**OOP**

# What is abstraction?

* Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.
* Wrapping common characteristics, states and functionality in abstract structures.
* There are two ways to achieve abstraction in java
  + Abstract class (0 to 100%)
  + Interface (100%)

# What is encapsulation?

* There are two main points to encapsulation – keeping stuff that belong together together and also expose only the stuff we want to other classes:
  + Put operations that operate on some data, and the data in the same place. In object-oriented programming, this means putting them in the same class (cohesion).
  + The other point is that you want to be able to keep classes loosely coupled, and only expose data that you want to share with other classes in the system.

What is the difference between public, private, and protected keywords?

* + We use these keywords to specify access levels for member variables or methods.
    - Public variables are visible to all classes.
    - Private variables are visible only to the class to which they belong.
    - Protected variables are visible only to the class to which they belong, and any subclasses.
  + A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as package-private), it is visible only within its own package (packages are named groups of related classes).

# What is inheritance?

* Inheritance is a mechanism in which one object acquires all the properties and behaviors of parent object.
* The idea is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
* Inheritance represents the IS-A relationship, also known as parent-child relationship.

Why use inheritance?

* For Code Reusability.
* For Method Overriding.

# What is polymorphism?

* A concept by which we can perform a single action by different ways.
* Different functionality, while sharing the same interface. *Example*: if we have base class Animal with method Talk() and 2 sub classes – cat and dog, when executing the method they behave differently – cat says “maow”, dog says “bow-bow”.
* The provision of a single interface to entities of different types

# What is upcasting and downcasting?

* Up-casting is casting to a supertype, while downcasting is casting to a subtype. Supercasting is always allowed, but subcasting involves a type check and can throw a ClassCastException.
* To call a superclass's method you can do super.method() or be performing the upcast.
* To call a subclass's method you have to do a downcast and risk the ClassCastException.

# What is composition?

* Composition is simply, when an object is made up of other types. In the composition relationship, one object owns the other object that is made of, just like your hand is part of your body. To use composition in Java, you use instance variables of one object to hold references to other objects

What is the difference between composition and inheritance?

* + Inheritance is an "is-a" relationship. Composition is a "has-a".You do composition by having an instance of another class C as a field of your class, instead of extending C. *Example*: Car has a Engine and Car is a Automobile

Why it is preferred over inheritance?

* + With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type.
  + Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates Inheritance.
  + Does TypeB only want only some/part of the behavior exposed by TypeA? Indicates need for Composition.

How it differs from aggregation?

* + Aggregation implies a relationship where the child can exist independently of the parent. *Example*: Car ->Tires. The tires can be taken off of the car object and installed on a different one. Also, if the car gets totaled, the tires do not necessarily have to be destroyed.
  + Composition implies a relationship where the child cannot exist independent of the parent. *Example*: Body ->Blood cell. When the Body object is destroyed the BloodCells get destroyed with it.
  + Dependency is relationship between two objects where changing one may affect the other.

**Data Structures**

# Линейни – към тях спадат списъците, стековете и опашките

# Дървовидни – различни типове дървета

# Речници – хеш-таблици

# Множества

Array:

* Collection of elements that are indexed by a unique ordered index that can be used to retrieve a specific value in the array.
* They are almost always fixed in size, meaning that once they're created they can't grow or shrink.

When should we use?

* + Достъпът по индекс става директно – O(1).
  + Добавянето на нов елемент в масив е много бавна операция (трябва да се задели нов масив с размерност по-голяма с 1 от текущата и да се прехвърлят старите данни в новия масив).
  + Изтриването от масив е много бавна операция (заделяме масив с размер с 1 по-малък от текущия и преместване на всички елементи без изтрития в новия масив.
  + *Масивите трябва да се ползват само когато трябва да обработим фиксиран брой елементи, до които е необходим достъп по индекс.*

Vector (java – ArrayList, python - list):

* Имплементира се с масив.
* Вектора ни дава константно добавяне в края – O(1).
* При добавяне в началото или на произволно място сложността е линейна – O(n).
* При запълване на начално зададения масив се заделя нов масив (обикновено с големина два пъти по-голяма).

Link list:

* A list of elements better connected to each other by a reference from the current element to the next element. Link list can also be doubly linked or the current element reference is the next, and the next reference is the current as well. Linked list generally composed of a head, the starting point, and a series of notes which contain whatever data is being stored in a reference to the next element in the list.

When should we use?

* + The main reason for using linked list is to have the ability to quickly insert and remove data without having to reallocate the entire structure like we would have to do when using an array.
  + Linked list are fast at inserts and removal but slow at getting in the element at a specific index because the linked list must be traversed in order.
  + Inserting - an item into a linked list involves finding the location to do the insertion then creating a new node in changing the reference from the previous node to point to new node and making the reference in the new node point to what the previous node used to point to. The only real guide you hear is to be careful for the case of inserting at the beginning of a linked list or the end.

Stack:

* Stack - common data structure that is like a list where items can only be inserted at the end of the list and the items can only be removed from the end of the list. The key thing to know about a stack is that it is last in first out or LIFO. This means that when you add an element to the stack it is the first item that will be removed from the stack. The last element add to a stack is called the top. Adding an element to the stack is called pushing. Removing an element from a stack is called popping. And stacks often have a peak operation which returns the top of the stack without removing it.

When sould we use?

* Когато е необходимо да реализираме поведението "последен влязъл, пръв излязъл" (LIFO) - искаме елементите в обратен ред.
* При Depth-first-search се използва стек. Ходейки надолу по клоните push-ваме всеки клон и след като стигнем листо pop-ваме от стека.

Queue:

* Queue - instead of having a last and first out rule applied to the elements added to it, the rule is first in first out or FIFO. Queues can be implemented at the list that items are always added to the back off and items are always removed from the front. A queue usually has enqueue operation which is used to add an item to the end of the queue and a dequeue operation which is used to remove an item from the front of the queue.

When should we use?

* Когато е необходимо да реализираме поведението "пръв влязъл, пръв излязъл" (FIFO) - искаме елементите в реда, в който сме ги добавили.
* При Breadth-first-seatch (обхождане в широчина).

Trees:

* Дърво (tree) – рекурсивна структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:
  + Всеки връх може да има 0 или повече преки наследници (деца).
  + Всеки връх има най-много един баща.
  + Всички върхове са достижими от корена - съществува път от корена до всички тях.
* Tree - data structure that looks like a tree, it has a root, it has branches and those branches can have either more branches or leaves. A tree is as similar in some ways to a linked list that could have multiple elements it links to.
* Binary tree - you have a root which is the first node in a tree, a left child node and a right child node. The left and right child nodes work just like references to the next item in a linked list except there can be two next items. You can either take the left path or the right path. Sometimes, a binary tree will be implemented in a way so that the child nodes also have a reference to the parent, similar to a doubly linked list. A leaf node is a node that has no children.
  + Наредено двоично дърво (дърво за търсене, binary search tree) e двоично дърво, в което всеки два от ключовете са сравними и което е организирано, така че за всеки връх да е изпълнено:
    - Всички ключове в лявото му поддърво са по-малки от неговия ключ.
    - Всички ключове в дясното му поддърво са по-големи от неговия ключ.
  + Балансирано двоично дърво – двоично дърво, в което никое листо не е на "много по-голяма" дълбочина от всяко друго листо. Дефиницията на "много по-голяма" зависи от конкретната балансираща схема.
  + Идеално балансирано двоично дърво – двоично дърво, в което разликата в броя на върховете на лявото и дясното поддърво на всеки от върховете е най-много единица.
    - Ако дадено двоично дърво е балансирано, дори и да не е идеално балансирано, то операциите за добавяне, търсене и изтриване на елемент в него са с логаритмична сложност и дори и в най-лошия случай.
* A popular topic in interviews is that of tree traversal. Many questions take the form of write an algorithm to traverse each node in the tree (BFS, DFS→има ги в алгоритмите!)

Graphs:

* Множество от върхове и множество от ребра, които описват връзката между някакви неща.
  + Всяко ребро се задава чрез двойка върхове.
  + Ако ребрата са зададени чрез наредена двойка от върхове, те се наричат дъги. Тогава графът е ориентиран.
  + Ребрата (дъгите) могат да се свържат с етикети от произволен тип – тези етикети се наричат тегла.
* Прост граф – неориентиран, без loops и без multiple edges (повече от един път между два върха).
* Дърво – свързан неориентиран граф без цикли (свързан означава, че всеки връх е свързан към поне един друг връх).

Hash tables:

* Масив с наредени двойки (ключ-стойност).
* Hash-функция приложена върху ключа ни дава хеш-код (число), което обикновено отговаря на индекса на елемента (наредената двойка) в масива.
* Целта на хеширащата функция е да разпредели ключовете равномерно в масива без повторения (колизии – когато два ключа имат един и същ хеш).
* В случай на колизия се използва свързан списък за запазване на елементите.
* Друг вариант е probbing. При линейния probbing се търсят най-близките свободни позиции до тази на колизията.

When should we use?

* Когато искаме бързо да добавяме елементи и да търсите по ключ.

Sets:

* Множествата са колекции, в които няма повтарящи се елементи.
* Друго важно нещо, което отличава множеството от списъците и масивите е, че неговите елементи си нямат номер. Елементите на множеството не могат да бъдат достъпвани по някакъв друг ключ, както е при речниците..

When should we use?

* Когато трябва бързо да добавяме елементи към множество и да проверяваме дали даден елемент е от множеството.

ВЪПРОСИ

# Array:

* Questions involving arrays usually involve doing things like reversing the order of an array, shifting an array so that the values in the array either moved left or right and calculating some value from the elements in array, things like finding the pairs that add up to some value or removing duplicate items.
* Many array questions have brute force answers that can be expensive in terms of space or time, but there's usually a way to solve that array problem that will minimize the amount of space in memory required to solve it. For example, if we want to reverse an array, we could swap the first and the last value then move to the second value and the second to last value and swap those and so on instead of creating a new array and copying each item from the first to the second. One key thing to think about when solving array problems is that many times, the interviewer is interested in seeing if you can solve a problem with an efficient use of space since arrays are often used for that purpose.

# Link List:

* Another common question is to delete from a linked list. Deleting from a linked list is even easier than inserting. The basic process is to find the node before the node that needs to be deleted and to set each reference to point to the node that to be deleted node is referencing. Just be sure again to properly handle the case of the first element in the list and to allocate memory for the node being deleted if you're programming language requires you to manually manage memory.
* Some interviewers will ask you to write an algorithm to insert items in the linked list or remove them, but occasionally, you might run to questions asking you to either reverse the linked list or to determine if there is a cycle or a loop contained in a linked list. If you think you might be asked these kinds of questions for the particular job you're applying for, you should probably practice both of these things and it's a not a bad idea to practice solving either of these problems just to become familiar with linked list in general.

Stacks and Queues:

* Stack and queue questions usually involve explaining the operations of each of the data types or implementing the data types using either an array or a linked list. There's a good idea to try and implement your own queue or stack in your programming language of choice
* Another common question is how to build a queue from a stack, which is pretty simple once you know how. To solve this problem you can use two stacks, one stack to hold the items as they come in or are enqueued and another to hold items in reverse order to be dequeued. When an item is enqueued you simply push it to the enqueue stack. When an item is dequeued you check to see if the dequeue stack is empty. If it is empty you pop each item from the enqueue stack and push it to the dequeue stack until the enqueue stack is empty, then you pop the first item from the dequeue stack. If the dequeue stack was not empty you simply pop the first item from it.

Trees:

* A popular topic in interviews is that of tree traversal. Many questions take the form of write an algorithm to traverse each node in the tree (BFS, DFS).
* You should expect to be asked to create a tree structure especially a binary tree, and be prepared to find the depth of a given tree or the number of elements in the tree. Tree traversal can easily give you an answer to the depth of a tree or the number of elements contained in that tree.

**Algorithms**

There are two major types of algorithms in computer science that are used on the data structures we talked about, sorting and searching.

# Sorting:

* A sorting algorithm is an algorithm used to take a set of data and rearrange it and then order based on the values of that data.
* The most common types of sorting algorithms that are often asked about in interview questions are bubble sort, selection sort, merge sort, and quick sort. Let's briefly talk about each sorting algorithm:
  + Bubble sort / O(n2) / - you take your list of elements and you traverse the list looking at each pair of elements that are next to each other. If they are out of order, you swap the two and repeat all the way through the list. You keep making passes through the list until you have a pass where no elements were swapped. The biggest and smallest elements bubble to the front and back of the list.
  + Selection sort / O(n2) / - the idea is to divide your list into two list, a sorted and then unsorted list. Typically, you just keep track of the index that signifies the boundary between the two sublist. Then you start from the beginning of the unsorted list and traverse the entire list, keeping track of which item is the smallest in that list or largest if you're sorting from largest to smallest. And you simply swap the item with the first item in the unsorted list and move the index that divides the list up by one. Now, it becomes part of the sorted list. You repeat the process until the entire list is sorted.
  + Insertion sort / O(n2) / - the idea is to iterate over each element and grow a sorted list.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

* + Merge sort / O(n.log(n)) / - the idea here is to divide up your list into a sorted list that end up being one element in length which means that they are effectively sorted. Then you keep merging sublist together in the right order until there are no more sublist. The sorting algorithm is usually implemented as a recursive function because it is a divide and conquer type of algorithm.
  + Quick sort / O(n.log(n)), worst case→O(n2)/ - another dividing conquer algorithm. The basic idea is that we first select what's called the pivot. The pivot value will be used to split the list. On one side of the pivot, we'll put the elements that are less than the pivot value. On the other side, we'll put all the elements that are greater. We'll then apply a sort to each half using the same algorithm and combine the sorted halves together. A list with one element is considered sorted.
  + Counting sort / *O*(*n* + *k*) / -
* Most sorting related questions will involve implementing a sorting algorithm, usually a bubble sort or a quick sort. Bubble sort because it is one of the easiest, and quick sort because it is one of the most common.
* Other sorting questions may involve sorting various data structures, most likely arrays or a link list. If you're up against particularly hard interview for a job that is very dependent on sorting, you may also want to familiarize yourself with sorting trees which is a bit harder.
* You may also see questions about how to insert items into an already sorted data structure which usually involves finding the right place and then doing the insert.

# Searching

* Most searching questions will involve starting with a sorted list and involve finding some element in that list using either a linear search or a binary search:
  + Linear search is just going through each element in a list and finding the one you're looking for.
  + Binary search - given you have a sorted list to implement a binary search, we just divide the list in half and determine if the item we are searching for is greater than or less than the value we selected halfway through the list. Of course, if you get lucky, and it is the selected item, you are done right then. We then repeat this process with the half of the list that our item is in until we find it. Each step in the search divides a task in half and brings us closer to the value.
* In addition, you may also be asked to write an algorithm or explain how to search a tree:
  + A sorted binary tree is very easy to search because you can just look at the node value and see if it matches. If it does, you are done. But if it doesn't, then if the value you are looking for is less than a node value, search the left node, otherwise, search the right node.
  + Searching in unsorted tree can be accomplished by visiting each node using the traversal algorithms we talked about previously and looking for the node that matches.
* Graphs:
  + BFS - a tree can be traversed via a breath first traversal by keeping track of all the nodes at each level and exploring the child nodes of each sibling node before moving deeper into the tree.
  + DFS - the basic depth first traversal of a tree involves starting at the root then traversing the left subtree and finally traversing the right subtree. To traverse each chapter you'll apply the same exact logic, visiting its root, then its left subtree and finally its right subtree.
  + Shortest path:
    - Dijkstra
    - A\*
  + Topological sorting /madness pls.. 6AM/

**Design Patterns**

# What is a design pattern?

* A design pattern is a solution to a common software problem that generally occurs in different situations in software development.
* Design patterns tell us how, in the context of a certain problem, we should structure the classes and objects. They do not translate directly into the solution; rather have to be adapted to suit the problem context (it is more of a template for solving a problem, rather than a specific implementation).
* Design Pattern Classifications
  + Creational Patterns – they describe how best an object can be created.
  + Structural Patterns - they describe how objects and classes can work together to achieve larger results.
  + Behavioral Patterns - they talk about interaction between objects.

# What is a singleton pattern?

* Singleton pattern provides a mechanism to limit the number of the instances of the class to one. Thus the same object is always shared by different parts of the code. Singleton can be seen as a more elegant solution to global variable because actual data is hidden behind Singleton class interface.

How singleton patterns are often overused, and can be considered an anti-pattern when used incorrectly?

* + Difficult to test (ако променим инстанцията след това трябва ние да я върнем към предното й положение (ние се грижим за това!))
  + Bad with multhithreading (ако двама използват инстанцията и някой промени нещо (докато единият увеличава каунтър до 1000 а другия го прави = 2) става зле)

# What is a MVC pattern?

* In MVC Design Pattern, the application is divided into three interacting categories known as the Model, the View and the Controller.
* Separation of Concerns - the pattern aims at separating out the inputs to the application (the Controller part), the business processing logic (the Model part) and the output format logic (the View part).
  + Controller associates the user input to a Model and a View
  + Model fetches the data to be presented from persistent storage
  + View deals with how the fetched data is presented to the user

# What is Strategy pattern?

* ...

# What are some design patterns you have used?

* You should be ready for this question with some good examples of design patterns you have used. Instead, think about some of the times you have used design patterns in the past ahead of time. So that you know what to say if you're asked this question.

**JAVA**

# How does the same Java code run on multiple operating systems?

* When you compile Java code, you're actually compiling the code to bytecode. Since that bytecode is interpreted by the JVM. And the JVM has implementations from many different operating systems. That code can be run anywhere the JVM can run.

# What does the final keyword do?

* A final class cannot be subclassed. All methods in a final class are implicitly final.
* A final method cannot be overridden or hidden by subclasses. This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class
* A final variable can only be initialized once, either via an initializer or an assignment statement. If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references is still mutable, if it was originally mutable. Unlike the value of a constant, the value of a final variable is not necessarily known at compile time.

# What is the difference between an interface and an abstract class?

* Abstract classes can have constants, members, method stubs (methods without a body) and defined methods, whereas interfaces can only have constants and methods stubs.
* Methods and members of an abstract class can be defined with any visibility, whereas all methods of an interface must be defined as public (they are defined public by default).
* When inheriting an abstract class, a concrete child class must define the abstract methods, whereas an an abstract class can extend another abstract class and abstract methods from the parent class don't have to be defined.
* Similarly, an interface extending another interface is not responsible for implementing methods from the parent interface. This is because interfaces cannot define any implementation.
* A child class can only extend a single class (abstract or concrete), whereas an interface can extend or a class can implement multiple other interfaces.
* A child class can define abstract methods with the same or less restrictive visibility, whereas a class implementing an interface must define the methods with the exact same visibility (public).

# Static vs Non Static:

* A static method in Java belongs to class, which means you can call that method by using class name, you don't need to create any object to access these method, which is what you need to do to access non static method of a class.
* Static method are treated differently by compiler and JVM than non static methods, static methods are bonded during compile time, as opposed to binding of non static method, which happens at runtime.
* You can not access non static members inside static context, which means you can not use non static variables inside static methods, you can not call non static methods from static ones, all those will result in compile time error. But reverse is fine i.e. you can access static variables or call static method from a non static method without any compile time error.
* You can't override static method in Java, they are bonded during compile time using static binding. Though you can create a similar static method in sub class, that is know as method hiding in Java.

*Example:* Each individual pizza piece is an instance of the Pizza class. A method like addTopping() is an instance (non-static) method because it operates on an individual instance of the class. Static methods (and variables) are not associated with any particular instance, they are associated with the class. If you wanted to keep a count of how many pizzas you had made, you could use a static variable 'count' on the Pizza class.

**Additional Stuff**

# Strong cohesion – отговорностите и задачите на една единица код са свързани помежду си и се стремят да решат общ проблем → нещо, което винаги се стремим.

# Loose coupling (функционална независимост) – промяната в имплементацията на един компонент не се отразява на другите, с които той общува (не трябва да разчитаме на вътрешни характеристики на компонентите)

# Сложности:

* Сложност на алгоритъм е груба оценка на броя стъпки, които алгоритъмът ще направи в зависимост от обема на входните данни.

Видове сложности:

* Константна / O(1) / - За извършване на дадена операция са необходими константен брой стъпки (примерно 1, 5, 10 или друго число) и този брой не зависи от обема на входните данни.
* Логаритмична / O(log(N)) / - за извършване на дадена операция върху N елемента са необходими брой стъпки от порядъка на log(N), където основата на логаритъма е най-често 2.
* Линейна / O(N) / - за извършване на дадена операция върху N елемента са необходими приблизително толкова стъпки, колкото са елементите.
* O(n\*log(n)) - за извършване на дадена операция върху N елемента са необходими приблизително N\*log(N) стъпки.
* Квадратична / O(n2) / - за извършване на дадена операция са необходими N 2 на брой стъпки, където N характеризира обема на входните данни.
* Експоненциална - за извърпване на дадена операция или изчисление са необходими брой стъпки, който е в експоненциалназависимост спрямо размера на входните данни.



# HTTP:

* Информацията се предава под формата на текст.
* Архитектура → един сървър, който обслужва много клиенти.
* При HTTP - сървъра не може да отговори без да бъде питан.
* IP:PORT/... port 80 → стандартен за http (през https port-а е 443)

(SOCKET)/Resource

Заявки:

* GET – дай ми някакъв ресурс.
* POST – вземи това и го запази на сървъра.

# SOLID:

* Single responsibility:
  + Стараем се всеки един модул, клас или метод да има точно една ясна цел и тя да бъде максимално енкапсулирана в класа. Не трябва да имаме повече от една причина даден клас да се промени (ако видим много на брой места, където може да правим промени, най-вероятно не сме си разделили класовете и методите)
  + Strong cohesion
  + Loose coupling

Нарушения:

* + - Обекти, които се принтират
    - Обекти, които се запазват / load-ват

Решения:

* + - Изнасяме притирането и запазването (сейването)
* Open-Closed:
  + Трябва лесно да можем да добавяме нови функционалности без да променяме вътрешно класа.

Как да променим държанието без да променяме кода?

* + - Зависейки винаги от абстракции (интерфейси, базови класове), а не от имплементации.
    - Да не се лимитираме от това по колко начина един код може да бъде написан.

Нарушения:

* + - Ако след всяка промяна трябва да ретестваме.
    - При промяна в един модул се чупи друг.

Решения:

* + - Добавяме нови класове → нищо не депендва от тях.
* Liskov substitution:
  + Всеки един тип данни трябва да е лесно заменяем от неговите базови типове (наследниците трябва да могат да заместват базовите класове)
  + Наследниците не трябва да премахват поведение на базовия клас или променливите му, които биха променили поведението му.

Нарушения:

* + - Ако имаме Type checking → сме счупили принципа
* Interface segregation:
  + Когато клиент има нужда от два метода, а ние му даваме интерфейс с 50 неща, очевидно сме сгрешили (клиента не трябва да бъде форсиран да депендва на някакви методи/пропъртита, от които няма нужда).
  + Трябва да имплементираме само нещата, които ни трябват.

Нарушения:

* + - Неимплементирани методи.
    - Използване само малка част от класа.
* Dependency inversion:
  + Всички high-end модули НЕ трябва да депендват на low-end модулите. Те трябва да депендват на абстракция, не на имплементация.
  + Инжектираме в класовете това, коети ни трябва.
  + Не трябва да скриваме депендънситата (от какво има нужда един клас) – когато класа има нужда от 5 неща, човека отвън му ги дава.
  + Don't call us, we will call you.

# Concurrency:

* The definition of concurrency is when several things are happening in the same time period. It doesn't necessarily mean that they're happening in parallel but they could be.
* Locks:
  + A lock is simply some way of preventing multiple threads from accessing a piece of data or other resource. It is used to prevent issues in concurrent applications or multiple threads of execution are manipulating the same data impossibly causing corruption of that data. A common question is something along the lines of, what is the reason for using a lock? The answer is that you want to prevent one thread from accessing a piece of data or resource while another thread is accessing that data. A common example is incrementing a value. If threads have to first read the value then add one to the value and then store back that value, you can have a problem if multiple threads read the value before a thread updates a value. Say that two threads read the value, and that is set to two. Each one wants to increment the value, both of them will set the value to three which is not correct. A lock can force the second thread that tried to read the value to wait until the first thread is done updating the value.
* Deadlocks:
  + Deadlocks happen when there is a circular dependency and the resources needing to acquire locks. If thread one needs a lock and resource A to continue it is going to wait until it can get that lock. But if thread two happens to currently have resource A locked, and is waiting on getting a lock for resource B, but thread one has that lock nothing can ever happen unless someone gives up a lock. Deadlocks can be fairly complex as they spend many threads in resources. A common interview question is to ask what a deadlock is or how to create or prevent one?
* Race Conditions:
  + Race conditions happen when the behavior of a system is in determinant because it depends on the timing of thread execution in order which could change from run to run. In the example we used earlier to explain a lock where two threads were trying to increment the same piece of data, we had a race condition because the behavior depended on how the threads happen to execute.