Admin - JPMIS / GTI

=======================> Topics covered in the Hadoop Foundation Module

1) What is Big Data Very Basics
2) What is Hadoop Very Basics
3) Pseudo Cluster Setup - Hadoop 2.7.2 Full
4) HDFS Basics Full
5) Map Reduce No
6) Hive No
7) Pig No
8) Sqoop No
9) NoSQL No
10) HBase No

What we will cover in this module

1) What and why of Apache Spark
2) Spark Core - Transformations + Actions
3) Run Spark Code - REPL [ Interactive ], Batch [ spark-submit ]
Languages - Scala, Python, Java
4) Notebooks - Jupyter, Zepplin, Databricks
5) Spark SQL - sqlContext.sql, DataFrames
6) Spark Streaming
7) Apache Storm
8) Kafka

House-keeping

1) Hard Start - 10.10
2) First Break - 11.45 for 15 min
3) Lunch Break - 1.30 for 45 min
4) Tea Break - 3.45
Wrap up by 5.45

Every day - spend approx - 1 hour of pre-reading for the next day

=============================================>

JPMIS -- Infra component - CDH - Multi tenant --> deploy the applications

Pluralsight --> Each of the Eco-System -->

Venkat -

ETL -->
Data Warehouse + ETL tools

HDFS ==> Hive, SparkSQL - Talend, Pentaho

================================>

Big Data? 3 v's of Big Data

1) Volume --> 8 PB [ JPMIS ]
  --> 13 PB Data Lake - Citi
  --> 3 PB Data Lake

2) Variety -->
Structured - Transactional Data
Semi-Structured - Textual data but not as per a schema [ logs, emails, comments, blogs ]
UnStructured -
Videos
Audio - Voice records of customer calls.
Image - Scanned cheque, Pre-Trading Docs

3) Velocity --> Real time streaming data. --> speed of the data arrival

Use Cases of Big Data

1) https://wiki.apache.org/hadoop/PoweredBy
2) http://hadoopilluminated.com/hadoop_illuminated/Hadoop_Use_Cases.html
3) https://www.cloudera.com/more/customers.html

Hadoop ?
Framework for Distributed Storage & Distributed Processing in commodity cluster.

Programming - Map Reduce
Storage - HDFS
Sql Interface - Hive
Scripting Interface- Pig
Import / Export - Sqoop
Low Latent-OLTP - HBase / Pheonix [ SQL layer to nosql ]

==================================> HDFS

a) Distributed Storage - Understanding the complete process of file ingestion.

--> block -> min chuck of data which can be read / written in one go.
block size -->
Gen1 - 64 MB
Gen2 - 128 MB - hadoop 2.7.2

--> replication: redundancy. - Default Level - 3 copies of every data will be available.

Existing Application --> DR
Hadoop Environment --> Commodity Nodes - failure probability is high
Hence the framework internally handles redundancy.

Client --> 200 MB of data - sample
Cluster --> set of machines [ Master + Slave ] --> 10 Node cluster

EdgeNode Master

1 2 3 4 5 6 7 8 9 10

128 72 128 72 128 72

1) Whom will the client talk to? Master
2) How will the client know who the master is?
a) IP --> config files -- hadoop should also be present on the client - Thick Client
b) Edge Node: - Interface node
http://www.dummies.com/programming/big-data/hadoop/edge-nodes-in-hadoop-clusters/

3) What will be the masters response to the client request? Write - pipeline
4) How many ips will be given as a part of the write pipeline?

The default behaviour is to distributed the data in a horizontal fashion.
128 [ 1,5,9 ]
72 [ 2,7,10 ]

5) How is the first node for every block decided? Network proximity between client and slaves, provided they are available and free to do the ingestion.

6) How are the other nodes for the blocks decided? NOT BASED ON PROXIMITY but based on availability and if multiple nodes are available, then it is chosen randomly, at the Rack Level.

Rack? Physical grouping of nodes in a location
Hadoop is rack aware.

1) The first copy will always go to a node in a rack which is closest to the client.

2) All the other copies [ irrespective of the replication factor ] should be in a different rack. How is the second rack selected?
a) Based on data distribution - rack which is least loaded will be selected.
b) If as a part of a), multiple racks are available, THEN ONLY the rack proximity from the first rack is considered.

7) Who will break the file in to 128 and 72 MB chunks? Hadoop on the client side.

8) Will write happen parallely from client / edge node to the slaves?

9) When will replication start? immediately when a linux block size of 4k is written to a node, it will write it to the next node in the pipeline.

10) When is the file considered as written or when will client tell the NN that the write is done? When the complete file [ including the replicated copies ] are finished.

How will 1 know that 5 is done [ via reverse checksum ]

=========================> Post Morning Tea

File Read Process

EdgeNode Master

1 2 3 4 5 6 7 8 9 10

128 72 128 72 128 72

Client B
read - sample file
1) Client B will talk to Master / Egde Node
2) Response to Client B -- Read Pipeline

[9,5,1] --> reverse proximity order
[10,7,2 ]

--> Historically if a node goes down, what should the client do again? Resubmit the request.
--> In hadoop if node 10 goes down while reading, what will happen? Ideally it should go to the next available node,

====================> Names of Daemons - Hadoop Cluster
Storage Processing
HDFS MapReduce YARN
Gen1 Gen2
Master NameNode JobTracker ResourceManager
Slave DataNode TaskTracker NodeManager
SecondaryNN
PassiveNN

Pre-Flight checkup

1) Image --> Ubuntu14.4_2017.rar
2) Vmware workstation
3) From the workstation we will open our image --> File - Open - Navigate to the folder where Ubuntu 14.4 is extracted. Go inside till the time you see a file called Ubuntu-64
4) Before powering on the image --> we will have to configure 2 things
a) memory --> 8 GB --> 1/2 of whatever you are having
b) processor --> 1/2 of what you have.

5) Power on the Image --> Choose i copied it for the first promnt and No for the IDE:0 prompt
6) Credentials for the image: notroot - hadoop123
7) start winscp --> check the ip of the image --> ifconfig and note down the InetAddress --> 192.168.XXX.XXX
Using that IP connect to Winscp
8) Login in to putty

============================> Set up a Pseudo Cluster - Hadoop 2.7.2

3 different types of deployment of hadoop

1) Standalone --> no daemons - your local FS will act like HDFS
2) Pseudo Cluster --> all deamons would be running in 1 machine. Cluster simulation in one machine
3) Fully Distributed --> different daemons would be running on diff machines.

Exercise: Setup a Pseudo Cluster
Document: Hadoop2.7_Setup.pdf in Day1 folder

In the image provided, i have done the following

1) installed open JDK 1.7
2) created some folders so that everyone will follow the proper instructions
3) downloads - copied all the basic tar files needed

Exercise 1: From step 4 to 11 [ as we have already extracted winscp and putty and logged in to it ]

Exercise 2: From step 12 to 14.
Setting up dir for NN and DN
The NN will contain the metadata of hadoop and the DN will contain the blocks.

permissions for the DN + SNN ==> so that the NN can start these services.

enable passwordless authentication

4 node cluster look like in a distributed environment

Master { NN SNN RM } --> 3 systems
Slaves { DN+NM DN+NM DN+NM DN+NM }

Security --> Kerberos security.

who will start the DFS services --> NN
who will start the YARN services --> RM

What should we do for the password less authentication?
ssh-keygen -t rsa -P ''

--> id_rsa
--> id_rsa.pub

Add this id_rsa.pub key to --> authorized_keys file.

==========>

We will work with the Deployment Descriptor files

--> core --> core-site.xml
--> hdfs --> hfds-site.xml
--> mapred --> mapred-site.xml
--> yarn --> yarn-site.xml

We will have to configure the basic settings for these files.

==========> Post Lunch

Step 16 --> When we start hadoop services, since the whole of hadoop is written in java, it would need JAVA_HOME.

Hence we have to give the complete JAVA_HOME path in the hadoop-env.sh file always.

Step 17 --> format the namenode --> initializes the FS [ HDFS ] for the first time. How to confirm this - look at a folder called current inside hdfs/namenode dir to see if some new files are created.

The formatting of the nn will be done only for the first time for initializing the FS.

Step 18 --> start-dfs.sh --> this will start the dfs services -- 3 new services

Step 19 --> start-yarn.sh --> this will start the the 2 yarn services.

JPS - Java process status tool -: PID for the linux processes

==========>

Step 21 --> we will have to explicitly start JobHistoryServer -->

When a job is finished, it will go to the JHS and the RM immediately. Once YARN is restarted all the completed jobs in the RM will be removed.

Step 22 & 23 --> Checking the HTTP port nos for different components

NN --> 50070 -- Utilities Tab --> Browse the FS
SNN --> 50090
RM --> 8088
JHS --> 19888

==========> Metadata and Data of Hadoop

Step No 24--> Where is the metadata of the cluster?

1) Main memory of the NN PLUS
2) A persisted copy will be in the current dir of the dfs.namenode.name.dir property value

What constitutes the metadata of hadoop

-> fsimage_XXX - Snapshot of the FS at a point of time.
-> fsimage_XXX.md5 - Checksum of the FSimage
-> edits_inprogress_XXX - logs of what happened after the last snapshot
-> edits_XXX-XXX - Range of a series of transactions that was
checkpointed
-> seen_txid

What is Checkpointing --> Applying the edits_inprogress on the fsimage and creating a new fsimage, also creating a new edits_inprogress file, also creating a new edits_XXX-XXX

This is happen only if we have a SNN --> The SNN performs the checkpointing on behalf of the NN.

For Active - Passive NN --> The checkpointing is done immediately.

Step No 25--> Where is the data of the cluster?

The BP-1656218346-127.0.1.1-1509959619371 is the ID of the NN.

In a cluster, can we have multiple NNs? YES

a) Federation --> where we will have multiple NNs for different lines of XXX. Active Active

b) HA --> Active Passive

==========> Running the FS commands

1) hdfs dfs --> press enter --> give you the list of linux commands that can be executed in HDFS

--> Who will run the hdfs dfs commands? clients -->

is HDFS a Physical or a Virtual FS? Virtual
HDFS is a layer above all the DNs.

If we want to interact with HDFS --> hdfs dfs only.

1) hdfs dfs -mkdir /input --> creating a directory in HDFS

Now can we see the input dir on the DNs? NO
We can only see this at the HDFS level
a) hdfs dfs -ls / OR
b) via browse the FS - 50070

2) client is moving a file to HDFS
hdfs dfs -copyFromLocal txns /input

a) Can we see this txns file on the DNs? No
We can only see the blk files
b) We can see the file as txns only in HDFS

Exercise: Step 26 to 29

==========>

Step No 28 --> what is the .meta file present for each block? checksum info

Step No 29 --> which file would have changed at the namenode level --> only the edits_inprogress file.

Basics of Hadoop + HDFS -->

==========> Basics of YARN processing

teragen and terasort are the bench marking applications for hadoop

http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/

https://www.systutorials.com/3235/hadoop-terasort-benchmark/

--> hadoop jar Name.jar packagename.classname applicationparameters

Step 30 --> Will go to a specific dir -->

hadoop jar
hadoopXXXX.jar
teragen
Number of lines to generate --> 500000
Dir in HDFS where the results would be stored -->/benchmark_gen2

Step 31

Step 32

==========> Post Tea Break

Process that is followed when a job is submitted to the cluster

Master

1 2 3 4 5 6 7 8 9 10

128 72
AM

1) The hadoop jar or any application submission command first connects to the RM.
2) The RM will create a Applicaton Master on any node in the cluster
3) AM will talk to NN to get the read pipeline.
4) AM will talk to the RM for finding whether resources are available on the nodes for processing and if yes, then request for the resources [ containers ] on those nodes.
5) The AM will ask the NM to start the containers on the respective nodes

How many containers are needed in any MR application

1) AM container
2) Map container per block
3) By default 1 reduce container per job.

teragen example -->
RM - 17 sec
JHS - 9 sec - actual time taken only for MR

JHS - container details.

The total number of containers --> logs / userlogs dir in hadoop

==========> Big Data Time Lines

1) 2003 --> GFS white paper - Google - Sanjay Ghemawat
2) 2004 --> MR white paper - Google - Jeff Dean
3) 2006 --> doug cutting - ASF
4) 2007 --> Apache Pig - Yahoo
5) 2006-7 --> Big Table - Google - base for HBase
6) 2009 --> Hive - Facebook
7) 2009 --> Spark - Berkely university - Matie Zaharia
8) 2010 --> Sqoop - Cloudera
9) 2012 --> Spark - Open Sourced
10) 2014 --> Spark - ASF

YARN - Hadoop 2.2 GA --> Oct 15, 2013

==========> Let us start with Spark

--> Distributed in-memory processing engine for Big Data

--> Supports different workloads
MR Spark

a) Programming MapReduce RDD
b) SQL Hive SparkSQL
c) Scripting Pig Nothing
d) Machine Learning Mahout Spark ML
e) Graph Processing GraphLabs Graphx
f) Search SOLR / ES Nothing
g) Streaming Apache Storm Spark Streaming
[ Micro Batch ]

--> Can be written in multiple languages --> Developed in Scala, but supports python, java, R

--> Spark is primarily for Batch Processing [ fast In memory ]

a) Interactive Mode --> REPL
b) Batch Mode - Scripts --> controlM, Autosys

--> Spark has a about 80 inbuilt operators - MR has only [ map, combine, partitioner, reducer ]

https://spark.apache.org/docs/1.6.2/programming-guide.html#transformations

--> Spark has different cluster managers
a) stand alone
b) Mesos
c) YARN

http://tm.durusau.net/wp-content/uploads/2013/06/YARN2.png

--> Supports different Data Sources

a) HDFS
b) Local FS
c) Object Store [ S3 ]
d) HBase
e) Cassandra

--> Spark can be deployed
a) on prem
b) on cloud - AWS, Azure, Rackspace

==========> Let us setup spark

Sample file -->
How are you
I am fine
How about you

Exercise: Step No 1 - 4

export SPARK_HOME=/home/notroot/lab/software/spark-1.6.2-bin-hadoop2.6
export PATH=$PATH:$SPARK_HOME/bin
export PATH=$PATH:$SPARK_HOME/sbin

Spark Architecture:- https://0x0fff.com/wp-content/uploads/2015/03/Spark-Architecture-Official.png

Component in spark

1) Spark Driver --> Also on the Client Side
2) SparkContext --> connects to the Cluster Manager
3) Cluster Manager
4) Worker Node --> Slave System
5) Inside the worker --> Executor --> Container. The same container can execute any number of tasks given by the Cluster Manager. So no need to shut down and bring up the container. So definite performance improvement.
6) Inside the Executor --> We will have tasks [ smallest unit of a work ] and a cache.

==========>

Hadoop --> Promise - Commodity cluster - limited RAM and distributed processing

RAM - 64 - 128 GB RAM

Spark --> Infact for spark, we need huge amount of memory.

Real world Deployment -- 2 ways

Hadoop + Spark together in 1 cluster -- provided you have enough memory.

OR ELSE

Hadoop Cluster

Seperate Spark Cluster

==========>

What got started when spark-shell REPL was triggered.

1) It started with a inbuilt web server --> Apache Jetty
2) Displays the Spark Version - 1.6.2
3) Displays the Scala Version - 2.10
4) Starts with SparkDriver
5) Starts with SparkUI --> 4040
6) Gives the SparkContext handle as sc.
7) It starts the basic components of Hive -- Database that hive uses? Apache Derby. Where is the actual data stored of hive in HDFS? /user/hive/warehouse
8) Gives the SQLContext handle as sqlContext

==========>

Exercise: Word Count --> Finding the number of unique words and its count.

1) Load the data in to something.

Difference between val [ immutable] v/s  var [ mutable ]

val fname = sc.textFile() --> What is the default location for text file? HDFS location --> via the COMMON_HOME

val fname = sc.textFile("file:///home/notroot/lab/data/sample")
fname.collect()

==> Lazy Evaluation --> unless something is explicitly triggered, nothing will happen.

2 types of operations

a) transformations - Always lazily evaluated and will be added to a DAG
b) action --> DAG will be materialized

--> Ensure that you exit spark
--> VMWare workstation --> VM menu - Power - Suspend

Pre-Req for tomorrow

1) What is Apache Spark-.mp4
https://www.youtube.com/watch?v=SxAxAhn-BDU&t=13s
2) Introducing Apache Spark Into Your Big Data Architecture
https://www.youtube.com/watch?v=aKFnfSbNkm4

Andrew NG course in Coursera:- https://www.coursera.org/learn/machine-learning

---

# Interview Questions: System Design

By far the most exhaustive: https://github.com/donnemartin/system-design-primer
Start here -> https://github.com/donnemartin/system-design-primer#index-of-system-design-topics
- References from the link above:
  - https://github.com/checkcheckzz/system-design-interview#qs
  - https://sourcemaking.com/design-patterns-and-tips
  - https://sourcemaking.com

Scalability for Dummies: http://www.lecloud.net/post/7295452622/scalability-for-dummies-part-1-clones

https://www.interviewbit.com/courses/system-design/topics/interview-questions/
http://blog.gainlo.co/index.php/category/system-design-interview-questions/
https://hackernoon.com/top-10-system-design-interview-questions-for-software-engineers-8561290f0444
https://www.youtube.com/playlist?list=PLrmLmBdmIlps7GJJWW9I7N0P0rB0C3eY2

https://lethain.com/introduction-to-architecting-systems-for-scale/

---

# Interview Questions: SQL

https://www.toptal.com/sql/interview-questions

---

# Interview Questions: Programming

Good For revision:
http://javaconceptoftheday.com/java-interview-programs-with-solutions/

Most common algo questions
https://www.geeksforgeeks.org/top-10-algorithms-in-interview-questions/
https://www.toptal.com/algorithms/interview-questions
https://www.geeksforgeeks.org/commonly-asked-algorithm-interview-questions-set-1/

Dynamic Programming
https://www.geeksforgeeks.org/top-20-dynamic-programming-interview-questions/

https://javarevisited.blogspot.in/2017/07/top-50-java-programs-from-coding-Interviews.html
https://howtodoinjava.com/java-interview-puzzles-answers/
https://codingpuzzles.com/

https://leetcode.com/problemset/top-interview-questions/
https://www.w3resource.com/java-exercises/basic/index.php
http://www.instanceofjava.com/2016/03/java-collections-interview-questions.html
http://www.instanceofjava.com/2017/05/java-practice-programs-with-solutions.html
https://www.w3resource.com/java-exercises/collection/index.php
http://www.instanceofjava.com/p/frequently-asked-java-programs-in.html
http://javaconceptoftheday.com/java-interview-programs-with-solutions/
https://javarevisited.blogspot.in/2017/07/top-50-java-programs-from-coding-Interviews.html
https://codingpuzzles.com/

http://www.java2novice.com/java-interview-programs/
http://www.java67.com/2012/08/10-java-coding-interview-questions-and.html

**Java Concepts**
http://www.java67.com/2012/09/top-10-tricky-java-interview-questions-answers.html
http://www.java67.com/2012/09/java-collection-interview-questions.html

---

# Interview Prep References

https://www.fromdev.com/2016/02/best-interview-preparation-sites.html

---

# List/ArrayList/LinkedList key APIs

ArrayList/List

- creation

    - new ArrayList() // logical size zero, physical size 10
    - new ArrayList(30) // logical size zero, physical size 30
    -
    - new ArrayList(another collection) // all the elements from the other collection are added)
-
- isEmpty - whether it's empty
- Retrieval related
    - contains
    - indexOf and lastIndexOf (first/last occurrence of an object)
    - get(positon)
- arraylist to array - list.toArray
- Add
    - set(position, newElement) - overwrites element at position
    - addAll (list)
    - addll(pos, list)
    - add - adds at end
    - add(pos, newElement) - adds at position and shift the others
- Removal
    - remove(pos)
    - remove(obj)
    - clear - empties the list
- subList - a "view" on the list (so still backed by the original list)

ArrayList specific

- ensureCapacity (ArrayList method) - to set a particular size (increase)
- trimToSize

LinkedList specific

- offer and poll can make a linkedList work as a queue
- push and pop can make a linkedList work as a stack
- removeFirst/LastOccurrence

---

# HashSet

Java **HashSet** is very powerful Collection type when you want a collection of unique objects. HashSet doesn't allow duplicate elements. HashSet also gives constant time performance for insertion, removal and retrieval operations. It is also important to note that HashSet doesn't maintain any order. So, It is recommended to use HashSet if you want a collection of unique elements and order of elements is not so important. If you want your elements to be ordered in some way, you can use **LinkedHashSet** or **TreeSet**.

**LinkedHashSet** is an ordered version of HashSet. That means, HashSet doesn't maintain any order where as LinkedHashSet maintains insertion order of the elements. LinkedHashSet uses **doubly linked list** internally to maintain the insertion order of it's elements. We have seen this in How LinkedHashSet Works Internally In Java?. As LinkedHashSet maintains doubly linked list (along with HashMap), the performance of LinkedHashSet is slightly slower than the HashSet. But, LinkedHashSet will be very useful when you need a collection of elements placed in the order they have inserted. We will see one such example of LinkedHashSet in this article.
*-< There is a constructor param that can make insertion order to access order, so latest accessed element sits at the end of the iteration*

*All collections can be initialized with an initial capacity*

```
//Constructor - 1

  public LinkedHashSet(int initialCapacity, float loadFactor)
  {
        super(initialCapacity, loadFactor, true);              //Calling super class constructor
  }

  //Constructor - 2

  public LinkedHashSet(int initialCapacity)
  {
          super(initialCapacity, .75f, true);             //Calling super class constructor
  }

  //Constructor - 3

  public LinkedHashSet()
  {
          super(16, .75f, true);                //Calling super class constructor
  }

  //Constructor - 4
```

```
  public LinkedHashSet(Collection<? extends E> c)
  {
          super(Math.max(2*c.size(), 11), .75f, true);          //Calling super class constructor
          addAll(c);
}
```

**TreeSet** is another popular implementation of Set interface along with **HashSet** and **LinkedHashSet**. All these implementations of Set interface are required in different scenarios. If you don't want any order of elements, then you can use HashSet. If you want insertion order of elements to be maintained, then use LinkedHashSet. If you want elements to be ordered according to some Comparator, then use TreeSet. The common thing of these three implementations is that they don't allow duplicate elements.

*http://www.baeldung.com/java-linked-hashmap*

---

# Java Fundamentals: Multithreading with Concurrency

- What is concurrency
  - Multi-tasking - run several processes on the same machine, each can progress independently
    - Tasks are achieved by process in OS
      - usually standalone
      - often no knowledge of each other
      - own memory allocation and usually segmentation
        - processes can't access or modify other's data
        - any communication between related processes has to take place via
          - Filesystems
          - Sockets
          - Specifically designed shared memory
      - Processes have their own accounting data
    - Time Slicing
      - Allocated time for a process
      - Interrupted when time is up
      - Task's state saved
      - another task's state is restored
  - Multithreading
    - Definition
      - Running several threads of computation at the same time in the same process
      - Share memory and processes
      - lower overhead than processes
      - part of the same program
      - cannot exist outside the processes' lifetime
      - can be seen as light weight processes (but not free)
    - Run in parallel?
      - No guarantee
      - The scheduler users time slicing
      - JVM manages threads in the scheduler
    - Overhead
      - need their own stack
      - have some memory overhead
      - creating and destroying takes time
      - Scheduler swapping one to next takes time
      - can cause starvation (other threads not getting any cpu time)
  - Concurrency
    - **When a task is broken down in smaller pieces and run simultaneously**
    - Even without parallelism
      - When some parts are stuck, others can make progress
  - Optimum number of threads
    - maximize throughput
    - minimize time
    - Amdahl's law
    - Experimentation is best

What is concurrency?
When a task is broken down in smaller pieces and run simultaneously. Can be achieved in multiple ways

- Tasks in nulti-tasking
- Threads in multithreading
- Distribution of work across grid or the cloud

Benefits

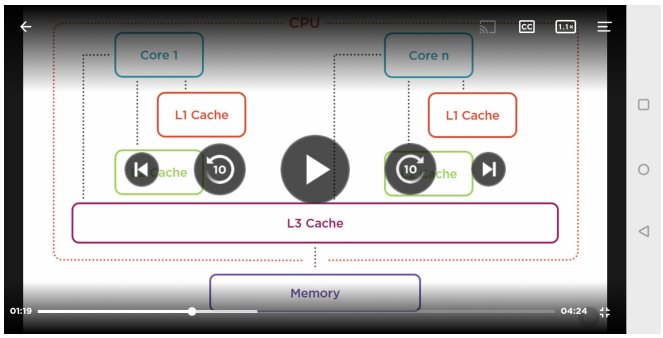
- quicker resolution of problems

Difference from multi-threading

- one way of achieving concurrency
- threads share memory and resources
- doesn't always run in parallel, time slicing takes place

Suggest issues related to caching in multithreaded systems

- Distances between CPU and memory
  - small but not insignificant (bus is the medium of transport)
  - exclusive access to bus may take many clock cycles
  - therefor cache memory inside CPU exists to save time
- Memory Structure
  - Larger the number, larger and slower is the cache
  - L1 and L2 are local to the core, L3 is common




- Having multiple copies of data in caches means all the troubles that go with keeping them in synch
- Issues
  - Cache Synchronization
    - it takes time
  - an individual thread will see the data it wrote it when it tries to read again
  - Inconsistent data
    - Some values are old, some new if some other thread is updating it
    - It will break invariant
- Prevention
  - Use monitors and locks
    - avoid others seeing data being updated
  - Allow only one thread to update something

**********

Multi-threading code

- ends when main thread exits main
- spawned non daemon threads exit run

**To create new threads, use start and not run (pretty easy mistake to make and I did!)**

Thread State Machine

New -----<start called> ----> Runnable ----------<exit called>----------> Terminated

calling start again results in an illegal state
Runnable is a functional interface

Daemon threads exist as long as non daemon threads exist (example - garbage collector)

setDaemon(true) before calling start will make a thread a daemon thread (should not be relied upon for things that need clean up before exiting)

- they can just go off suddenly when no non daemon threads are around, no clean up etc.

Thread.sleep(long millis) -> sleeps, but doesn't block others, so other can use the CPU
Handy unit conversion: TimeUnit Class to do unit conversions

Platform dependent stuff (best avoided)
sleep(0) -> may or may not sleep, (platform dependent)
yield() - give up control (platform dependent)
setPriority(newPrio)

**jStack and jConsole**

- java monitoring and management console
- jps - gives the process Id
- run jstack with id from JPS

Interrupt can be called on a thread, it will cause an interrupted exception to be thrown

isAlive tells whether thread is alive or not (use of join not very clear)
method join -> join() ->
join(millis) ->

Lot of work to catch exceptions from threads (SetUncaughtExceptionHandler)

ThreadGroup: to have a group of threads operated upon in combined fashion

ThreadPools coming in ,threadgroups are almost dead

ThreadLocal helps in setting a value private to a thread (the class just needs to extend ThreadLocal)
-> can cause memory leaks if memory is not cleared correctly

Key methods on Threads (all three depracated)

- Stop: Mutex can have problems (as they may not release the locks they hold)
- Suspend
- Resume

**Sharing Memory Across Threads**

- Threads need to share their data
- more involved than making values visible
- values must also be correct
    - Issue with caching
    - Problems with compiler trickery
- Consider how to share data at the design stage
- Plan it out properly

Thread safe- means runs correctly in a multi threaded environment
Program Data

- has been stored
- can be updated

Potentially shareable

- visible to other threads
- static variables, class data, composed class data, collection items

Is there a way to ensure data is correctly published (only way is to carefully inspect code)

Java Memory Model

- describes how threads in java programming language interact through memory
- defines a set of guarantees which when applied to a program ensure memory interactions between threads occur in a specified deterministic fashion
- Data Synchronization: ensure all threads see the updated value (different from synchronized keyword)

---

# SQL Interview Links

https://www.agent.media/grow/sql-interview-questions/
http://www.complexsql.com/hierarchical-queries/
http://www.complexsql.com/complex-sql-queries-examples-with-answers/
https://dwbi.org/database/sql/72-top-20-sql-interview-questions-with-answers

---

# Interview Questions: Oracle

https://www.javatpoint.com/oracle-interview-questions
https://career.guru99.com/top-50-oracle-interview-questions-and-answers/

---

# Spring - General Notes

Dependency Injection: Two objects dependent to each other, decouple them and be able to inject it rather than building it inside.
-> one form of inversion of control

What is inversion of control: In traditional programming, the custom code that expresses the purpose of the application calls the reusable library code , but with inversion of control,

framework calls into the custom, task specific code

example - Traditional code - have a thread to manage UI, which calls the library for displaying UI and acting upon user actions. In IoC, we may have a windowing UI system with commonly built behaviors and hooks to put in custom logic. App code is fit into the custom area (fill in the blanks)

Spring is a container of beans - what is a container, basically a program that pew-creates a bunch of objects that are essential for your application code to run. Tomcat does it with servelts using web.xml

Similarly spring will have beans - Spring container will manage the life-cycle of those objects. Stuff created using new is managed by app (not by spring).

Factory Pattern -> Factory is a creator class for producing objects of another type(s). In case of spring, you need to have a configuration that specifies what all kind of objects to be created and how.

Spring has Spring Bean Factory - it reads from xml (or annotations) and creates the needed beans
**BeanFactory** class - can be created with an xml in the constructor

**Autowiring**

Types

- byName
- byType
- Constructor

@Component - tells spring that this is something spring needs to handle
@Autowired - inject it at the right place - a variable marked with this means, spring will provide the impl

- If an interface is used, Spring will look for all the implementations (if there is one, it is easy)
- If there are more -> name the variable to match the implementation class name
    - or use a @Qualifier(value="Class Name")

**Key Annotations**

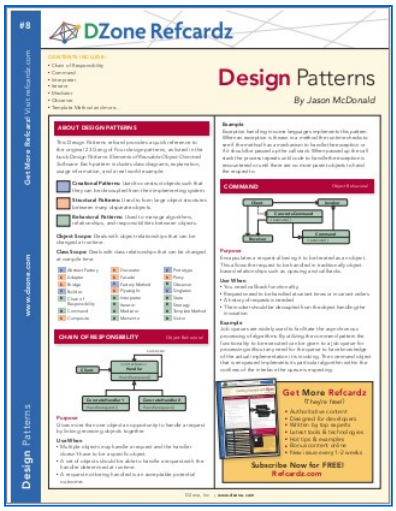
- Component: Things that will be managed by Spring
- Autowired: variables that need implementations to be supplied by Spring
- Configuration: Launch a Spring Application context
- ComponentScan(packages) - packages where components need to be searched
- RunWith(SpringJ4unitRunner) - to run a unit test using spring context
- @ContextConfiguration - which context to use - basically the class with configuration annotation

Front Controller Design Pattern

- All the requests from browser go to front controller
- Then it decides which controller to invoke
- Relevant controller will identify the view and return the control to front controller with the view/model information
- Dispatcher Servlet is the front controller for spring MVC
- ViewResolver in MVC
- FrontController can take care of all the common concerns like
    - view resolution
    - handler resolution
    - theme
    - internationalization

---

# Design Patterns Refcard



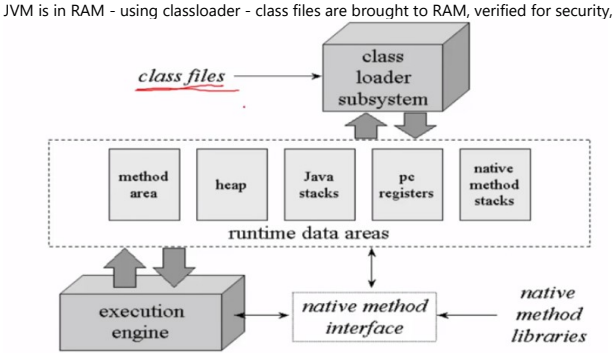

---

# JVM/Classloader Notes

**ClassLoaders**:

- System classloader (everything in classpath)
- Extension classloader (all classes in ext directory - typically jre/lib/ext)
- Bootstrap classloader (core java)

**JVM**

- Linking in C -> .c -> compiler -> .obj -> linker -> .exe
- In java no linking (.class files are generated)
- JVM is in RAM - using classloader - class files are brought to RAM, verified for security, JIT compiler converts to native machine code
- 


**Loading**
rt.jar - bootstrap classloader
jre/lib/ext - extension class loader
application classpath - your files (-cp)

**Linking**

- Verifying: Verified for security and compatibility (versions etc.) - ClassNotFoundException
- Preparing: Initialize all values to defaults
- Resolving: All references initialized during this phase - so basically checking if i have all the classes i need to instantiate a particular class - failure leads to ClassDefNotFoundException

**Initialization**

- static init blocks
- actual values assigned if there are any

**Memory Areas (all can be tuned according to your needs)**

- Method Area - Classes for your application
- Heap - all the objects created
- Stacks - load/execute the methods one by one - use this area for its variables
- pc registers - program counter registers - responsible for thread management

- native method stacks - os dependent stuff is loaded here, DLLs (inside lib in windows), diff for other OSs

**Execution Engine**

- Interpreter - interprets bytecode line by line - passes to os using native method interface
- Garbage Collection -
- JIT compiler - similar instructions received multiple times are stored and optimally returned for the next occurrence
- Hotspot profiler - provides the stats to JIT for optimization

---

# Java Garbage Collection

Heap Space

- all objects created here

Garbage collection - Automatic memory management system created by java. Unused objects killed and memory returned
new allocates the memory space

---

# Intervew Preparation: Key Terms and Tech Skills

Agile

- Scrum
- User Stories
- Estimation
- JIRA
- Lean and Agile

Architecture and Design

- Domain Driven Design
- Microservices
- UX
- Data
- CQSR
- Event Sourcing
- REST
- Spring Boot
- Patterns of Cloud Integration
- BDD

Software Engineering

- SCM
- Code security
- Code quality
- Frameworks
- UI Dev

Cloud

- Cloud Computing
- Cloud Native
- 12 Factor

Continuous Delivery

- CI/CD
    - Jenkins
- Unit Testing
- Test Pyramid

Logging and Monitoring

- ccc

Product Support

- DevOps
- Incident Management
- ELK - Elastic Logstatsh and Kibana
- Change Management
- Splunk
- AppD

- EPV
- Batch
- Scheduling
- Call Tree

Resiliency

- Cloud Resiliency
- Cloud Security
- Hystrix
-

---

# System Design/Scalability - Harvard Lecture/Others

Read this if you have time as this is really concise already and will be a great read before interviews - https://github.com/donnemartin/system-design-primer#step-2-review-the-scalability-article

In case you are really short on time - read the notes below

FTP - sends username and password on the wire in clear text
VPS- Virtual private servers - GoDaddy, Dreamhost, AWS (EC2) etc (generally use Hypervisor) - you receive a slice of resources (hardware, RAM)

- If someone has physical access to server, usually no security can save your data

Shared Web Hosts - ???

quad core - like 4 CPUs independently (within a core you will be time slicing but across cores, totally parallel)

Scaling your website under unusually high load

- Vertical Scaling - more resources (ram, cpu, storage). At some point you will hit a limit, beyond which it can't go (financially, technologically)
    - IDE, parallel ATA, SATA - hard drives ~ 720 rpm
    - mechanical drives - fastest -SAS- serial attached scsi - really fast - 15000 rpm
    - Solid state devices - faster than mechanical drives
- Horizontal Scaling
    - build a system that can use more machines (with average hardware specs)
    - that poses a problem as the requests actually are distributed but need to appear its going to the same place
        - Load Balancer - use the IP of load balancer (other hosts don't need a public IP addresses
            - Private vs Public - private IP addresses are not visible to external networks
            - There are schemes reserved for private IP address like 192.168...., 10.xyz....
            - decides where to route a particular request
                - based on load (any server can do anything)
                - based on function (dedicated for specific purpose)
                - Round Robin -> DNS can return multiple IPs one by one for the same url. Not very smart and can overload one server. OS, browsers everyone caches the responses from DNS server and can make this worse
                - Sessions are typically implemented per server, so any of these will create issues, where you lose session information - so you should manage shared state in a common resource (can be load balancer). Some shared storage techs
                    - FC, iSCSI, Database, NFS
            - Options for load balancers
                - Software
                    - ELB - Amazon - elastic load balancer
                    - HA Proxy
                    - LVS
                - Hardware
                    - Barracuda
                    - Cisco..
                    - Citrix (cost ->20k for the cheapest ones, avg $100k)
        - Sticky Sessions
            - Shared Storage
            - Cookies
                - few KBs
                - server Id can be stored in the cookie
                - downside: IP can change, sever can go down, principally - not a good idea to expose our IP scheme, add an ID that points to the final IP
            -
        -
    -

Redundant array of independent disks - RAID
RAID0 - 2 identical hard drives - every time a write happens, it alternates (writes are fast) - striping
RAID1 - 2 identical hard drives - every time a write happens, mirror it (either of the drives can die, and nothing will happen)
RAID10 - 4 drives, both striping and redundancy

Memcached - in memory cache - should be installed on the web server - its all in RAM
-> disks are slow
-> databases are better
-> In Memory cache (key value store) - caches are finite - so you can expire the oldest used

Replication

- Master/slave

Partitioning

High Availability - multiple servers ding the job and taking over when someone is down

**Scalability for Dummies**
Clones - with a distributed system, we need to ensure they all share same code and don't specifically store any information about the user, which should go to a shared storage
DB - Have master/slave architecture - many read nodes, fewer write nodes . Master may need to keep vertically scaling. Other option is go to NoSQL way and at the same time denormalize your data to reduce the cost of joins or let your app do the joins
Introduce a cache: In memory cache like Redis or memcached - cache is a key value store, should sit between your app and DB layer as a buffering layer. It's all in RAM so will be lightening fast
DB Cache - create a hash of your query - uses it as key and results is the output. Invalidation can be really hard as queries can be really complex, and we may end up deleting it for all the updates
Cached Objects - store the whole object in the cache
Some uses

- User sessions can go in the cache as well
- activity streams

Asynchronous processing

**Perf vs Scalability**

- If you have a **performance** problem, your system is slow for a single user.
- If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.

# Latency vs throughput

**Latency** is the time to perform some action or to produce some result.

**Throughput** is the number of such actions or results per unit of time.

# Availability vs consistency

**In a distributed computer system, you can only support two of the following guarantees:**

- **Consistency - Every read receives the most recent write or an error**
- **Availability - Every request receives a response, without guarantee that it contains the most recent version of the information**
- **Partition Tolerance - The system continues to operate despite arbitrary partitioning due to network failures**

***Networks aren't reliable, so you'll need to support partition tolerance. You'll need to make a software tradeoff between consistency and availability.***

**Consistency Patterns**
Weak consistency - most recent writes may not appear - a best efforts approach is taken (they may be lost - ok for real time scenarios - a video chat, few lost seconds in between are ok to be lost)
Eventual consistency - all reads will eventually see a write
Strong consistency - every read will see the latest write or error - updates replicated synchronously

**Availability Patterns**
*Fail over*

**Active-passive**

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.
In active-active, both servers are managing traffic, spreading the load between them.

Replication

- master/master
- master/slave

**Domain Naming System**
A Domain Name System (DNS) translates a domain name such as www.example.com to an IP address.
-> DNS results get cached by your browser, OS for certain period based on TTL

-> Routing can use one of the following schemes
  -> weighted round robin
  -> latency based
  -> geolocation

**Content Delivery Network (CDN)**
A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN

**Load Balancers**
Load balancers distribute incoming client requests to computing resources such as application servers and databases. In each case, the load balancer returns the response from the computing resource to the appropriate client. Load balancers can be implemented with hardware (expensive) or with software such as HAProxy. Can also do SSL work (SSL Termination) and saves everyone else that time. Also, can do a bit of session persistence

Reverse Proxy
A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public. Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client.

- **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client
- **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
    - Removes the need to install X.509 certificates on each server
- **Compression** - Compress server responses
- **Caching** - Return the response for cached requests
- **Static content** - Serve static content directly

## Microservices

can be described as a suite of independently deployable, small, modular services

**ACID** is a set of properties of relational database transactions.

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

There are many techniques to scale a relational database: **master-slave replication**, **master-master replication**, **federation (divide data functionally across DBs)**, **sharding**, **denormalization**, and **SQL tuning**.

Sharding -> Sharding distributes data across different databases such that each database can only manage a subset of the data

Denormalization attempts to improve read performance at the expense of some write performance

## NoSQL

NoSQL is a collection of data items represented in a **key-value store**, **document-store**, **wide column store**, or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency.

**BASE** is often used to describe the properties of NoSQL databases. In comparison with the CAP Theorem, BASE chooses availability over consistency.

- **Basically available** - the system guarantees availability.
- **Soft state** - the state of the system may change over time, even without input.
- **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

**Key-value store**

> Abstraction: hash table

**Document store**

> Abstraction: key-value store with documents stored as values

**Wide column store**

> Abstraction: nested map `ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>`

**Graph database**

> Abstraction: graph

Reasons for **SQL**:

- Structured data
- Strict schema
- Relational data
- Need for complex joins
- Transactions
- Clear patterns for scaling
- More established: developers, community, code, tools, etc
- Lookups by index are very fast

Reasons for **NoSQL**:

- Semi-structured data
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Store many TB (or PB) of data
- Very data intensive workload
- Very high throughput for IOPS

Sample data well-suited for NoSQL:

- Rapid ingest of clickstream and log data
- Leaderboard or scoring data
- Temporary data, such as a shopping cart
- Frequently accessed ('hot') tables
- Metadata/lookup tables

#### Caching

- Client caching
- Server side caching
- CDNs are a cache too
- Web servers can cache stuff too
- DB caching - hash of query and associated results
- Application Caching - Memcached, redis
- LRU is one of the algorithms for cache replacement - many others possible
- Objects can be cached too
- Update strategies
    - Cache aside: (Lazy Loading)
        - Look up a value in cache, if it's a miss
        - Load from DB
        - Add to cache
        - return val
    - Write through
        - Cache is the main layer that provide read/write support
        - Cache is responsible to keep DB up to date
    - Write behind
        - same as write through, except asynch
    - Refresh Ahead
        - Proactively refresh entries that are approaching TTL

#### Asynchronism

Asynchronous workflows help reduce request times for expensive operations that would otherwise be performed in-line

#### Network

OSI (Open Source Interconnection) 7 Layer Model

| Layer | Application/Example | | Central Device/ Protocols | |
|---|---|---|---|---|
| **Application (7)** Serves as the window for users and application processes to access the network services. | **End User layer** Program that opens what was sent or creates what is to be sent<br>Resource sharing • Remote file access • Remote printer access • Directory services • Network management | | **User Applications**<br>SMTP | |
| **Presentation (6)** Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network. | **Syntax layer** encrypt & decrypt (if needed)<br>Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation** | | JPEG/ASCII EBDIC/TIFF/GIF PICT | |
| **Session (5)** Allows session establishment between processes running on different stations. | **Synch & send to ports** (logical ports)<br>Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc. | | **Logical Ports**<br>RPC/SQL/NFS NetBIOS names | |
| **Transport (4)** Ensures that messages are delivered error-free, in sequence, and with no losses or duplications. | **TCP** Host to Host, Flow Control<br>Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing | PACKET FILTERING | TCP/SPX/UDP | |
| **Network (3)** Controls the operations of the subnet, deciding which physical path the data takes. | **Packets** ("letter", contains IP address)<br>Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting | | **Routers**<br>IP/IPX/ICMP | |
| **Data Link (2)** Provides error-free transfer of data frames from one node to another over the Physical layer. | **Frames** ("envelopes", contains MAC address)<br>[NIC card — Switch — NIC card] (end to end)<br>Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control | | **Switch Bridge WAP**<br>PPP/SLIP | Land Based Layers |
| **Physical (1)** Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium. | **Physical structure** Cables, hubs, etc.<br>Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts | | **Hub** | |

HTTP is a method for encoding and transporting data between a client and a server. It is a request/response protocol: clients issue requests and servers issue responses with relevant content and completion status info about the request. HTTP is self-contained, allowing requests and responses to flow through many intermediate routers and servers that perform load balancing, caching, encryption, and compression.

UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP support, UDP is generally more efficient. UDP is less reliable but works well in real time use cases such as VoIP, video chat, streaming, and realtime multiplayer games.

Use UDP over TCP when:

- You need the lowest latency
- Late data is worse than loss of data
- You want to implement your own error correction

Computer performance is a bit of a shell game. You're always waiting for one of four things:

Disk
CPU
Memory
Network

But which one? How long will you wait? And what will you do while you're waiting?

---

# Domain Driven Design

- Eric Evans coined in 2003
- Engage with domain experts
- Focus on one sub domain at a time
- Separation of concerns
- Design and develop sub domains independently
- More useful when there is substantial complexity in business/domain
    - If there is more technical complexity and domain is simple, not much use of DDD
- ubiquitous language
-

# GraphQL

https://www.howtographql.com

**RPC** - was the first standard for API design - 1960s - lient-server interaction model where a client causes a procedure (or method) to execute on a remote server. There are a lot of nice things about RPC. Its main principle is allowing a developer to treat code in a remote environment as if it were in a local one, albeit a lot slower and less reliable which creates continuity in otherwise distinct and disparate systems. RPC, by design, creates quite a lot of coupling between local and remote systems — you lose the boundaries between your local and your remote code

**SOAP**

The next major API type to come along was SOAP, which was born in the late 90s at Microsoft Research. An ambitious protocol spec for XML based communication between applications. SOAP endpoints allow you to request data with a predetermined structure. The most significant downside of SOAP is that being so verbose; **it is nearly impossible to use without lots of tooling**. SOAP uses HTTP as a dumb transport and builds its structure in the request and response body. number of bytes needed for the XML structure makes it a poor choice for serving mobile devices or chatty distributed systems.

| | SOAP | REST |
|---|---|---|
| HTTP Verbs | 🤦 | GET, PUT, POST, PATCH, DELETE |
| Data Format | XML | Whatever you want |
| Client/Server Contracts | All day 'ery day! | Who needs those |
| Type System | ✓ | JavaScript has unsigned short right? |
| URLs | Describe operations | Named resources |

**REST**

REST, on the other hand, throws out the client-server contracts, tooling, XML and bespoke headers, replacing them with HTTPs semantics as it is structure choosing instead to use HTTP verbs interact with data and URIs that reference a resource in some hierarchy of data.

REST completely and explicitly changes API design from modeling interactions to simply modeling the data of a domain

*Weaknesses*
you often have a collection of API calls to make that depend on each other to render a complete application or page. a lot has changed in the two decades since Fielding's paper. Around 2008, mobile computing went mainstream. With mobile, we effectively regressed a decade in terms of speed/performance overnight. In 2017, we have nearly 80% domestic and over 50% global smartphone penetration, and it is the time to rethink some of our assumptions about API design

**REST Is Chatty**
REST services tend to be at least somewhat "chatty" since it takes multiple round trips between client and server to get enough data to render an application. 55ms 4G latency * 8 requests = 440ms overhead

| Delay | User reaction |
| --- | --- |
| 0 - 100 ms | Instant |
| 100 - 300 ms | Feels sluggish |
| 300 - 1000 ms | Machine is working... |
| 1 s+ | Mental context switch |
| 10 s+ | I'll come back later... |

in many cases it takes less time to download one large request than many small ones (most cases it is due to http/1 design, http/2 will do ok)

HTTP aside, the final piece of why chattiness matters, has to do with how mobile devices, and specifically their radios work. The long and short of it is that operating the radio is one of the most battery intensive parts of a phone so the OS turns it off at every opportunity. Not only does starting the radio drain the battery but it adds even more overhead to each request.

**TMI (Overfetching) or under-fetching leading to n+1**
REST-style services send way more information than is needed

A case for GraphQL ----------->

### MINIMAL HTTP TRAFFIC

We know the cost of every (HTTP/1) network request is high on quite a few measures from latency to battery life. Ideally, clients of our new API will need a way to ask for all of the data that they need in as few round-trips as possible.

### MINIMAL PAYLOADS

We also know that the average client is resource constrained, in bandwidth, CPU, and memory, so our goal should be to send only the information our client needs. To do this, we will probably need a way for the client to ask for specific pieces of data.

**GraphQL (a new api standard) - Facebook - 2012 (2015 spoke publicly)**

- Declarative data fetching - helps with minimizing payload size and reduced network traffic. Exposes one endpoint and the input is the query (query language for APIs)
- Queries can be seen as data - so a lot of smart handling can be done to those
  - cache results if needed for certain queries
  - construct bigger queries using parts, run them in parallel or use caches etc.
- Tooling:
  - With a simple typed schema on the server, there are nearly endless possibilities for rich tooling.

GraphQL is just a type system for your data, a query language for traversing it - the rest is just detail.

There is enormous support of developer community to build tools around it, so a natural choice for API design

GraphQL - DB is replaced by services, so we need a query language on top of the restful services

Falcor -- netflix created a parallel product

n+1 problem -> first API doesn't give enough and then you have to run a new set of queries on each item in the list that came back in the first call

GQL better than Rest

- rapidly changing requirements on clients make it harder to maintain APIs
- rest leads to over/under fetching (under will lead to n+1 requests problem)
  - you will need to keep updating API if we want to avoid under-fetching, but you need server to keep changing
- It also tells us what the client "actually" needs, some fields may never get requested by users
- Resolvers are the main backend component and they can be tuned based on the usage pattern

GQL needs a type system and Schema

Core Concepts of GQL

- Schema
  - Schema Definition Language
  - Relation
  - Root Types - entry points for the APIs
  - Mutation Types - stuff that can be updated
  - Subscription Type
- Query (Read)
  - payload - everything after the root entity
  - Arguments in the query to filter the records
- Mutations (CUD)
  - create
  - update
  - Delete
- Subscriptions
  - client holds a steady connection with the server and an action is triggered based on the subscription
    - like creating a new item

GQL is only a spec, can be used with any protocol

**Use Cases**

- GQL server with a connected DB
- GQL server to integrate existing systems
- A hybrid approach with a connected DB and integration of existing systems

All the logical sits in the resolver functions and that is defined for every piece defined in the schema

---

# Interview Preparation: BI