|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Kotlin | <chare> | 1 | <pext> | <starting>Kotlin</heading> Kotlin is the latest JVM programming language from the JetBrains. Google has made it the official language for Android Development  along with Java. </ending> | </end> |
| <hitle> | Kotlin pros over Java, kotlin vs java | <chare> | 2 | <pext> | 1. Kotlin has null safely which ensures that our program will not crash  due to null pointer exception  2. Kotlin has coroutines which is a lightweight, simple and powerful way  of writing code which runs asynchronously.  3. Kotlin has extension functions which allows us to add more functionality  to existing class without inheriting  4. Kotlin has scope functions make our code concise  5. Kotlin is interoperable with Java  6. Kotlin has smart cast  7. Kotlin has when expression which is much flexible than switch statement in Java  8. Data classes are very handy to create POJO classes.  9. Kotlin has no boilerplate code. | </end> |
| <hitle> | val vs var, const vs val | <chare> | 2 | <pext> | 1. val vs var  val variables cannot be changed. They’re like final modifiers in Java. A var can be reassigned. The reassigned value must be of the same data type.  2. const vs val  By default val properties are set at runtime.  Adding a const modifier on a val would make a compile-time constant.  A const cannot be used with a var or on its own.  A const is not applicable on a local variable. | </end> |
| <hitle> | const vs val | <chare> | 2 | <pext> | By default val properties are set at runtime.  Adding a const modifier on a val would make a compile-time constant.  A const cannot be used with a var or on its own.  A const is not applicable on a local variable. | </end> |
| <hitle> | Null Safety ? Vs Nullable Types | <chare> | 2 | <pext> | <starting>Null Safety ? Vs Nullable Types</heading> Kotlin puts a lot of weight behind null safety which is an approach  to prevent the dreaded Null Pointer Exceptions by using nullable types  which are like String?, Int?, Float? etc. These act as a wrapper type and can hold null values. A nullable value cannot be added to another nullable or basic type of value. To retrieve the basic types we need to use safe calls  that unwrap the Nullable Types. If on unwrapping, the value is null we can choose to ignore  or use a default value instead. The Elvis Operator is used to safely unwrap the value from the Nullable. It’s represented as ?: over the nullable type. The value on the right hand side would be used if the nullable type holds a null. </ending> | </end> |
| <hitle> | kotlin primitive types | <chare> | 2 | <pext> | <starting>kotlin primitive types</heading> At the language level, we cannot use the above-mentioned types. But the JVM bytecode that’s compiled does certainly have them. </ending> | </end> |
| <hitle> | !! vs ?. (null check operator vs safe call operator) | <chare> | 2 | <pext> | <starting>!! vs ?. (null check operator vs safe call operator)</heading> !! is used to force unwrap the nullable type to get the value. If the value returned is a null, it would lead to a runtime crash. Hence a !! operator should be only used when you’re absolutely sure  that the value won’t be null at all. Otherwise, you’ll get the dreaded null pointer exception. On the other hand, a ?. is an Elvis Operator that does a safe call. We can use the lambda expression let on the nullable value  to unwrap safely as shown below. </ending> | </end> |
| <hitle> | == vs === | <chare> | 2 | <pext> | <starting>== vs ===</heading> == is used to compare the values are equal or not. === is used to check if the references are equal or not. </ending> | </end> |
| <hitle> | visibility modifier kotlin | <chare> | 2 | <pext> | <starting>visibility modifier kotlin</heading> 1. public Public modifier is accessible from anywhere in the project workspace.  If no access modifier is specified, then by default it will be in the public scope. This is the default visibility modifier. 2. internal Internal is a newly added modifier in Kotlin.  If anything is marked as internal, then the specific field will marked  as the internal field.  An Internal package is visible only inside the module  under which it is implemented.  An internal class interface is visible only by other class present  inside the same package or the module. 3. protected Protected is another access modifier for Kotlin, which is currently not available  for top level declaration like any package cannot be protected.  A protected class or interface or properties or function is visible  to the class itself and it's subclasses only. 4. private The classes, methods, packages and other properties  can be declared with a private modifier.  This modifier has almost the exact opposite meaning of public  which means a private member can not be accessed outside of its scope.  Once anything is declared as private, then it can be accessible  within its immediate scope only.  </ending> | </end> |
| <hitle> | constructor type kotlin | <chare> | 2 | <pext> | <starting>constructor kotlin</heading> Constructors in Kotlin are of two types:  \*\*Primary\*\* - These are defined in the class headers. They cannot hold any logic. There's only one primary constructor per class.  \*\*Secondary\*\* - They're defined in the class body. They must delegate to the primary constructor if it exists. They can hold logic. There can be more than one secondary constructors. </ending> | </end> |
| <hitle> | init block | <chare> | 2 | <pext> | <starting>init block</heading> `init` is the initialiser block in Kotlin. It's executed once the primary constructor is instantiated. If you invoke a secondary constructor,  then it works after the primary one as it is composed in the chain. </ending> | </end> |
| <hitle> | string interpolation kotlin | <chare> | 2 | <pext> | <starting>string interpolation kotlin</heading> String interpolation is used to evaluate string templates. We use the symbol $ to add variables inside a string. ``` val name = "Journaldev.com" val desc = "$name now has Kotlin Interview Questions too. ${name.length}" ``` Using `{}` we can compute an expression too. </ending> | </end> |
| <hitle> | argument type in constructor kotlin | <chare> | 2 | <pext> | <starting>argument type in constructor kotlin</heading> By default, the constructor arguments are `val` unless explicitly set to `var`. </ending> | </end> |
| <hitle> | new kotlin | <chare> | 2 | <pext> | <starting>new kotlin</heading> \*\*NO\*\*. Unlike Java, in Kotlin, new isn't a keyword. We can instantiate a class in the following way:  ``` class A var a = A() val new = A() </ending> | </end> |
| <hitle> | when switch kotlin | <chare> | 2 | <pext> | <starting>when switch kotlin</heading> when is the equivalent of `switch` in `Kotlin`. The default statement in a when is represented using the else statement. `when` statments have a default break statement in them. </ending> | </end> |
| <hitle> | data class | <chare> | 2 | <pext> | <starting>data class</heading> The Data class is a simple class that holds data and provides typical functions. To declare a class as a data class, use the data keyword.  The following functions are automatically derived by the compiler  for the data classes: 1. equals() - The equals() function returns true  if two objects have the identical contents. It operates similarly to "==,"  although for Float and Double values it works differently. 2. hashCode() - The hashCode() function returns the object's hashcode value. 3. copy() - The copy() function is used to duplicate an object,  changing only a few of its characteristics while leaving the rest unaltered. 4. toString() - This function returns a string  containing all of the data class's parameters. To ensure consistency, data classes must meet the following requirements: 1. At least one parameter is required for the primary constructor. 2. val or var must be used for all primary constructor parameters. 3. Abstract, open, sealed, or inner data classes are not possible. 4.Only interfaces may be implemented by data classes. </ending> | </end> |
| <hitle> | Destructuring Declaration | <chare> | 2 | <pext> | <starting>Destructuring Declaration</heading> Destructuring Declarations is a smart way to assign multiple values to variables from data stored in objects/arrays. Within paratheses, we've set the variable declarations. Under the hood, destructuring declarations create component functions for each of the class variables. In Kotlin, destructuring is a convenient way  to extract multiple values from data stored in objects and Arrays. It can be used in locations that receive data. It is used because sometimes, it is convenient to destructure an object  into several variables. For Example: val (name, age) = developer  Now, we can use name and age independently as follows: println(name)  println(age)  </ending> | </end> |
| <hitle> | inline vs infix functions | <chare> | 2 | <pext> | <starting>inline vs infix functions</heading> [Inline functions] are used to save us memory overhead by preventing  object allocations for the anonymous functions/lambda expressions called. Instead, it provides that functions body to the function that calls it at runtime. This increases the bytecode size slightly but saves us a lot of memory.  [infix functions] on the other are used to call functions  without parentheses or brackets. Doing so, the code looks much more like a natural language.  </ending> | </end> |
| <hitle> | Singleton kotlin | <chare> | 2 | <pext> | <starting>Singleton kotlin</heading> To use the singleton pattern for our class we must use the keyword `object`. An `object` cannot have a constructor set. We can use the init block inside it though. </ending> | </end> |
| <hitle> | Static equivalent kotlin | <chare> | 2 | <pext> | <starting>static kotlin</heading>  Can we use static in kotlin? \*\*NO\*\*. Kotlin doesn't have the static keyword. To create static method in our class we use the `companion object`. Following is the Java code: ``` class A {  public static int returnMe() { return 5; } } ``` The equivalent Kotlin code would look like this: ``` class A {  companion object {  fun a() : Int = 5  } } ``` To invoke this we simply do: `A.a()`. </ending> | </end> |
| <hitle> | Immutable vs Mutable Variables | <chare> | 2 | <pext> | <starting>Immutable vs Mutable Variables</heading> Immutable variables are also known as read-only variables. They are declared using the val keyword. Once these variables have been declared, we cannot change their values. Mutable Variables - In a mutable variable,  the value of the variable can be changed. We use the keyword “var” to declare such variables. </ending> | </end> |
| <hitle> | null safety ? | <chare> | 2 | <pext> | <starting>null safety ?</heading> If a program throws NullPointerExceptions at runtime  it might result in application failure or system crashes. If the Kotlin compiler finds a null reference it throws a NullPointerException. The Kotlin type system distinguishes between references  that can hold null (nullable references) and those that cannot  (non-null references). Null cannot be stored in a String variable. We get a compiler error if we try to assign null to the variable.  var a: String = "interview" a = null // results in compilation error If we want the above string to be able to hold null value as well,  we can declare it of type nullable using the ‘?’ operator  after the String keyword as follows : var a: String? = "interview" a = null // no compilation error Kotlin provides Safe Call (?.), Elvis (?:) and Not Null Assertion (!!) operators  which define what needs to be done in case of a null encounter. This makes the code more reliable and less prone to errors. Thus, Kotlin enforces null safety by having nullable,  non-nullable type variables and the different operators  to tackle null encounters. </ending> | </end> |
| <hitle> | Safe Call operator ( ?. ) | <chare> | 2 | <pext> | <starting>Safe Call operator ( ?. )</heading> Safe Call operator ( ?. ) - Null comparisons are trivial,  but the number of nested if-else expressions can be exhausting. So, in Kotlin, there's a Safe call operator,?,  that simplifies things by only doing an action  when a specified reference holds a non-null value. It allows us to use a single expression  to perform both a null check and a method call. </ending> | </end> |
| <hitle> | Elvis Operator ( ?: ) | <chare> | 2 | <pext> | <starting>Elvis Operator ( ?: )</heading> Elvis Operator ( ?: ) - When the original variable is null,  the Elvis operator is used to return a non-null value or a default value. In other words, the elvis operator returns the left expression if it is not null,  otherwise, it yields the right expression. Only if the left-hand side expression is null is the right-hand side evaluated. Furthermore, on the right side of the Elvis operator,  we may use throw and return expressions,  which is particularly handy in functions. As a result, instead of returning a default value  on the right side of the Elvis operator, we can throw an exception. </ending> | </end> |
| <hitle> | null check operator ( !! ) | <chare> | 2 | <pext> | <starting> null check operator ( !! )</heading> null check operator ( !! ) - If the value is null,  the not null assertion (!!) operator changes it to a non-null type  and throws an exception. Anyone who wants a NullPointerException can ask for it explicitly  with this operator. </ending> | </end> |
| <hitle> | function extension | <chare> | 2 | <pext> | <starting>function extension</heading> In Kotlin, we can add or delete method functionality using extensions,  even without inheriting or altering them. Extensions are statistically resolved. It provides a callable function that may be invoked with a dot operation,  rather than altering the existing class. Function Extension - Kotlin allows users to specify a method  outside of the main class via function extension. </ending> | </end> |
| <hitle> | Companion Object kotlin | <chare> | 2 | <pext> | <starting>Companion Object kotlin</heading> In some languages, such as Java, the static keyword is used  to declare class members and utilise them without creating an object,  i.e. by simply calling them by their class name. In Kotlin, there is nothing called the “static” keyword. So, if we want to achieve the functionality of static member functions, we use the companion objects. This is also referred to as Object Extension.  We must use the companion keyword in front of the object definition  to construct a companion object. </ending> | </end> |
| <hitle> | open and public kotlin | <chare> | 2 | <pext> | <starting>open and public kotlin</heading> The keyword “open” refers to the term "open for expansion". The open annotation on a class is the polar opposite  of the final annotation in Java: it allows others to inherit from it. By default, a class cannot be inherited in Kotlin. In Kotlin, an open method signifies that it can be overridden,  whereas it cannot be by default. Instead, any methods in Java can be overridden by default. In Kotlin, all the classes are final by default. If no visibility modifier is specified, the public is used by default, which means our declarations will be accessible everywhere  inside the program. </ending> | </end> |
| <hitle> | val mutableList vs var immutableList | <chare> | 2 | <pext> | <starting>val mutableList vs var immutableList</heading> 1. Immutable lists are frequently preferred for a variety of reasons: 1-1. They promote functional programming,  in which state is passed on to the next function,  which constructs a new state based on it, rather than being altered. This is evident in Kotlin collection methods like map, filter, reduce, and so forth. 1-2. It's often easier to understand and debug software that doesn't have  any side effects  (you can be sure that the value of an object will always be the one  at its definition). 1-3. Because no write access is required in multi-threaded systems,  immutable resources cannot induce race conditions. 2. However, there are some disadvantages of using immutable lists as well. They are as follows : 2-1. Copying large collections simply to add/remove a single piece  is very expensive. 2-2. When you need to alter single fields frequently,  immutability can make the code more difficult. Data classes in Kotlin provide a built-in copy() method that allows you  to clone an instance while changing only part of the fields' values. </ending> | </end> |
| <hitle> | lateinit | <chare> | 2 | <pext> | <starting>lateinit</heading> lateinit is an abbreviation for late initiation. If you don't want to initialize a variable in the constructor  and instead want to do it later, and you can guarantee the initialization  before using it, use the lateinit keyword to declare that variable. It won't start allocating memory until it's been initialized. Lateinit cannot be used for primitive type attributes like Int, Long, and so on. Because the lateinit variable will be initialized later, you cannot use val. When a lateinit property is accessed before it has been initialized,  a special exception is thrown that explicitly identifies the property  and the fact that it hasn't been initialized.  A lateinit property must be initialized at some point before it is accessed  or it will throw an UninitializedPropertyAccessException error.  There are a few scenarios in which this is particularly useful, for example: 1. Variables that are initialized in lifecycle methods in Android; 2. Using Dagger for DI: injected class variables are initialized  outside of the constructor and independently; 3. Setup for unit tests: in a @Before - annotated function,  test environment variables are initialized; 4. Annotations in Spring Boot (for example, @Autowired) </ending> | </end> |
| <hitle> | lazy | <chare> | 2 | <pext> | <starting>lazy</heading> There are some classes whose object initialization is so time-consuming  that it causes the entire class creation process to be delayed. Lazy initialisation helps in such problems. When we declare an object using lazy initialisation,  the object is initialised only once when the object is used. If the object is not used throughout, the object is not initialised. This makes the code more efficient and faster.  </ending> | </end> |
| <hitle> | lazy vs lateinit | <chare> | 2 | <pext> | <starting>lazy vs lateinit</heading> Both are used to delay the property initializations in Kotlin. `lateinit` is a modifier used with var and is used to set the value to the var  at a later point. `lazy` is a method or rather say lambda expression. It's set on a val only. The val would be created at runtime when it's required.  1. lazy, The main purpose is to delay the initialisation to a later point in time.  1-1. lateinit, The main purpose is to initialise an object  only when it is used at a later point in time. Also, a single copy of the object is maintained throughout the program.  2. lazy, It's possible to initialise the object from anywhere in the program. 2-1. lateinit, Only the initializer lambda can be used to initialise it. 3. lazy, Multiple initializations are possible in this case. 3-1. lateinit, Only a single initialisation is possible in this case. 4. lazy, It's not thread-safe. In a multi-threaded system,  it is up to the user to correctly initialise. 4-1. lateinit, Thread-safety is enabled by default,  ensuring that the initializer is only called once. 5. lazy, It works only with var. 5-1. lateinit, It works only with val. 6. lazy, The isInitialized method is added to verify  if the value has previously been initialised. 6-1. lateinit, It is impossible to uninitialize a property. 7. lazy, Properties of primitive types are not allowed 7-1. lateinit, Allowable on primitive type properties.  There are a few easy principles to follow when deciding whether to use lateinit or lazy initialisation for property initialization: 1. Use lateInit if properties are mutable (i.e., they may change later). 2. Use lateinit if properties are set externally (for example, if you need to pass in an external variable to set it). There is still a way to use lazy, but it isn't as obvious. 3. If they're only meant to be initialised once and shared by everybody, and they're more internally set (depending on a class variable), then lazy is the way to go. We could still use lateinit in a tactical sense, but utilising lazy initialisation would better encapsulate our initialization code. </ending> | </end> |
| <hitle> | scope function kotlin | <chare> | 2 | <pext> | <starting>scope function kotlin</heading>  Scoped functions are functions that execute a block of code  within the context of an object.  scope functions provide a way to give temporary scope to the object  under consideration where specific operations can be applied to the object  within the block of code, resulting in a clean and concise code.  When we use a lambda expression to call these functions on an object,  temporary scope is created. These functions are referred to as Scope functions. The object of these functions can be accessed without knowing its name.  Scope functions make code more clear, legible, and succinct,  which are key qualities of the Kotlin programming language. 1. let:-  Context object: it  Return value: lambda result The let function is frequently used for null safety calls. For null safety, use the safe call operator(?.) with ‘let'. It only runs the block with a non-null value. 2. apply:- Context object: this Return value: context object “Apply these to the object,” as the name suggests. It can be used to operate on receiver object members, primarily to initialise them. 3. with:- Context object: this Return value: lambda result When calling functions on context objects without supplying the lambda result,  ‘with' is recommended. 4. run:- Context object: this  Return value: lambda result The ‘run' function is a combination of the ‘let' and ‘with' functions. When the object lambda involves both initialization and computation  of the return value, this is the method to use. We can use run to make null safety calls as well as other calculations. 5. also:- Context object: it Return value: context object It's used when we need to do additional operations  after the object members have been initialised.  ------------------difference between scope functions----------------  1. The way to refer to the context object.  Either “this” or “it”  2. The return value  Either “Context object” or the “lambda result”  -----------whentouse whattouse scope functions-------------------  1. with: If you want to operate on a non-null object  2. let: If you want to just excute lambda expression on a nullable object  and avoid NullPointerException  3. run: If you want to operate on a nullable object, excute lambda expression  and avoid NullPointerException  4. apply: If you want to initialize or configure an object  5. also: If you want to do some additional object configuratioin or operations | </end> |
| <hitle> | let:- | <chare> | 2 | <pext> | <starting>let:-</heading> Context object: it  Return value: lambda result The let function is frequently used for null safety calls. For null safety, use the safe call operator(?.) with ‘let'. It only runs the block with a non-null value. </ending> | </end> |
| <hitle> | apply:- | <chare> | 2 | <pext> | <starting>apply:-</heading> Context object: this Return value: context object “Apply these to the object,” as the name suggests. It can be used to operate on receiver object members,  primarily to initialise them. </ending> | </end> |
| <hitle> | with:- | <chare> | 2 | <pext> | <starting>with:-</heading> Context object: this Return value: lambda result When calling functions on context objects without supplying the lambda result,  ‘with' is recommended. </ending> | </end> |
| <hitle> | run:- | <chare> | 2 | <pext> | <starting>run:-</heading> Context object: this  Return value: lambda result The ‘run' function is a combination of the ‘let' and ‘with' functions. When the object lambda involves both initialization and computation  of the return value, this is the method to use. We can use run to make null safety calls as well as other calculations. </ending> | </end> |
| <hitle> | also:- | <chare> | 2 | <pext> | <starting>also:-</heading> Context object: it Return value: context object It's used when we need to do additional operations  after the object members have been initialised. </ending> | </end> |
| <hitle> | sealed class | <chare> | 2 | <pext> | Sealed Classes allow us to fix type hierarchies  and forbid from creating new subclasses.  They are useful when we have a very strict inheritance hierarchy,  with a specific set of possible subclasses and no others.  sealed classes can have subclasses in all files of the same compilation unit  and the same package.  Sealed classes are also implicitly abstract.  Sealed classes can have fields and methods defined in them,  including both abstract and implemented functions.  This means that we can have a base representation of the class  and then adjust it to suit the subclasses.  ----------whentouse sealed class-----------  Sealed classes are designed to be used  when there are a very specific set of possible options for a value,  and where each of these options is functionally different.  We can use it to implement API call’s response orState Machine.  Anytime we have multiple options and they only differ in the meaning of the data,  we may be better using sealed class instead of Enum Classes. | </end> |
| <hitle> | backing field | <chare> | 2 | <pext> | A backing field is an auto-generated field for any property  that may only be used inside accessors (getter or setter)  and will only be present if it utilizes the default implementation of  at least one of the accessors, or if a custom accessor refers to it  through the field identifier. This backing field is used to avoid an accessor's recursive call,  which would result in a StackOverflowError. Fields are not allowed in Kotlin classes. When employing custom accessors, however,  it is occasionally required to have a backing field. Kotlin includes an automatic backing field for these purposes,  which may be accessed by the field identifier.  var counter = 0 // the initializer assigns the backing field directly  set(value) {  if (value >= 0)  field = value  // counter = value // ERROR StackOverflow: Using actual name 'counter' would make setter recursive  } | </end> |
| <hitle> | Backing property | <chare> | 2 | <pext> | If we want to do something that does not fit into implicit backing field scheme,  we can always fall back to having a backing property.  private var \_table: Map<String, Int>? = null  public val table: Map<String, Int>  get() {  if (\_table == null) {  \_table = HashMap() // Type parameters are inferred  }  return \_table ?: throw AssertionError("Set to null by another thread")  } | </end> |
| <hitle> | fold vs reduce | <chare> | 2 | <pext> | <starting>fold vs reduce</heading> 1. fold takes an initial value, and the first invocation of the lambda you pass to it will receive that initial value and the first element of the collection as parameters. listOf(1, 2, 3).fold(0) { sum, element -> sum + element } The first call to the lambda will be with parameters 0 and 1. Having the ability to pass in an initial value is useful if you have to provide some sort of default value or parameter for your operation. 2. reduce doesn't take an initial value, but instead starts with the first element of the collection as the accumulator (called sum in the following example) listOf(1, 2, 3).reduce { sum, element -> sum + element } The first call to the lambda here will be with parameters 1 and 2. </ending> | </end> |
| <hitle> | constant kotlin | <chare> | 2 | <pext> | <starting>constant kotlin</heading> In Kotlin, if you want to create the local constants which are supposed  to be used with in the class then you can create it like below: val MY\_CONSTANT\_1 = "Constants1" // or  const val MY\_CONSTANT\_2 = "Constants2" Like val, variables defined with the const keyword are immutable. The difference here is that const is used for variables that are known at compile-time. Also avoid using companion objects. Behind the hood, getter and setter instance methods are created for the fields  to be accessible. Calling instance methods is technically more expensive  than calling static methods. Instead define the constants in object: object DbConstants {  const val TABLE\_USER\_ATTRIBUTE\_EMPID = "\_id"  const val TABLE\_USER\_ATTRIBUTE\_DATA = "data" } </ending> | </end> |
| <hitle> | IntArray vs Array<Int> | <chare> | 2 | <pext> | <starting>IntArray vs Array<Int></heading> Array<Int> is an Integer[] under the hood, while IntArray is an int[]. This means that when you put an Int in an Array<Int>,  it will always be boxed (specifically, with an Integer.valueOf() call). In the case of IntArray, no boxing will occur,  because it translates to a Java primitive array. So no, we can't use them interchangeably. </ending> | </end> |
| <hitle> | Companion Object purpose | <chare> | 2 | <pext> | <starting>Companion Object purpose</heading> Unlike Java or C#, Kotlin doesn’t have static members or member functions. If you need to write a function that can be called without having  a class instance but needs access to the internals of a class,  you can write it as a member of a companion object declaration  inside that class. class EventManager {  companion object FirebaseManager {  }  } val firebaseManager = EventManager.FirebaseManager The companion object is a singleton. The companion object is a proper object on its own,  and can have its own supertypes - and you can assign it to a variable  and pass it around. If you're integrating with Java code and need a true static member,  you can annotate a member inside a companion object with @JvmStatic. </ending> | </end> |
| <hitle> | not-null assertion operator, double-bang !! | <chare> | 2 | <pext> | <starting>not-null assertion operator !!</heading> The not-null assertion operator !! converts any value to a non-null type  and throws a KotlinNullPointerException exception if the value is null. Consider: fun main(args: Array<String>) {  var email: String?  email = null  println(email!!) } This operator should be used in cases where the developer is guaranteeing  – it allows you to be 100% sure that its value is not null. </ending> | </end> |
| <hitle> | suspending vs blocking | <chare> | 2 | <pext> | <starting>suspending vs blocking</heading> 1. A blocking call to a function means that a call to any other function,  from the same thread, will halt the parent’s execution. Following up, this means that if you make a blocking call  on the main thread’s execution, you effectively freeze the UI. Until that blocking calls finishes, the user will see a static screen,  which is not a good thing. 2. Suspending doesn’t necessarily block your parent function’s execution. If you call a suspending function in some thread,  you can easily push that function to a different thread. In case it is a heavy operation, it won’t block the main thread. If the suspending function has to suspend, it will simply pause its execution. This way you free up its thread for other work. Once it’s done suspending, it will get the next free thread from the pool,  to finish its work. </ending> | </end> |
| <hitle> | List vs Array kotlin | <chare> | 2 | <pext> | <starting>List vs Array kotlin</heading> 1. Array is implemented using Array<T> class. 1-1. List<T> or MutableList<T> interfaces are used to implement a List in Kotlin. 2. Array<T> is mutable, i.e., the values can be changed. 2-1. List<T> is immutable in nature. In order to create a mutable list, MutableList<T> interface needs to be used. 3. Array is of fixed size. It cannot increase and decrease in size. 3-1. MutableList<T> do have 'add' and 'remove' functions in order to increase or decrease the size of the MutableList. 4. Array is used for better performance, as array is optimized for different primitive data types such as IntArray[], DoubleArray[]. 4-1. List is used for better accessibility in the code. As the size is dynamic in nature, hence good memory management. </ending> | </end> |
| <hitle> | static method kotlin | <chare> | 2 | <pext> | <starting>static method kotlin</heading> 1. By placing the function in the companion object. class Foo {  public static int a() { return 1; } } will become: class Foo {  companion object {  fun a() : Int = 1  } } // to run Foo.a(); 2. Another way is to solve most of the needs for static functions  with package-level functions. They are simply declared outside a class in a source code file. The package of a file can be specified at the beginning of a file  with the package keyword. Under the hood these "top-level" or "package" functions  are actually compiled into their own class. +++++++++++++++++++++++++++++++++++++++++++++ In the above example, the compiler would create a class FooPackage with all of the top-level properties and functions, and route all of your references to them appropriately. Consider: package foo fun bar() = {} usage: import foo.bar </ending> | </end> |
| <hitle> | Unit-returning | <chare> | 2 | <pext> | <starting>Unit-returning</heading> fun printHello(name : String?) : Unit {   if (name != null)   print("Hello, $name!")   else   print("Hi there!")   // We don't need to write 'return Unit.VALUE' or 'return', although we could  } The purpose is the same as Java's void. Only Unit is a proper type,  so it can be passed as a generic argument etc. Why we don't call it "Void": because the word "void" means "nothing",  and there's another type, Nothing, that means just "no value at all",  i.e. the computation did not complete normally  (looped forever or threw an exception). We could not afford the clash of meanings. Why Unit has a value (i.e. is not the same as Nothing):  because generic code can work smoothly then. If you pass Unit for a generic parameter T,  the code written for any T will expect an object,  and there must be an object, the sole value of Unit. How to access that value of Unit: since it's a singleton object, just say Unit. UNIT actually contains valuable information, it basically just means "DONE". It just returns the information to the caller, that the method has been finished. </ending> | </end> |
| <hitle> | open kotlin | <chare> | 2 | <pext> | <starting>open kotlin</heading> In Kotlin, the classes and functions are final by default. So, it is not possible to inherit the class or override the functions. To achieve this,  we need to use the open keyword before the class and function. </ending> | </end> |
| <hitle> | safe calls(?.) vs null check(!!) | <chare> | 2 | <pext> | <starting>safe calls(?.) vs null check(!!)</heading> The safe call operator ?. is used to check if the variable's value is null or not.  If it is null, then null will be returned otherwise it will return the desired value. If you want to throw NullPointerException when the variable's value is null,  you can use the null check or !! Operator. </ending> | </end> |
| <hitle> | when pros over switch | <chare> | 2 | <pext> | <starting>when pros over switch</heading> 1. two or more choices: when(number) {   1 -> println("One")   2, 3 -> println("Two or Three")   4 -> println("Four")   else -> println("Number is not between 1 and 4")  }  2. "when" without arguments: when {   number < 1 -> print("Number is less than 1")   number > 1 -> print("Number is greater than 1")  }  3: Any type passed in "when": fun describe(obj: Any): String =   when (obj) {   1 -> "One"   "Hello" -> "Greeting"   is Long -> "Long"   !is String -> "Not a string"   else -> "Unknown"   }  4:Smart casting: when (x) {   is Int -> print("X is integer")   is String -> print("X is string")  }  5:Ranges: when(number) {   1 -> println("One") //statement 1   2 -> println("Two") //statement 2   3 -> println("Three") //statement 3   in 4..8 -> println("Number between 4 and 8") //statement 4   !in 9..12 -> println("Number not in between 9 and 12") //statement 5   else -> println("Number is not between 1 and 8") //statement 6  }  </ending> | </end> |
| <hitle> | ternary operator kotlin | <chare> | 2 | <pext> | <starting>ternary operator kotlin</heading> No. In Kotlin, we don't have a ternary operator like Java,  but we can use the functionality of the ternary operator by using if-else  or Elvis operator. </ending> | </end> |
| <hitle> | interoperable | <chare> | 2 | <pext> | <starting>interoperable</heading> Kotlin is interoperable with Java because it uses JVM bytecode. It provides the facility to compile it directly to bytecode that helps to achieve  faster compile-time and makes no difference between Java and Kotlin for JVM. </ending> | </end> |
| <hitle> | extension methods | <chare> | 2 | <pext> | <starting>extension methods</heading> Following are some extension methods that Kotlin provides to java.io.File: 1. bufferedReader(): It is used for reading the contents of a file  into BufferedReader. 2. readBytes(): It is used for reading the contents of the file to ByteArray. 3. readText(): It is used for reading contents of the file to a single String. 4. forEachLine(): It is used for reading a file line by line in Kotlin. 5. readLines(): It is used for reading lines in the file to List. </ending> | </end> |
| <hitle> | null exceptions in Kotlin | <chare> | 2 | <pext> | <starting>null exceptions in Kotlin</heading> In Kotlin, Elvis Operator is used to handling null expectations. </ending> | </end> |
| <hitle> | features kotlin only | <chare> | 2 | <pext> | <starting>features kotlin only</heading> 1. Null Safety 2. Operator Overloading 3. Coroutines 4. Range expressions 5. Smart casts 6. Companion Objects </ending> | </end> |
| <hitle> | extension functions | <chare> | 2 | <pext> | <starting>extension functions</heading> Extension functions are like extensive properties attached to any class in Kotlin. Extension functions are used to add methods or functionalities  to an existing class even without inheriting the class. For example: Suppose, we have views  where we need to play with the visibility of the views. So, we can create an extension function for views as follows: fun View.show() {   this.visibility = View.VISIBLE  }  fun View.hide() {   this.visibility = View.GONE  }  and to use it, we use, like,  toolbar.hide() </ending> | </end> |
| <hitle> | string types in kotlin | <chare> | 2 | <pext> | <starting>string types in kotlin</heading> A collection of characteristics together is known as a string. There are two types of strings available in the kotlin: 1. Raw string  2. Escaped string  </ending> | </end> |
| <hitle> | structural expressions in kotlin | <chare> | 2 | <pext> | <starting>structural expressions in kotlin</heading> 1. Break: break expression helps to break the closest enclosing loop 2. Return: This expression helps to return from the closest functions  or default functions. 3. Continue: This expression helps to proceed for the next loop.  </ending> | </end> |
| <hitle> | @JvmStatic, @JvmOverloads, and @JvmField | <chare> | 2 | <pext> | 1. @JvmStatic: This annotation is used to tell the compiler  that the method is a static method and can be used in Java code. 2. @JvmOverloads  Kotlin, with the help of default parameters and named arguments  helps us to reduce the number of overloads  that we generally need in Java world.  This annotation instructs the Kotlin compiler to generate overloads  for the function or constructor that reduced by default parameter values  so that they can be used in Java.  3. @JvmField  If we want a particular field to be used as normal field  and not as getter or setter then we have to tell the compiler not to generate  any getter and setter for the same and this can be done  by using the @JvmField annotation  so that we can access the fields of the class in the same way as in Kotlin. | </end> |
| <hitle> | data class vs regular class | <chare> | 2 | <pext> | <starting>data class vs regular class</heading> A data class is a class intended to hold data. A regular class is a class that can perform arbitrary operations. Data classes are typically simpler and more efficient than regular classes. </ending> | </end> |
| <hitle> | lambda expression | <chare> | 2 | <pext> | <starting>lambda expression</heading> A lambda expression is an anonymous function  that can concisely represent a function with a single parameter. Lambda expressions are often used in conjunction with higher-order functions,  such as map and filter. </ending> | </end> |
| <hitle> | function vs method | <chare> | 2 | <pext> | <starting>function vs method</heading> A function is a named code block invoked from other locations  within the source code. A method is a function associated with an object and can be invoked  from other code with the dot notation. </ending> | </end> |
| <hitle> | class vs object kotlin | <chare> | 2 | <pext> | <starting>class vs object</heading> A class is a template for creating objects. An object is an instance of a class. Classes can contain properties and methods, while objects contain only data. </ending> | </end> |
| <hitle> | constructor vs initializer in Kotlin | <chare> | 2 | <pext> | <starting>constructor vs initializer in Kotlin</heading> A constructor is a special method invoked when an object is created. An initializer is a special method you can use to initialize an object  before its first use. Both constructors and initializers are typically declared with the unit keyword. </ending> | </end> |
| <hitle> | default argument in Kotlin | <chare> | 2 | <pext> | <starting>default argument in Kotlin</heading> To declare a default argument in Kotlin, use the default keyword  when defining a function parameter. This will specify a default value for that argument, which will be used  if no actual value is supplied when calling the function. </ending> | </end> |
| <hitle> | define an object in Kotlin | <chare> | 2 | <pext> | <starting>define an object in Kotlin</heading> To define an object in Kotlin, simply declare a class and instantiate it  with the object keyword. This will create a new class instance, which can perform various actions. val newObject= object {  val one = "Hello"  val two = "World"  override fun toString() = "$one $two" } </ending> | </end> |
| <hitle> | higher-order function in Kotlin | <chare> | 2 | <pext> | <starting>higher-order function in Kotlin</heading> A higher-order function takes one or more functions as arguments  or returns a function as its result. Higher-order functions are often used in conjunction with lambda expressions  to create concise and powerful code. </ending> | </end> |
| <hitle> | suspend function vs regular function | <chare> | 2 | <pext> | <starting>suspend function vs regular function</heading> A suspend function can be suspended,  meaning that you can pause its execution and resume it at a later time. A regular function cannot be suspended and will always execute to completion. </ending> | </end> |
| <hitle> | create lambda expression | <chare> | 2 | <pext> | <starting>create lambda expression</heading> First, you must define the parameters the expression accepts  using the parentheses operator. You can then provide an executable block of code within curly braces  and use the arrow operator to indicate that this code is the body  of the lambda expression. val items = listOf(1, 2, 3, 4, 5) items.fold(0, {  acc: Int, i: Int ->  print("acc = $acc, i = $i, ")  val result = acc + i  println("result = $result")  result }) To pass a lambda expression as an argument to a function, include it within parentheses after the function name and any necessary arguments. This will cause the code within that lambda expression to be executed whenever the function is called. </ending> | </end> |
| <hitle> | lambda expression create | <chare> | 2 | <pext> | <starting>lambda expression create</heading> First, you must define the parameters the expression accepts using the parentheses operator. You can then provide an executable block of code within curly braces and use the arrow operator to indicate that this code is the body of the lambda expression. val items = listOf(1, 2, 3, 4, 5) items.fold(0, {  acc: Int, i: Int ->  print("acc = $acc, i = $i, ")  val result = acc + i  println("result = $result")  result }) To pass a lambda expression as an argument to a function, include it within parentheses after the function name and any necessary arguments. This will cause the code within that lambda expression to be executed whenever the function is called. </ending> | </end> |
| <hitle> | lambda expression vs anonymous function | <chare> | 2 | <pext> | <starting>lambda expression vs anonymous function</heading> A lambda expression is a function that can be passed as an argument  to another function. An anonymous function is a function that does not have a name  and cannot be passed as an argument to another function. Thus, they’re actually opposites. </ending> | </end> |
| <hitle> | inline functions in Kotlin | <chare> | 2 | <pext> | <starting>inline functions in Kotlin</heading> Inline functions are expanded inline at the call site,  meaning that the function code is copied and pasted into the body of the code  where it is called. This can improve performance by eliminating the need for a function call  while also increasing the readability of the code. </ending> | </end> |
| <hitle> | extension function vs regular function | <chare> | 2 | <pext> | <starting>extension function vs regular function</heading> An extension function is a function defined for a specific type  and can be called on variables of that type. A regular function is not defined for a specific type  and can be called on any type of variable. </ending> | </end> |
| <hitle> | type inference in Kotlin | <chare> | 2 | <pext> | <starting>type inference in Kotlin</heading> Type inference is the process  of automatically determining the type of a variable or expression  based on its value. In Kotlin, type inference determines the type of variables  when they are first declared and the return type of functions. </ending> | </end> |
| <hitle> | reified types in Kotlin | <chare> | 2 | <pext> | <starting> reified types in Kotlin</heading> Reified types can be accessed at runtime rather than just at compile time. In Kotlin, reified types provide metadata about a type at run time,  such as its name or the names of its members. They are also commonly used in reflection and generic programming.  It is an advanced feature  not commonly used by beginners to the Kotlin language. </ending> | </end> |
| <hitle> | create complier plugin kotlin | <chare> | 2 | <pext> | <starting>create complier plugin kotlin</heading> To create a compiler plugin in Kotlin, you first need to create an abstract class  that extends the CompilerPlugin class. This class defines several functions that should be implemented by your plugin,  including load and init. After this, you can use the CompilerInstance class  to access the Kotlin compiler and use its API. </ending> | </end> |
| <hitle> | Kotlin limitations cons | <chare> | 2 | <pext> | <starting>major limitations of Kotlin</heading> The major limitations of Kotlin are its lack of support for operator overloading  and variable arguments. Additionally, Kotlin does not have any built-in string interpolation  or formatting features. </ending> | </end> |
| <hitle> | asynchronous code vs concurrent code | <chare> | 2 | <pext> | <starting>asynchronous code vs concurrent code</heading> Asynchronous code is code that can run in the background  without blocking the main thread. Concurrent code is code that can run in parallel with other code. </ending> | </end> |
| <hitle> | concurrently instead of sequentially | <chare> | 2 | <pext> | <starting>concurrently instead of sequentially</heading> There are a few reasons you might want to run two tasks concurrently  instead of sequentially.  One reason is if the tasks are independent of each other  and can be run in parallel.  Another reason is if one task is dependent on the other task  and you want to avoid blocking the main thread.  Finally, if you have a limited number of resources available,  you might want to run tasks concurrently in order to make  better use of those resources. </ending> | </end> |
| <hitle> | kotlin pros/cons | <chare> | 2 | <pext> | <starting>kotlin pros/cons</heading> pros: 1. Easy Language − Kotlin supports object-oriented and functional constructs  and very easy to learn.  The syntax is pretty much similar to Java, hence for any Java programmer  it is very easy to remember any Kotlin Syntax. 2. Very Concise − Kotlin is based on Java Virtual Machine  and it is a functional language.  Thus, it reduce lots of boiler plate code used in other programming languages. 3. Runtime and Performance − Kotlin gives a better performance  and small runtime for any application. 4. Interoperability − Kotlin is mature enough  to build an interoperable application in a less complex manner. 5. Brand New − Kotlin is a brand new language  that gives developers a fresh start.  It is not a replacement of Java, though it is developed over JVM.  Kotlin has been accepted as the first official language  of Android Application Development.  Kotlin can also be defined as - Kotlin = Java + Extra updated new features.  cons: 1. Namespace declaration − Kotlin allows developers to declare the functions  at the top level.  However, whenever the same function is declared  in many places of your application, then it is hard to understand  which function is being called. 2. No Static Declaration − Kotlin does not have usual static handling modifier  like Java, which can cause some problem to the conventional Java developer. </ending> | </end> |
| <hitle> | Kotlin Mutable Variables var | <chare> | 2 | <pext> | <starting>Kotlin Mutable Variables var</heading> Mutable means that the variable can be reassigned  to a different value after initial assignment.  To declare a mutable variable, we use the var keyword. </ending> | </end> |
| <hitle> | Kotlin Read-only Variables val | <chare> | 2 | <pext> | <starting>Kotlin Read-only Variables val</heading> A read-only variable can be declared using val (instead of var)  and once a value is assigned, it can not be re-assigned. </ending> | </end> |
| <hitle> | Read-only vs Mutable | <chare> | 2 | <pext> | <starting>Read-only vs Mutable</heading> 1. The Mutable variables will be used to define variables,  which will keep charging their values based on different conditions  during program execution. 2. You will use Read-only variable to define different constant values  i.e. the variables which will retain their value throughout of the program. </ending> | </end> |
| <hitle> | Kotlin Variable Naming Rules | <chare> | 2 | <pext> | <starting>Kotlin Variable Naming Rules</heading> 1. Kotlin variable names can contain letters, digits, underscores,  and dollar signs. 2. Kotlin variable names should start with a letter, $ or underscores 3. Kotlin variables are case sensitive which means Zara and ZARA  are two different variables. 4. Kotlin variable can not have any white space or other control characters. 5. Kotlin variable can not have names like var, val, String, Int  because they are reserved keywords in Kotlin. </ending> | </end> |
| <hitle> | Kotlin String Templates | <chare> | 2 | <pext> | <starting>Kotlin String Templates</heading> Kotlin string templates are pieces of code that are evaluated  and whose results are interpolated into the string.  A template expression starts with a dollar sign ($) and may consist of  either a name or an expression. </ending> | </end> |
| <hitle> | Creating Arrays in Kotlin arrayOf(), arrayOf<T>() | <chare> | 2 | <pext> | <starting>Creating Arrays in Kotlin arrayOf(), arrayOf<T>()</heading> To create an array in Kotlin, we use the arrayOf() function,  and place the values in a comma-separated list inside it. </ending> | </end> |
| <hitle> | Kotlin Array Length | <chare> | 2 | <pext> | <starting>Kotlin Array Length</heading> Kotlin provides array property called size which returns the size  i.e. length of the array. We can also use count() member function to get the size of the array. </ending> | </end> |
| <hitle> | Kotlin Distinct Values from Array distinct() | <chare> | 2 | <pext> | <starting>Distinct Values from Array distinct()</heading> Kotlin allows to store duplicate values in an array,  but same time you can get a set of distinct values  stored in the array using distinct() member function. </ending> | </end> |
| <hitle> | Kotlin Dropping Elements from Array | <chare> | 2 | <pext> | <starting>Kotlin Dropping Elements from Array</heading> We can use drop() or dropLast() member functions to drop elements  from the beginning or from the last respectively. </ending> | </end> |
| <hitle> | Kotlin Checking an Empty Array | <chare> | 2 | <pext> | <starting>Kotlin Checking an Empty Array</heading> We can use isEmpty() member function to check if an array is empty or not. This function returns true if the array is empty. </ending> | </end> |
| <hitle> | Kotlin range | <chare> | 2 | <pext> | <starting>Kotlin range</heading> Kotlin range is defined by its two endpoint values  which are both included in the range.  Kotlin ranges are created with rangeTo() function,  or simply using downTo or (. .) operators.  The main operation on ranges is contains,  which is usually used in the form of in and !in operators. Example: 1.rangeTo(4) 1..4 4 downTo 1 1..10 step 2 'a'..'d' (1..5).reversed() 1 until 5 </ending> | </end> |
| <hitle> | Kotlin range rangeTo() | <chare> | 2 | <pext> | <starting>Kotlin range rangeTo()</heading> To create a Kotlin range we call rangeTo() function on the range start value  and provide the end value as an argument. 1.rangeTo(4) </ending> | </end> |
| <hitle> | Kotlin range downTo | <chare> | 2 | <pext> | <starting>Kotlin range downTo</heading> If we want to define a backward range we can use the downTo operator: 4 downTo 1 </ending> | </end> |
| <hitle> | Kotlin range step() | <chare> | 2 | <pext> | <starting>Kotlin range step()</heading> We can use step() function to define the distance  between the values of the range.  1..10 step 2 </ending> | </end> |
| <hitle> | Kotlin range of Characters | <chare> | 2 | <pext> | <starting>Kotlin range of Characters</heading> Ranges can be created for characters  like we have created them for integer values. 'a'..'d' </ending> | </end> |
| <hitle> | Kotlin range reversed() | <chare> | 2 | <pext> | <starting>Kotlin range reversed()</heading> The function reversed() can be used to reverse the values of a range. (1..5).reversed() </ending> | </end> |
| <hitle> | Kotlin range until() | <chare> | 2 | <pext> | <starting>Kotlin range until()</heading> The function until() can be used to create a range  but it will skip the last element given. 1 until 5 </ending> | </end> |
| <hitle> | Kotlin range first, last, step | <chare> | 2 | <pext> | <starting>Kotlin range first, last, step</heading> We can use first, last and step properties of a range to find the first,  the last value or the step of a range.  println((5..10).first)  println((5..10 step 2).step)  println((5..10).reversed().last) </ending> | </end> |
| <hitle> | Kotlin Filtering Ranges filter() | <chare> | 2 | <pext> | <starting>Kotlin Filtering Ranges filter()</heading> The filter() function will return a list of elements matching a given predicate:  val a = 1..10  val f = a.filter { T -> T % 2 == 0 } </ending> | </end> |
| <hitle> | Kotlin Distinct Values in a Range distinct() | <chare> | 2 | <pext> | <starting>Kotlin Distinct Values in a Range distinct()</heading> The distinct() function will return a list of distinct values  from a range having repeated values:  val a = listOf(1, 1, 2, 4, 4, 6, 10)  println(a.distinct()) </ending> | </end> |
| <hitle> | Kotlin Range Utility Functions min, max, sum, average, count | <chare> | 2 | <pext> | <starting>Kotlin Range Utility Functions min, max, sum, average, count</heading> There are many other useful functions we can apply to our range, like min, max, sum, average, count:  val a = 1..10  println(a.min())  println(a.max())  println(a.sum())  println(a.average())  println(a.count()) </ending> | </end> |
| <hitle> | Kotlin Recursive Function | <chare> | 2 | <pext> | <starting>Kotlin Recursive Function</heading> Kotlin Recursive Function Recursion functions are useful in many scenerios like calculating factorial  of a number or generating fibonacci series. Kotlin supports recursion which means a Kotlin function can call itself. fun functionName(){   ...  functionName()  ... }  </ending> | </end> |
| <hitle> | Kotlin Tail Recursion | <chare> | 2 | <pext> | <starting>Kotlin Tail Recursion</heading> A recursive function is eligible for tail recursion  if the function call to itself is the last operation it performs. </ending> | </end> |
| <hitle> | Kotlin Higher-Order Functions | <chare> | 2 | <pext> | <starting>Kotlin Higher-Order Functions</heading> A higher-order function is a function that takes another function  as parameter and/or returns a function. fun main(args: Array<String>) {    val result = calculate(4, 5, ::sum)   println( result )   } fun sum(a: Int, b: Int) = a + b   fun calculate(a: Int, b: Int, operation:(Int, Int) -> Int): Int {  return operation(a, b)  } Here we are calling the higher-order function passing in two integer values and the function argument ::sum. Here :: is the notation that references a function by name in Kotlin. </ending> | </end> |
| <hitle> | Kotlin Lambda Function | <chare> | 2 | <pext> | <starting>Kotlin Lambda Function</heading> Kotlin lambda is a function which has no name  and defined with a curly braces {} which takes zero or more parameters  and body of function. The body of function is written after variable (if any) followed by -> operator. Syntax: {variable with type -> body of the function}  val upperCase = { str: String -> str.toUpperCase() }  </ending> | </end> |
| <hitle> | Kotlin Inline Function | <chare> | 2 | <pext> | <starting>Kotlin Inline Function</heading> An inline function is declared with inline keyword.  The use of inline function enhances the performance of higher order function. The inline function tells the compiler to copy parameters and functions  to the call site. fun main(args: Array<String>) {   myFunction({println("Inline function parameter")}) } inline fun myFunction(function:()-> Unit){  println("I am inline function - A")  function()  println("I am inline function - B") }  I am inline function - A Inline function parameter I am inline function - B </ending> | </end> |
| <hitle> | Kotlin List | <chare> | 2 | <pext> | <starting>Kotlin List</heading> List is an ordered collection with access to elements by indices.  Elements can occur more than once in a list. </ending> | </end> |
| <hitle> | Kotlin Set | <chare> | 2 | <pext> | <starting>Kotlin Set</heading> Set is a collection of unique elements which means a group of objects  without repetitions. </ending> | </end> |
| <hitle> | Kotlin Map | <chare> | 2 | <pext> | <starting>Kotlin Map</heading> Map (or dictionary) is a set of key-value pairs. Keys are unique,  and each of them maps to exactly one value. </ending> | </end> |
| <hitle> | Kotlin Collection Types | <chare> | 2 | <pext> | <starting>Kotlin Collection Types</heading> 1. Collection or Immutable Collection 2. Mutable Collection </ending> | </end> |
| <hitle> | Kotlin - Collections | <chare> | 2 | <pext> | <starting>Kotlin - Collections</heading> Collections are a common concept for most programming languages.  A collection usually contains a number of objects of the same type and Objects  in a collection are called elements or items. The Kotlin Standard Library provides a comprehensive set of tools for  managing collections.  The following collection types are relevant for Kotlin: 1. Kotlin List - List is an ordered collection with access to elements by indices. Elements can occur more than once in a list. 2. Kotlin Set - Set is a collection of unique elements which means a group of objects without repetitions. 3. Kotlin Map - Map (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. </ending> | </end> |
| <hitle> | Kotlin Immutable Collection | <chare> | 2 | <pext> | <starting>Kotlin Immutable Collection</heading> Immutable Collection or simply calling a Collection interface provides read-only methods which means once a collection is created, we can not change it because there is no method available to change the object created. </ending> | </end> |
| <hitle> | Kotlin Methods of Immutable Collection | <chare> | 2 | <pext> | <starting>Kotlin Methods of Immutable Collection</heading> Immutable Collection or simply calling a Collection interface provides  read-only methods which means once a collection is created,  we can not change it because there is no method available  to change the object created. List: listOf(), listOf<T>() Map: mapOf() Set: setOf() </ending> | </end> |
| <hitle> | Kotlin Methods of Mutable Collection | <chare> | 2 | <pext> | <starting>Kotlin Methods of Mutable Collection</heading> Mutable collections provides both read and write methods. List: ArrayList<T>(), arrayListOf(), mutableListOf() Map: HashMap, hashMapOf(), mutableMapOf() Set: hashSetOf(), mutableSetOf() </ending> | </end> |
| <hitle> | Kotlin Iterator | <chare> | 2 | <pext> | <starting>Kotlin Iterator</heading> val itr = theList.listIterator()   while (itr.hasNext()) {  println(itr.next())  } </ending> | </end> |
| <hitle> | Kotlin Lists listOf(), mutableListOf() | <chare> | 2 | <pext> | <starting>Kotlin Lists listOf(), mutableListOf()</heading> Kotlin list is an ordered collection of items.  A Kotlin list can be either mutable (mutableListOf) or read-only (listOf).  The elements of list can be accessed using indices.  Kotlin mutable or immutable lists can have duplicate elements. </ending> | </end> |
| <hitle> | Size of Kotlin List | <chare> | 2 | <pext> | <starting>Size of Kotlin List</heading> We can use size property to get the total number of elements in a list: </ending> | </end> |
| <hitle> | Kotlin "in" Operator | <chare> | 2 | <pext> | <starting>Kotlin "in" Operator</heading> The in operator can be used to check the existence of an element in a list.  if("two" in theList){  println(true)  }else{  println(false)  } </ending> | </end> |
| <hitle> | Kotlin contain() | <chare> | 2 | <pext> | <starting>Kotlin contain()</heading> The contain() method can also be used to check the existence  of an element in a list.  if(theList.contains("two")){  println(true)  }else{  println(false)  } </ending> | </end> |
| <hitle> | Kotlin indexOf() | <chare> | 2 | <pext> | <starting>Kotlin indexOf()</heading> The indexOf() method returns the index of the first occurrence of the specified element in the list, or -1 if the specified element is not contained in the list. </ending> | </end> |
| <hitle> | Kotlin List Addition | <chare> | 2 | <pext> | <starting>Kotlin List Addition</heading> We can use + operator to add two or more lists into a single list.  This will add second list into first list, even duplicate elements  will also be added. resultList = firstList + secondList </ending> | </end> |
| <hitle> | Kotlin List Subtraction | <chare> | 2 | <pext> | <starting>Kotlin List Subtraction</heading> We can use - operator to subtract a list from another list.  This operation will remove the common elements from the first list  and will return the result. resultList = firstList - secondList </ending> | </end> |
| <hitle> | Kotlin Slicing a List | <chare> | 2 | <pext> | <starting>Kotlin Slicing a List</heading> We can obtain a sublist from a given list using slice() method which makes use of range of the elements indices. resultList = theList.slice( 2..4) </ending> | </end> |
| <hitle> | Kotlin Remove null a List **filterNotNull()** | <chare> | 2 | <pext> | <starting>Kotlin Remove null a List filterNotNull()</heading> We can use filterNotNull() method to remove null elements from a Kotlin list. </ending> | </end> |
| <hitle> | Kotlin Dropping Elements drop() | <chare> | 2 | <pext> | <starting>Kotlin Dropping Elements drop()</heading> We can use drop() method to drop first N elements from the list. </ending> | </end> |
| <hitle> | Kotlin Grouping List Elements groupBy() | <chare> | 2 | <pext> | <starting>Kotlin Grouping List Elements groupBy()</heading> We can use groupBy() method to group the elements matching  with the given predicate.  val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)  val resultList = theList.groupBy{ it % 3} {1=[10, 31, 40], 0=[12, 30, 9, -3, 0]} </ending> | </end> |
| <hitle> | Kotlin Mapping List map() | <chare> | 2 | <pext> | <starting>Kotlin Mapping List map()</heading> We can use map() method to map all elements using the provided function:.  val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)  val resultList = theList.map{ it / 3 } [3, 4, 10, 10, 13, 3, -1, 0] </ending> | </end> |
| <hitle> | Kotlin Chunking List chunked() | <chare> | 2 | <pext> | <starting>Kotlin Chunking List chunked()</heading> We can use chunked() method to create chunks of the given size from a list. Last chunk may not have the elements equal to the number of chunk size  based on the total number of elements in the list.  val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)  val resultList = theList.chunked(3) [[10, 12, 30], [31, 40, 9], [-3, 0]] </ending> | </end> |
| <hitle> | Kotlin Windowing List windowed() | <chare> | 2 | <pext> | <starting>Kotlin Windowing List windowed()</heading> We can use windowed() method to a list of element ranges  by moving a sliding window of a given size over a collection of elements.  val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)  val resultList = theList.windowed(3) [[10, 12, 30], [12, 30, 31], [30, 31, 40], [31, 40, 9], [40, 9, -3], [9, -3, 0]] </ending> | </end> |
| <hitle> | Kotlin mutable List mutableListOf() | <chare> | 2 | <pext> | <starting>Kotlin mutable List mutableListOf()</heading> We can create mutable list using mutableListOf(), later we can use add()  to add more elements in the same list, and we can use remove() method  to remove the elements from the list. </ending> | </end> |
| <hitle> | Kotlin Sets setOf(), mutableSetOf() | <chare> | 2 | <pext> | <starting>Kotlin Sets setOf(), mutableSetOf()</heading> Kotlin set is an unordered collection of items.  A Kotlin set can be either mutable (mutableSetOf) or read-only (setOf).  Kotlin mutable or immutable sets do not allow to have duplicate elements. </ending> | </end> |
| <hitle> | Loop through Kotlin Sets | <chare> | 2 | <pext> | <starting>Loop through Kotlin Sets</heading> Using Iterator  val theSet = setOf("one", "two", "three", "four")  val itr = theSet.asIterable().iterator()  while (itr.hasNext()) {  println(itr.next())  } Using for loop  val theSet = setOf("one", "two", "three", "four")  for (i in theSet.indices) {  println(theSet.elementAt(i))  } Using forEach  val theSet = setOf("one", "two", "three", "four")  theSet.forEach { println(it) } </ending> | </end> |
| <hitle> | Size of Kotlin Set | <chare> | 2 | <pext> | <starting>Size of Kotlin Set</heading> We can use size property to get the total number of elements in a set: </ending> | </end> |
| <hitle> | Kotlin Set elementAt() | <chare> | 2 | <pext> | <starting>Kotlin Set elementAt()</heading> The elementAt() method can be used to get the element  at the specified index in the set. </ending> | </end> |
| <hitle> | Kotlin Sorting Elements sorted(), sortedDescending() | <chare> | 2 | <pext> | <starting>Kotlin Sorting Elements sorted(), sortedDescending()</heading> We can use sorted() method to sort the elements in ascending order, or sortedDescending() method to sort the set elements in descending order. </ending> | </end> |
| <hitle> | Kotlin mutable Set | <chare> | 2 | <pext> | <starting>Kotlin mutable Set</heading> We can create mutable set using mutableSetOf(), later we can use add()  to add more elements in the same set, and we can use remove() method  to remove the elements from the set. </ending> | </end> |
| <hitle> | Kotlin Maps mapOf(), mutableMapOf() | <chare> | 2 | <pext> | <starting>Kotlin Maps mapOf(), mutableMapOf()</heading> Kotlin map is a collection of key/value pairs, where each key is unique,  and it can only be associated with one value.  The same value can be associated with multiple keys though.  We can declare the keys and values to be any type; there are no restrictions. A Kotlin map can be either mutable (mutableMapOf) or read-only (mapOf). Maps are also known as dictionaries or associative arrays  in other programming languages. </ending> | </end> |
| <hitle> | Creating Kotlin Maps | <chare> | 2 | <pext> | <starting>Creating Kotlin Maps</heading>  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  println(theMap)  val theMutableMap = mutableSetOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  println(theMutableMap) </ending> | </end> |
| <hitle> | Creating Map using HashMap | <chare> | 2 | <pext> | <starting>Creating Map using HashMap</heading>  val theMap = HashMap<String, Int>()  theMap["one"] = 1  theMap["two"] = 2  theMap["three"] = 3  theMap["four"] = 4 </ending> | </end> |
| <hitle> | Pair while Creating Map pair() | <chare> | 2 | <pext> | <starting>Pair while Creating Map</heading> val theMap = mapOf(Pair("one", 1), Pair("two", 2), Pair("three", 3)) </ending> | </end> |
| <hitle> | Kotlin map properties entries keys values | <chare> | 2 | <pext> | <starting>Kotlin map properties entries keys values</heading> Kotlin map has properties to get all entries, keys, and values of the map.  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  println("Entries: " + theMap.entries)  println("Keys:" + theMap.keys)  println("Values:" + theMap.values)  Entries: [one=1, two=2, three=3, four=4] Keys:[one, two, three, four] Values:[1, 2, 3, 4] </ending> | </end> |
| <hitle> | Loop through Kotlin Maps | <chare> | 2 | <pext> | <starting>Loop through Kotlin Maps</heading> Using Iterator  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  val itr = theMap.keys.iterator()  while (itr.hasNext()) {  val key = itr.next()  val value = theMap[key]  println("${key}=$value")  } Using For Loop  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  for ((k, v) in theMap) {  println("$k = $v")  } Using forEach  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  theMap.forEach {   k, v -> println("Key = $k, Value = $v")   } </ending> | </end> |
| <hitle> | Size of Kotlin Map | <chare> | 2 | <pext> | <starting>Size of Kotlin Map</heading> We can use size property or count() method to get the total number of elements in a map: </ending> | </end> |
| <hitle> | Kotlin containsKey(), containsValue() | <chare> | 2 | <pext> | <starting>Kotlin containsKey(), containsValue()</heading> The The containsKey() checks if the map contains a key.  The containsValue() checks if the map contains a value. </ending> | </end> |
| <hitle> | Sorting Map Elements toSortedMap(), sortedDescending(), sortedMapOf() | <chare> | 2 | <pext> | <starting>Sorting Map Elements toSortedMap(), sortedDescending(), sortedMapOf()</heading> We can use toSortedMap() method to sort the elements in ascending order,  or sortedDescending() method to sort the set elements in descending order. You can also create a sorted map with the given key/values using  sortedMapOf() method. Just use this method in place of mapOf(). </ending> | </end> |
| <hitle> | Filtering Map Elements filter(), filterKeys, filterValues() | <chare> | 2 | <pext> | <starting>Filtering Map Elements filter(), filterKeys, filterValues()</heading> We can use either filterKeys() or filterValues() method to filter out the entries. We can also use filter() method to filter out the elements  matching the both key/value  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)  var resultMap = theMap.filterValues{ it > 2}  resultMap = theMap.filterKeys{ it == "two"}  resultMap = theMap.filter{ it.key == "two" || it.value == 4} {three=3, four=4} {two=2} {two=2, four=4} </ending> | </end> |
| <hitle> | Mapping Map Elements map() | <chare> | 2 | <pext> | <starting>Mapping Map Elements map()</heading> We can use map() method to map all elements using the provided function:.  val theMap = mapOf("one" to 1, "two" to 2, "three" to 3)  val resultMap = theMap.map{ (k, v) -> "Key is $k, Value is $v" } [Key is one, Value is 1, Key is two, Value is 2, Key is three, Value is 3] </ending> | </end> |
| <hitle> | Kotlin Mutable Map mutableMapOf() | <chare> | 2 | <pext> | <starting>Kotlin Mutable Map mutableMapOf()</heading> We can create mutable set using mutableMapOf(), later we can use put  to add more elements in the same map, and we can use remove() method  to remove the elements from the set. </ending> | </end> |
| <hitle> | Kotlin Inner Class | <chare> | 2 | <pext> | <starting>Kotlin Inner Class</heading> When a nested class is marked with a keyword inner, then it will be called  as an Inner class.  An inner class can be accessed by the data member of the outer class. Unlike a nested class, inner class can access members of the outer class.  We cannot directly create an object of the inner class but it can be created  using the outer class object. class OuterClass{  // Members of Outer Class  class inner InnerClass{  // Members of Inner Class  } } val outerObj = OuterClass() val innerObj = outerObj.InnerClass() </ending> | </end> |
| <hitle> | Anonymous Inner Class | <chare> | 2 | <pext> | <starting>Anonymous Inner Class</heading> Anonymous inner class is a pretty good concept  that makes the life of a programmer very easy.  Whenever we are implementing an interface,  the concept of anonymous inner block comes into picture.  The concept of creating an object of interface using runtime object reference  is known as anonymous class. fun main(args: Array<String>) {  var programmer :Human = object:Human { // Anonymous class  override fun think() { // overriding the think method  print("I am an example of Anonymous Inner Class ")  }  }  programmer.think() } interface Human {  fun think() } result: I am an example of Anonymous Inner Class </ending> | </end> |
| <hitle> | Kotlin Type Aliases | <chare> | 2 | <pext> | <starting>Kotlin Type Aliases</heading> Kotlin Type Aliases means a way to give an alternative name to an existing type. Type alias provides a cleaner way to write a more readable code.  typealias User = Triple<String, String, Int> fun userInfo():User{  return Triple("Zara","Ali",21) } </ending> | </end> |
| <hitle> | Kotlin Class | <chare> | 2 | <pext> | <starting>Kotlin Class</heading> A class is a blueprint for the objects which defines a template to be used  to create the required objects. Classes are the main building blocks of any Object Oriented Programming language.  A Kotlin class is defined using the class keyword. </ending> | </end> |
| <hitle> | Kotlin Objects | <chare> | 2 | <pext> | <starting>Kotlin Objects</heading> he objects are created from the Kotlin class  and they share the common properties and behaviours  defined by a class in form of data members (properties) and  member functions (behaviours) respectively. </ending> | </end> |
| <hitle> | constructor types kotlin | <chare> | 2 | <pext> | <starting>constructor types kotlin</heading> 1. Primary Constructor - This type of constructor is initialised  in the class header and is provided after the class name. It is declared using the “constructor” keyword. Parameters are optional in this type of constructor. If no annotations or access modifiers are provided,  the constructor keyword can be omitted. The initialization code can be placed in a separate initializer block  prefixed with the init keyword because the primary constructor  cannot contain any code.  2. Secondary Constructor - Secondary constructors  allow for the initialization of variables  as well as the addition of logic to the class. They have the constructor keyword prefixed to them. The compiler determines which secondary constructor  will be called based on the inputs provided. We don't specify which constructor to use in the above program,  so the compiler chooses for us. In Kotlin, a class can contain one or more secondary constructors  and at most one primary constructor. The primary constructor initializes the class, while the secondary constructor  initialises the class and adds some additional logic. </ending> | </end> |
| <hitle> | primary constructor kotlin | <chare> | 2 | <pext> | <starting>primary constructor kotlin</heading> In Kotlin, the primary constructor is a part of the class header. Unlike Java, it doesn't need you to declare a constructor in the body of the class. Kotlin facilitates you to declare the constructor in the class header itself: class Person constructor(name: String, age: Int, salary: Int) {  }  Just like functions or methods, it takes a series of parameters with their type. These parameters initialize the variables present in the class. If you do not have any annotations or modifiers (public, private, protected), you can omit the constructor keyword like the following example. class Person (name: String, age: Int, salary: Int) {  }  By removing the constructor keyword, you can get code that is simplified and easy to understand. </ending> | </end> |
| <hitle> | Kotlin Initializer Block | <chare> | 2 | <pext> | <starting>Kotlin Initializer Block</heading> The primary constructor cannot contain any code.  Initialization code can be placed in initializer blocks prefixed  with the init keyword.  There could be more than one init blocks and during the initialization  of an instance, the initializer blocks are executed in the same order  as they appear in the class body, interleaved with the property initializers: class Person (val \_name: String, val \_age: Int) {  // Member Variables  var name: String  var age: Int   // Initializer Block  init {  this.name = \_name  this.age = \_age  println("Name = $name")  println("Age = $age")  } }  fun main(args: Array<String>) {  val person = Person("Zara", 20) } </ending> | </end> |
| <hitle> | Kotlin Secondary Constructor | <chare> | 2 | <pext> | <starting>Kotlin Secondary Constructor</heading> As mentioned earlier, Kotlin allows to create one or more  secondary constructors for your class.  This secondary constructor is created using the constructor keyword.  It is required whenever you want to create more than one constructor in Kotlin  or whenever you want to include more logic in the primary constructor  and you cannot do that because the primary constructor may be called  by some other class. class Person{  // Member Variables  var name: String  var age: Int   // Initializer Block  init {  println("Initializer Block")  }   // Secondary Constructor  constructor ( \_name: String, \_age: Int) {  this.name = \_name  this.age = \_age  println("Name = $name")  println("Age = $age")  } }  fun main(args: Array<String>) {  val zara = Person("Zara", 20) } </ending> | </end> |
| <hitle> | Secondary constructor not val/var parameter | <chare> | 2 | <pext> | <starting>Secondary constructor not val/var parameter</heading> Secondary constructor do not allow to use val or var  with secondary constructor parameters.  </ending> | </end> |
| <hitle> | Primary vs secondary constructor | <chare> | 2 | <pext> | A class in Kotlin can have at most one primary constructor,  and one or more secondary constructors.  The primary constructor initializes the class,  while the secondary constructor is used to initialize the class  and introduce some extra logic.  The secondary constructor needs to call explicitly primary constructor. | </end> |
| <hitle> | Kotlin Inheritance | <chare> | 2 | <pext> | <starting>Kotlin Inheritance</heading> Inheritance can be defined as the process  where one class acquires the members (methods and properties)  of another class.  With the use of inheritance the information is made manageable  in a hierarchical order. A class which inherits the members of other class is known as subclass  (derived class or child class) and the class whose members are being inherited  is known as superclass (base class or parent class). Inheritance is one of the key features of object-oriented programming  which allows user to create a new class from an existing class.  Inheritance we can inherit all the features from the base class  and can have additional features of its own as well. All classes in Kotlin have a common superclass called Any,  which is the default superclass for a class with no supertypes declared.  ---------type of Inheritance----------------  1. Single  One parent and one child  2. Multi-Level  Grand parent and parent and child  3. Hirarchical  One parent and two childs  4. Multiple  One parents can have multiple childs.  One child from multiple parents is not allowed in Kotlin  But we can achive it by Multiple interfaces  5. Hybrid  One child from two parents from one grad parent is not allowed.  </ending> | </end> |
| <hitle> | Kotlin Any | <chare> | 2 | <pext> | <starting>Kotlin Any</heading> Kotlin superclass Any has three methods: equals(), hashCode(), and toString(). Thus, these methods are defined for all Kotlin classes. Everything in Kotlin is by default final,  hence, we need to use the keyword open in front of the class declaration  to make it inheritable for other classes.  Kotlin uses operator ":" to inherit a class. </ending> | </end> |
| <hitle> | Kotlin class by default final | <chare> | 2 | <pext> | <starting>Kotlin class by default final</heading> Everything in Kotlin is by default final,  hence, we need to use the keyword open in front of the class declaration  to make it inheritable for other classes.  </ending> | </end> |
| <hitle> | override a val property with a var | <chare> | 2 | <pext> | <starting>override a val property with a var</heading> You can also override a val property with a var property, but not vice versa. This is allowed because a val property essentially declares a get method,  and overriding it as a var additionally declares a set method  in the derived class. </ending> | </end> |
| <hitle> | override keyword in a primary constructor | <chare> | 2 | <pext> | <starting>override keyword in a primary constructor</heading> We can also can use the override keyword as part of the property declaration  in a primary constructor. </ending> | </end> |
| <hitle> | Derived Class Initialization Order | <chare> | 2 | <pext> | <starting>Derived Class Initialization Order</heading> When we create an object of a derived class then constructor initialization  starts from the base class.  Which means first of all base class properties will be initialized,  after that any derived class instructor will be called and same applies  to any further derived classes. This means that when the base class constructor is executed, the properties  declared or overridden in the derived class have not yet been initialized. </ending> | </end> |
| <hitle> | Kotlin Overriding rules | <chare> | 2 | <pext> | <starting>Kotlin Overriding rules</heading> If a child class inherits multiple implementations of the same member  from its immediate superclasses, then it must override this member  and provide its own implementation. This is different from a child class which inherits members from a single parent,  in such case it is not mandatory for the child class to provide  the implementation of all the open members. </ending> | </end> |
| <hitle> | abstract class open | <chare> | 2 | <pext> | <starting>abstract class open</heading> Abstract classes are always open.  You do not need to explicitly use open keyword to inherit subclasses  from them. </ending> | </end> |
| <hitle> | kotlin multiple inheritances | <chare> | 2 | <pext> | <starting>kotlin multiple inheritances</heading> Kotlin doesn’t support multiple inheritances,  however, the same thing can be achieved by implementing more than  two interfaces at a time. interface A {  fun printMe() {  println(" method of interface A")  } } interface B {  fun printMeToo() {  println("I am another Method from interface B")  } } // implements two interfaces A and B class multipleInterfaceExample: A, B fun main(args: Array<String>) {  val obj = multipleInterfaceExample()  obj.printMe()  obj.printMeToo() } </ending> | </end> |
| <hitle> | kotlin default visibility -public | <chare> | 2 | <pext> | <starting>kotlin default visibility -public</heading> The default visibility is public </ending> | </end> |
| <hitle> | Extended Library Classes | <chare> | 2 | <pext> | <starting>Extended Library Classes</heading> Kotlin allows to extend the standard library classes  as well as user-defined classes.  For example, if you need a specialised function for standard Kotlin String class  which will return the number of vowels available in the string,  such method is not already available in String class  but you can use an extension function to accomplish this task. </ending> | </end> |
| <hitle> | Companion Object Extensions | <chare> | 2 | <pext> | <starting>Companion Object Extensions</heading> Kotlin provides another mechanism to implement static functionality of Java. This can be achieved using the companion object which is declared  inside a class and marked with the companion keyword.  Using this mechanism, we can create an object of a class  inside a factory method and later we can just call that method  using the reference of the class name. </ending> | </end> |
| <hitle> | Extension with Nullable Receiver | <chare> | 2 | <pext> | <starting>Extension with Nullable Receiver</heading> Kotlin allows to define Extension Functions with a nullable class type.  These extension function can be called on a nullable object variable. To define an extension for Nullable receiver, we just need to add a check  for null receiver inside the extension function,  and the appropriate value is returned. fun main(args: Array<String>) {  var str1 = "Good morning Kotlin"  var str2 : String? = null    var result = str1.countVowels()  println("Number of vowels in str1 : $result")    result = str2.countVowels()  println("Number of vowels in str2 : $result") } fun String?.countVowels(): Any{  if (this == null) return "null"    var vowels = 0  for (i in 0.. this.length - 1) {  val ch = this[i]  if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {  ++vowels  }  }  return vowels;  } result: Number of vowels in str1 : 6 Number of vowels in str2 : null </ending> | </end> |
| <hitle> | Extension Properties | <chare> | 2 | <pext> | <starting>Extension Properties</heading> Kotlin allows to define extension properties in very similar way like  we defined extension function.  Extension properties are also defined outside of the class.  Since extensions do not actually insert members into classes,  there is no efficient way for an extension property to have a backing field.  This is why initializers are not allowed for extension properties. We can add getter and setter along with the property which are nothing  but the extension functions. class Temperature(var celsius: Float) fun main(args: Array<String>) {  val t = Temperature(40f)  println(t.fahrenheit)  t.fahrenheit = 85f  println(t.celsius) } var Temperature.fahrenheit: Float  get() = (celsius \* 9 / 5) + 32  set(value) {  celsius = (value - 32) \* 5 / 9  } result: 104.0 29.444445 </ending> | </end> |
| <hitle> | kotlin data class conditioin | <chare> | 2 | <pext> | <starting>kotlin data class conditioin</heading> 1. The primary constructor needs to have at least one parameter. 2. All primary constructor parameters need to be marked as val or var. 3. Data classes cannot be abstract, open, sealed, or inner. 4. The class may extend other classes or implement interfaces.  </ending> | </end> |
| <hitle> | kotlin copy() | <chare> | 2 | <pext> | <starting>kotlin copy()</heading> The copy() function is created automatically  when we define a Kotlin Data Class.  This copy function can be used to copy an object  altering some of its properties but keeping the rest unchanged. </ending> | </end> |
| <hitle> | default functions automatically derived by data class | <chare> | 2 | <pext> | <starting>default functions automatically derived by data class</heading> 1. equals() - The equals() function returns true  if two objects have the identical contents. It operates similarly to "==,"  although for Float and Double values it works differently. 2. hashCode() - The hashCode() function returns the object's hashcode value. 3. copy() - The copy() function is used to duplicate an object,  changing only a few of its characteristics while leaving the rest unaltered. 4. toString() - This function returns a string  containing all of the data class's parameters. </ending> | </end> |
| <hitle> | Kotlin - Delegation | <chare> | 2 | <pext> | <starting>Kotlin - Delegation</heading> Kotlin supports “delegation” design pattern  by introducing a new keyword “by”.  Using this keyword or delegation methodology, Kotlin allows the derived class  to access all the implemented public methods of an interface  through a specific object. interface Base {  fun printMe() //abstract method } class BaseImpl(val x: Int) : Base {  override fun printMe() { println(x) } //implementation of the method } class Derived(b: Base) : Base by b // delegating the public method on the object b fun main(args: Array<String>) {  val b = BaseImpl(10)  Derived(b).printMe() // prints 10 :: accessing the printMe() method  } </ending> | </end> |
| <hitle> | Property Delegation | <chare> | 2 | <pext> | <starting>Property Delegation</heading> Delegation means passing the responsibility to another class or method.  When a property is already declared in some places,  then we should reuse the same code to initialize them.  We use some standard delegation methodology provided by Kotlin  and some standard library function while implementing delegation. 1. Using Lazy() Lazy is a lambda function which takes a property as an input  and in return gives an instance of Lazy<T>,  where <T> is basically the type of the properties it is using. 2. Delegetion.Observable() Observable() takes two arguments to initialize the object  and returns the same to the called function.  </ending> | </end> |
| <hitle> | Kotlin - Exception Handling | <chare> | 2 | <pext> | <starting>Kotlin - Exception Handling</heading> This technique restricts our application from generating the wrong output  at runtime. The exceptions in Kotlin is pretty similar to the exceptions in Java.  All the exceptions are descendants of the “Throwable” class. fun main(args: Array<String>) {  try {  val myVar:Int = 12;  val v:String = "Tutorialspoint.com";  v.toInt();  } catch(e:Exception) {  e.printStackTrace();  } finally {  println("Exception Handeling in Kotlin");  } } </ending> | </end> |
| <hitle> | kotlin final | <chare> | 2 | <pext> | <starting>kotlin final</heading> The final modifier mark classes and methods as not allowed to be overridden. In Kotlin this is the default. </ending> | </end> |
| <hitle> | Kotlin Infix notation | <chare> | 2 | <pext> | <starting>Kotlin Infix notation</heading> Functions marked with the infix keyword can also be called  using the infix notation (omitting the dot and the parentheses for the call).  To be Infix function: 1. They must be member functions or extension functions. 2. They must have a single parameter. 3. The parameter must not accept variable number of arguments  and must have no default value. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* infix fun Int.shl(x: Int): Int { ... } // calling the function using the infix notation 1 shl 2 // is the same as 1.shl(2) </ending> | </end> |
| <hitle> | Access/call kotlin extension function in java | <chare> | 2 | <pext> | All Kotlin functions declared in a file will be compiled by default  to static methods in a class within the same package  and with a name derived from the Kotlin source file  (First letter capitalized and ".kt" extension replaced with the "Kt" suffix).  Methods generated for extension functions  will have an additional first parameter with the extension function receiver type.  Java compiler will see Kotlin source file with the name xxx.kt  package com.test.extensions  public fun MyModel.bar(): Int { /\* actual code \*/ }  as if the following Java class was declared  package com.test.extensions  class ExampleKt {  public static int bar(MyModel receiver) { /\* actual code \*/ }  } | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |