|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | JDK in Java | <chare> | 1 | <pext> | The Java Development Kit (JDK) is  a cross-platformed software development environment  that offers a collection of tools and libraries necessary  for developing Java-based software applications and applets.  It is a core package used in Java,  along with the JVM (Java Virtual Machine)  and the JRE (Java Runtime Environment).  The Java Development Kit is an implementation  of one of the Java Platform:  1. Standard Edition (Java SE),  2. Java Enterprise Edition (Java EE),  3. Micro Edition (Java ME),  JDK contains:  1. Java Runtime Environment (JRE),  3. An interpreter/loader (Java),  4. A compiler (javac),  5. An archiver (jar) and many more.  Most Popular JDKs are Oracle JDK, OpenJDK, etc. | </end> |
| <hitle> | JRE in Java | <chare> | 1 | <pext> | The Java Runtime Environment (JRE) is software  that Java programs require to run correctly.  The JRE is the underlying technology that communicates  between the Java program and the operating system.  It acts as a translator and facilitator, providing all the resources  so that once you write Java software, it runs  on any operating system without further modifications.  The JRE runs on top of the operating system,  providing additional Java-specific resources.  The JDK and JRE interact to create a sustainable runtime environment  that runs Java program files on any machine.  A software program needs a runtime environment  that provides access to memory and other system resources  such as program files and dependencies.  In the past, most software used the operating system directly  as its runtime environment.  However, this meant that developers had to write different code  for each operating system that they wanted their applications  to run on.  The JRE technology was created as a solution to this problem. | </end> |
| <hitle> | JDK vs JRE vs JVM | <chare> | 1 | <pext> | JThe JDK is a software layer above the JRE that contains  a compiler, a debugger, and other tools commonly found  in any software development environment.  We write code in English-like syntax in the JDK.  The JDK compiles it and passes the byte code to the JRE.  In contrast, the JRE contains class libraries, supporting files,  and the JVM.  It uses these software components  to run the byte code on any device. | </end> |
| <hitle> | Java | <chare> | 1 | <pext> | Java is a programming language and computing platform first released  by Sun Microsystems in 1995. It has evolved from humble beginnings  to power a large share of today’s digital world,  by providing the reliable platform  upon which many services and applications are built. New, innovative products and digital services designed  for the future continue to rely on Java, as well. | </end> |
| <hitle> | Object-Oriented | <chare> | 2 | <pext> | <starting>Object-Oriented</heading> Object-Oriented: Java follows the object-oriented paradigm  which allows us to maintain our code  as the combination of different type of objects  that incorporates both data and behavior. It is a programming paradigm based on objects  having data and methods defined in the class to which it belongs.  Object-oriented paradigm aims to incorporate the advantages  of modularity and reusability.  Objects are the instances of classes  which interacts with one another to design applications and programs.  The features of the object-oriented paradigm are: 1. Follows the bottom-up approach in program design. 2. Focus on data with methods to operate upon the object's data 3. Includes the concept like Encapsulation and abstraction  which hides the complexities from the user and show only functionality. 4. Implements the real-time approach like inheritance, abstraction, etc. 5. The examples of the object-oriented paradigm  are C++, Simula, Smalltalk, Python, C#, etc. </ending> | </end> |
| <hitle> | static public void vs public static void | <chare> | 2 | <pext> | <starting>static public void vs public static void</heading> The program compiles and runs correctly  because the order of specifiers doesn't matter in Java. </ending> | </end> |
| <hitle> | default value of variable | <chare> | 2 | <pext> | <starting>default value of variable</heading> The local variables are not initialized to any default value,  neither primitives nor object references. </ending> | </end> |
| <hitle> | access specifiers | <chare> | 2 | <pext> | <starting>access specifiers</heading> 1. Public The classes, methods, or variables which are defined as public,  can be accessed by any class or method. 2. Protected Protected can be accessed by the class of the same package,  or by the sub-class of this class, or within the same class. 3. Default Default are accessible within the package only.  By default, all the classes, methods, and variables are of default scope. 4. Private The private class, methods, or variables defined as private  can be accessed within the class only. </ending> | </end> |
| <hitle> | static | <chare> | 2 | <pext> | <starting>static</heading> The methods or variables defined as static  are shared among all the objects of the class.  The static is the part of the class and not of the object.  The static variables are stored in the class area,  and we do not need to create the object to access such variables.  Therefore, static is used in the case,  where we need to define variables or methods  which are common to all the objects of the class. For example, In the class simulating the collection of the students in a college,  the name of the college is the common attribute to all the students.  Therefore, the college name will be defined as static. </ending> | </end> |
| <hitle> | object | <chare> | 2 | <pext> | <starting>object</heading> The Object is the real-time entity having some state and behavior.  In Java, Object is an instance of the class having the instance variables  as the state of the object and the methods as the behavior of the object.  The object of a class can be created by using the new keyword. </ending> | </end> |
| <hitle> | object-oriented vs object-based | <chare> | 2 | <pext> | <starting>object-oriented vs object-based</heading> 1. Object-oriented languages follow all the concepts of OOPs 1-1. whereas, the object-based language doesn't follow  all the concepts of OOPs like inheritance and polymorphism. 2. Object-oriented languages do not have the inbuilt objects 2-1. whereas Object-based languages have the inbuilt objects,  for example, JavaScript has window object. 3. Examples of object-oriented programming are Java, C#, Smalltalk, etc. 3-1. whereas the examples of object-based languages are JavaScript, VBScript, etc. </ending> | </end> |
| <hitle> | object reference | <chare> | 2 | <pext> | <starting>object reference</heading> All object references are initialized to null in Java. </ending> | </end> |
| <hitle> | constructor | <chare> | 2 | <pext> | <starting>constructor</heading> The constructor can be defined as the special type of method  that is used to initialize the state of an object.  It is invoked when the class is instantiated,  and the memory is allocated for the object.  Every time, an object is created using the new keyword,  the default constructor of the class is called.  The name of the constructor must be similar to the class name.  The constructor must not have an explicit return type. </ending> | </end> |
| <hitle> | constructor types | <chare> | 2 | <pext> | <starting>constructor types</heading> 1. Default Constructor:  default constructor is the one which does not accept any value.  The default constructor is mainly used to initialize the instance variable  with the default values.  It can also be used for performing some useful task on object creation.  A default constructor is invoked implicitly by the compiler  if there is no constructor defined in the class. 2. Parameterized Constructor:  The parameterized constructor is the one  which can initialize the instance variables with the given values.  In other words, we can say that the constructors  which can accept the arguments are called parameterized constructors. </ending> | </end> |
| <hitle> | constructor- default constructor | <chare> | 2 | <pext> | <starting>constructor- default constructor</heading> The purpose of the default constructor is to assign the default value  to the objects.  The java compiler creates a default constructor implicitly  if there is no constructor in the class. </ending> | </end> |
| <hitle> | constructor final | <chare> | 2 | <pext> | <starting>constructor final</heading> No, the constructor can't be final. </ending> | </end> |
| <hitle> | constructor overload | <chare> | 2 | <pext> | <starting>constructor overload</heading> Yes, the constructors can be overloaded  by changing the number of arguments accepted by the constructor  or by changing the data type of the parameters.  Consider the following example. </ending> | </end> |
| <hitle> | constructor object copy | <chare> | 2 | <pext> | <starting>constructor copy object</heading> There is no copy constructor in java.  However, we can copy the values from one object to another  like copy constructor in C++. There are many ways to copy the values of one object into another in java. They are: 1. By constructor 2. By assigning the values of one object into another 3. By clone() method of Object class </ending> | </end> |
| <hitle> | constructor vs method | <chare> | 2 | <pext> | <starting>constructor vs method</heading> 1. A constructor is used to initialize the state of an object. 1-1. A method is used to expose the behavior of an object. 2. A constructor must not have a return type. 2-1. A method must have a return type. 3. The constructor is invoked implicitly. 3-1. The method is invoked explicitly. 4. The Java compiler provides a default constructor  if you don't have any constructor in a class. 4-1. The method is not provided by the compiler in any case. 5. The constructor name must be same as the class name. 5-1. The method name may or may not be same as class name. </ending> | </end> |
| <hitle> | static variable | <chare> | 2 | <pext> | <starting>static variable</heading> The static variable is used to refer to the common property of all objects  (that is not unique for each object),  e.g., The company name of employees, college name of students, etc.  Static variable gets memory only once  in the class area at the time of class loading.  Using a static variable makes your program more memory efficient  (it saves memory).  Static variable belongs to the class rather than the object. </ending> | </end> |
| <hitle> | static method | <chare> | 2 | <pext> | <starting>static method</heading> 1. A static method belongs to the class rather than the object. 2. There is no need to create the object to call the static methods. 3. A static method can access and change the value of the static variable. </ending> | </end> |
| <hitle> | static method restriction | <chare> | 2 | <pext> | <starting>static method restriction</heading> Two main restrictions are applied to the static methods. 1. The static method can not use non-static data member  or call the non-static method directly. 2. this and super cannot be used in static context as they are non-static. </ending> | </end> |
| <hitle> | static methods override | <chare> | 2 | <pext> | <starting>static methods override</heading> No, we can't override static methods. </ending> | </end> |
| <hitle> | static block | <chare> | 2 | <pext> | <starting>static block</heading> Static block is used to initialize the static data member. It is executed before the main method, at the time of classloading. </ending> | </end> |
| <hitle> | static method vs instance method | <chare> | 2 | <pext> | <starting>static method vs instance method</heading> 1. A method that is declared as static is known as the static method. 1-1. A method that is not declared as static is known as the instance method. 2. We don't need to create the objects to call the static methods. 2-1. The object is required to call the instance methods. 3. Non-static members cannot be accessed in the static context directly (static method, static block, and static nested class) 3-1. Static and non-static variables both can be accessed in instance methods. 4. For example: public static int cube(int n){ return n\*n\*n;} 4-1. For example: public void msg(){...}. </ending> | </end> |
| <hitle> | constructor static | <chare> | 2 | <pext> | <starting>constructor static</heading> As we know that the static context (method, block, or variable)  belongs to the class, not the object.  Since Constructors are invoked only when the object is created,  there is no sense to make the constructors static.  However, if you try to do so, the compiler will show the compiler error. </ending> | </end> |
| <hitle> | abstract method static | <chare> | 2 | <pext> | <starting>abstract method static</heading> In Java, if we make the abstract methods static,   It will become the part of the class,  and we can directly call it which is unnecessary.  Calling an undefined method is completely useless therefore it is not allowed. </ending> | </end> |
| <hitle> | static variable in abstract class | <chare> | 2 | <pext> | <starting>static variable in abstract class</heading> Yes, we can declare static variables and methods in an abstract method.  As we know that there is no requirement to make the object  to access the static context, therefore, we can access the static context  declared inside the abstract class by using the name of the abstract class.  </ending> | </end> |
| <hitle> | this java | <chare> | 2 | <pext> | <starting>this java</heading> The this keyword is a reference variable that refers to the current object.  There are the various uses of this keyword in Java.  It can be used to refer to current class properties such as instance methods,  variable, constructors, etc.  It can also be passed as an argument into the methods or constructors.  It can also be returned from the method as the current class instance. </ending> | </end> |
| <hitle> | this uses | <chare> | 2 | <pext> | <starting>this uses</heading> 1. this can be used to refer to the current class instance variable. 2. this can be used to invoke current class method (implicitly) 3. this() can be used to invoke the current class constructor. 4. this can be passed as an argument in the method call. 5. this can be passed as an argument in the constructor call. 6. this can be used to return the current class instance from the method. </ending> | </end> |
| <hitle> | this assign reference | <chare> | 2 | <pext> | <starting>this assign reference</heading> No, this cannot be assigned to any value because it always points  to the current class object and this is the final reference in Java.  However, if we try to do so, the compiler error will be shown.  </ending> | </end> |
| <hitle> | this static members | <chare> | 2 | <pext> | <starting>this static members</heading> Yes, It is possible to use this keyword to refer static members  because this is just a reference variable  which refers to the current class object.  However, as we know that, it is unnecessary to access static variables  through objects, therefore, it is not the best practice to use this  to refer static members. </ending> | </end> |
| <hitle> | this constructor chaining | <chare> | 2 | <pext> | <starting>this constructor chaining</heading> Constructor chaining enables us to call one constructor  from another constructor of the class with respect to the current class object. We can use this keyword to perform constructor chaining  within the same class.  </ending> | </end> |
| <hitle> | this passing into method | <chare> | 2 | <pext> | <starting>this passing into a method</heading> 1. this is a final variable. Therefore, this cannot be assigned to any new value  whereas the current class object might not be final and can be changed. 2. this can be used in the synchronized block. </ending> | </end> |
| <hitle> | Inheritance | <chare> | 2 | <pext> | <starting>Inheritance</heading> Inheritance is a mechanism by which  one object acquires all the properties and behavior of another object  of another class.  It is used for Code Reusability and Method Overriding.  The idea behind inheritance in Java is that you can create new classes  that are built upon existing classes.  When you inherit from an existing class, you can reuse methods and fields  of the parent class.  Moreover, you can add new methods and fields in your current class also.  There are five types of inheritance in Java. 1. Single-level inheritance 2. Multi-level inheritance 3. Multiple Inheritance 4. Hierarchical Inheritance 5. Hybrid Inheritance Multiple inheritance is not supported in Java through class. </ending> | </end> |
| <hitle> | Inheritance pros | <chare> | 2 | <pext> | <starting>Inheritance pros</heading> 1. Inheritance provides code reusability.  The derived class does not need to redefine the method of base class  unless it needs to provide the specific implementation of the method. 2. Runtime polymorphism cannot be achieved without using inheritance. 3. We can simulate the inheritance of classes with the real-time objects  which makes OOPs more realistic. 4. Inheritance provides data hiding.  The base class can hide some data from the derived class by making it private. 5. Method overriding cannot be achieved without inheritance.  By method overriding, we can give  a specific implementation of some basic method contained by the base class. </ending> | </end> |
| <hitle> | multiple inheritance | <chare> | 2 | <pext> | <starting>multiple inheritance</heading> To reduce the complexity and simplify the language,  multiple inheritance is not supported in java.  Consider a scenario where A, B, and C are three classes.  The C class inherits A and B classes.  If A and B classes have the same method and you call it from child class object,  there will be ambiguity to call the method of A or B class. Since the compile-time errors are better than runtime errors,  Java renders compile-time error if you inherit 2 classes.  So whether you have the same method or different,  there will be a compile time error. </ending> | </end> |
| <hitle> | aggregation | <chare> | 2 | <pext> | <starting>aggregation</heading> Aggregation can be defined as the relationship between two classes where the aggregate class contains a reference to the class it owns. Aggregation is best described as a has-a relationship. For example, The aggregate class Employee having various fields such as age, name, and salary also contains an object of Address class having various fields such as Address-Line 1, City, State, and pin-code. In other words, we can say that Employee (class) has an object of Address class. </ending> | </end> |
| <hitle> | composition | <chare> | 2 | <pext> | <starting>composition</heading> Holding the reference of a class within some other class  is known as composition.  When an object contains the other object,  if the contained object cannot exist without the existence of container object,   then it is called composition.  In other words, we can say that composition  is the particular case of aggregation which represents a stronger relationship  between two objects.  Example: A class contains students. A student cannot exist without a class. There exists composition between class and students. </ending> | </end> |
| <hitle> | aggregation vs composition | <chare> | 2 | <pext> | <starting>aggregation vs composition</heading> Aggregation represents the weak relationship  whereas composition represents the strong relationship.  For example, the bike has an indicator (aggregation),  but the bike has an engine (composition). </ending> | </end> |
| <hitle> | super | <chare> | 2 | <pext> | <starting>super</heading> The super keyword in Java is a reference variable that is used to refer  to the immediate parent class object.  Whenever you create the instance of the subclass,  an instance of the parent class is created implicitly which is referred  by super reference variable.  The super() is called in the class constructor implicitly by the compiler  if there is no super or this. </ending> | </end> |
| <hitle> | constructor chaining by super | <chare> | 2 | <pext> | <starting>constructor chaining by super</heading> class Person  {   String name,address;   int age;   public Person(int age, String name, String address)   {   this.age = age;   this.name = name;   this.address = address;   }  }  class Employee extends Person  {   float salary;   public Employee(int age, String name, String address, float salary)   {   super(age,name,address);   this.salary = salary;   }  }  public class Test  {   public static void main (String args[])   {   Employee e = new Employee(22, "Mukesh", "Delhi", 90000);   System.out.println("Name: "+e.name+" Salary: "+e.salary+" Age: "+e.age+" Address: "+e.address);   }  }  </ending> | </end> |
| <hitle> | super uses | <chare> | 2 | <pext> | <starting>uses of super</heading> 1. super can be used to refer to the immediate parent class instance variable. 2. super can be used to invoke the immediate parent class method. 3. super() can be used to invoke immediate parent class constructor. </ending> | </end> |
| <hitle> | this vs super | <chare> | 2 | <pext> | <starting>this vs super</heading> 1. The super keyword always points to the parent class contexts  whereas this keyword always points to the current class context. 2. The super keyword is primarily used for initializing the base class variables  within the derived class constructor  whereas this keyword primarily used to differentiate  between local and instance variables when passed in the class constructor. 3. The super and this must be the first statement inside constructor  otherwise the compiler will throw an error. </ending> | </end> |
| <hitle> | this(),super() in constructor | <chare> | 2 | <pext> | <starting>this(),super() in constructor</heading> No, because this() and super() must be the first statement in the class constructor. </ending> | </end> |
| <hitle> | object clone | <chare> | 2 | <pext> | <starting>object clone</heading> The object cloning is used to create the exact copy of an object.  The clone() method of the Object class is used to clone an object.  The java.lang.Cloneable interface must be implemented by the class  whose object clone we want to create.  If we don't implement Cloneable interface,  clone() method generates CloneNotSupportedException. </ending> | </end> |
| <hitle> | overloading | <chare> | 2 | <pext> | <starting>overloading</heading> Method overloading is the polymorphism technique  which allows us to create multiple methods with the same name  but different signature.  We can achieve method overloading in two ways. 1. By Changing the number of arguments 2. By Changing the data type of arguments Method overloading increases the readability of the program.  Method overloading is performed to figure out the program quickly.  Why is method overloading not possible by changing the return type in java? In Java, method overloading is not possible  by changing the return type of the program due to avoid the ambiguity.  Can we overload the methods by making them static? No, We cannot overload the methods by just applying the static keyword to them (number of parameters and types are the same). </ending> | </end> |
| <hitle> | overloading return type | <chare> | 2 | <pext> | <starting>overloading type promotion</heading> In Java, method overloading is not possible  by changing the return type of the program due to avoid the ambiguity. </ending> | </end> |
| <hitle> | overloading static method | <chare> | 2 | <pext> | <starting>overloading static method</heading> We cannot overload the methods by just applying the static keyword to them (number of parameters and types are the same). </ending> | </end> |
| <hitle> | overloading type promotion | <chare> | 2 | <pext> | <starting>overloading type promotion</heading> By Type promotion is method overloading, we mean that one data type can be promoted to another implicitly if no exact matching is found. The byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on. Consider the following example. </ending> | </end> |
| <hitle> | overriding | <chare> | 2 | <pext> | <starting>overriding</heading> If a subclass provides a specific implementation of a method  that is already provided by its parent class, it is known as Method Overriding. It is used for runtime polymorphism and to implement the interface methods. Rules for Method overriding 1. The method must have the same name as in the parent class. 2. The method must have the same signature as in the parent class. Two classes must have an IS-A relationship between them.  Can we override the static method? No, you can't override the static method because they are the part of the class, not the object.  Why can we not override static method? It is because the static method is the part of the class, and it is bound with class whereas instance method is bound with the object, and static gets memory in class area, and instance gets memory in a heap.  Can we override the overloaded method? Yes.  Can we override the private methods? No, we cannot override the private methods because the scope of private methods is limited to the class and we cannot access them outside of the class. </ending> | </end> |
| <hitle> | overloading vs overriding | <chare> | 2 | <pext> | <starting>overloading vs overriding</heading> 1. Method overloading increases the readability of the program.  1-1. Method overriding provides the specific implementation of the method  that is already provided by its superclass. 2. Method overloading occurs within the class. 2-1. Method overriding occurs in two classes that have IS-A relationship between them. 3. In this case, the parameters must be different. 3-1. In this case, the parameters must be the same. </ending> | </end> |
| <hitle> | overriding vs overloading | <chare> | 2 | <pext> | <starting>overriding vs overloading</heading> 1. Method overloading increases the readability of the program.  1-1. Method overriding provides the specific implementation of the method that is already provided by its superclass. 2. Method overloading occurs within the class. 2-1. Method overriding occurs in two classes  that have IS-A relationship between them. 3. In this case, the parameters must be different. 3-1. In this case, the parameters must be the same. </ending> | </end> |
| <hitle> | overriding change scope | <chare> | 2 | <pext> | <starting>overriding change scope</heading> Yes, we can change the scope of the overridden method in the subclass. However, we must notice that we cannot decrease the accessibility  of the method.  Some of points must be taken care of  while changing the accessibility of the method. 1. The private can be changed to protected, public, or default. 2. The protected can be changed to public or default. 3. The default can be changed to public. 4. The public will always remain public. </ending> | </end> |
| <hitle> | overriding throws clause | <chare> | 2 | <pext> | <starting>overriding throws clause</heading> Yes, we can modify the throws clause of the superclass method  while overriding it in the subclass.  However, there are some rules which are to be followed while overriding  in case of exception handling. 1. If the superclass method does not declare an exception,  subclass overridden method cannot declare the checked exception,  but it can declare the unchecked exception. 2. If the superclass method declares an exception,  subclass overridden method can declare same,  subclass exception or no exception but cannot declare parent exception. </ending> | </end> |
| <hitle> | virtual | <chare> | 2 | <pext> | <starting>virtual</heading> Yes, all functions in Java are virtual by default. </ending> | </end> |
| <hitle> | covariant return type | <chare> | 2 | <pext> | <starting>covariant return type</heading> Now, since java5, it is possible to override any method  by changing the return type if the return type  of the subclass overriding method is subclass type.  It is known as covariant return type.  The covariant return type specifies that the return type may vary  in the same direction as the subclass. </ending> | </end> |
| <hitle> | final variable | <chare> | 2 | <pext> | <starting>final variable</heading> In Java, the final variable is used to restrict the user from updating it.  If we initialize the final variable, we can't change its value.  In other words, we can say that the final variable once assigned to a value,  can never be changed after that.  The final variable which is not assigned to any value can only be assigned  through the class constructor. </ending> | </end> |
| <hitle> | final method | <chare> | 2 | <pext> | <starting>final method</heading> If we change any method to a final method, we can't override it. </ending> | </end> |
| <hitle> | final class | <chare> | 2 | <pext> | <starting>final class</heading> If we make any class final, we can't inherit it into any of the subclasses. </ending> | </end> |
| <hitle> | final blank variable | <chare> | 2 | <pext> | <starting>final blank variable</heading> A final variable, not initialized at the time of declaration, is known  as the final blank variable.  We can't initialize the final blank variable directly.  Instead, we have to initialize it by using the class constructor.  It is useful in the case when the user has some data  which must not be changed by others, for example, PAN Number.  </ending> | </end> |
| <hitle> | final blank variable initialize | <chare> | 2 | <pext> | <starting>final blank variable initialize</heading> Yes, if it is not static, we can initialize it in the constructor. If it is static blank final variable, it can be initialized only in the static block. </ending> | </end> |
| <hitle> | constructor as final | <chare> | 2 | <pext> | <starting>constructor as final</heading> The constructor can never be declared as final because it is never inherited. Constructors are not ordinary methods;  therefore, there is no sense to declare constructors as final.  However, if you try to do so, The compiler will throw an error. </ending> | </end> |
| <hitle> | interface as final | <chare> | 2 | <pext> | <starting>interface as final</heading> No, we cannot declare an interface as final  because the interface must be implemented by some class  to provide its definition.  Therefore, there is no sense to make an interface final.  However, if you try to do so, the compiler will show an error. </ending> | </end> |
| <hitle> | final method vs abstract method | <chare> | 2 | <pext> | <starting>final method vs abstract method</heading> The main difference between the final method and abstract method  is that the abstract method cannot be final  as we need to override them in the subclass to give its definition. </ending> | </end> |
| <hitle> | compile-time vs runtime | <chare> | 2 | <pext> | <starting>compile-time vs runtime</heading> 1. In compile-time polymorphism,  call to a method is resolved at compile-time. 1-1. In runtime polymorphism,  call to an overridden method is resolved at runtime. 2. compile-time, It is also known as static binding,  early binding, or overloading. 2-1. runtime, It is also known  as dynamic binding, late binding, overriding, or dynamic method dispatch. 3. compile-time, Overloading is a way to achieve compile-time polymorphism  in which, we can define multiple methods or constructors  with different signatures. 3-1. runtime, Overriding is a way to achieve runtime polymorphism  in which, we can redefine some particular method or variable  in the derived class.  By using overriding, we can give some specific implementation  to the base class properties in the derived class. 4. compile-time, It provides fast execution because the type of an object  is determined at compile-time. 4-1. runtime, It provides slower execution as compare to compile-time  because the type of an object is determined at run-time. 5. Compile-time polymorphism provides less flexibility  because all the things are resolved at compile-time. 5-1. Run-time polymorphism provides more flexibility  because all the things are resolved at runtime. </ending> | </end> |
| <hitle> | Runtime Polymorphism | <chare> | 2 | <pext> | <starting>Runtime Polymorphism</heading> Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.  Can you achieve Runtime Polymorphism by data members? No, because method overriding is used to achieve runtime polymorphism and data members cannot be overridden. We can override the member functions but not the data members.  </ending> | </end> |
| <hitle> | static binding vs dynamic binding | <chare> | 2 | <pext> | <starting>static binding vs dynamic binding</heading> In case of the static binding,  the type of the object is determined at compile-time  whereas, in the dynamic binding, the type of the object  is determined at runtime. </ending> | </end> |
| <hitle> | instanceOf | <chare> | 2 | <pext> | <starting>instanceOf</heading> The instanceof in Java is also known as type comparison operator  because it compares the instance with type.  It returns either true or false.  If we apply the instanceof operator with any variable that has a null value,  it returns false. </ending> | </end> |
| <hitle> | abstraction | <chare> | 2 | <pext> | <starting>abstraction</heading> Abstraction is a process of hiding the implementation details  and showing only functionality to the user.  It displays just the essential things to the user  and hides the internal information, for example, sending SMS  where you type the text and send the message.  You don't know the internal processing about the message delivery. Abstraction enables you to focus on what the object does instead of  how it does it.  Abstraction lets you focus on what the object does instead of how it does it. In Java, there are two ways to achieve the abstraction. 1. Abstract Class 2. Interface </ending> | </end> |
| <hitle> | abstraction vs encapsulation | <chare> | 2 | <pext> | <starting>abstraction vs encapsulation</heading> Abstraction hides the implementation details  whereas encapsulation wraps code and data into a single unit. </ending> | </end> |
| <hitle> | abstract class | <chare> | 2 | <pext> | <starting>abstract class</heading> A class that is declared as abstract is known as an abstract class.  It needs to be extended and its method implemented.  It cannot be instantiated.  It can have abstract methods, non-abstract methods, constructors,  and static methods.  It can also have the final methods which will force the subclass  not to change the body of the method.  Can there be an abstract method without an abstract class? No, if there is an abstract method in a class, that class must be abstract.   Can you use abstract and final both with a method? No, because we need to override the abstract method  to provide its implementation, whereas we can't override the final method.  Can you use abstract and final both with a method? No, because we need to override the abstract method  to provide its implementation, whereas we can't override the final method.  Is it possible to instantiate the abstract class? No, the abstract class can never be instantiated  even if it contains a constructor and all of its methods are implemented. </ending> | </end> |
| <hitle> | interface | <chare> | 2 | <pext> | <starting>interface</heading> The interface is a blueprint for a class  that has static constants and abstract methods.  It can be used to achieve full abstraction and multiple inheritance.  It is a mechanism to achieve abstraction.  There can be only abstract methods in the Java interface, not method body.  It is used to achieve abstraction and multiple inheritance in Java.  In other words, you can say that interfaces can have  abstract methods and variables.  Java Interface also represents the IS-A relationship.  It cannot be instantiated just like the abstract class.  However, we need to implement it to define its methods.  Since Java 8, we can have the default, static,  and private methods in an interface.  Can you declare an interface method static? No, because methods of an interface are abstract by default,  and we can not use static and abstract together.  Can the Interface be final? No, because an interface needs to be implemented by the other class  and if it is final, it can't be implemented by any class.  Can we define private and protected modifiers for the members in interfaces? No, they are implicitly public. </ending> | </end> |
| <hitle> | Marker interface | <chare> | 2 | <pext> | <starting>Marker interface</heading> A Marker interface can be defined as the interface  which has no data member and member functions.  For example, Serializable, Cloneable are marker interfaces.  </ending> | </end> |
| <hitle> | abstract class vs interface | <chare> | 2 | <pext> | <starting>abstract class vs interface</heading> 1. An abstract class can have a method body (non-abstract methods). 1-1. The interface has only abstract methods. 2. An abstract class can have instance variables. 2-1. An interface cannot have instance variables. 3. An abstract class can have the constructor. 3-1. The interface cannot have the constructor. 4. An abstract class can have static methods. 4-1. The interface cannot have static methods. 5. You can extend one abstract class. 5-1. You can implement multiple interfaces. 6. The abstract class can provide the implementation of the interface. 6-1. The Interface can't provide the implementation of the abstract class. 7. The abstract keyword is used to declare an abstract class. 7-1. The interface keyword is used to declare an interface. 8. An abstract class can extend another Java class  and implement multiple Java interfaces. 8-1. An interface can extend another Java interface only. 9. An abstract class can be extended using keyword extends 9-1. An interface class can be implemented using keyword implements 10. A Java abstract class can have class members like private, protected, etc. 10-1. Members of a Java interface are public by default. </ending> | </end> |
| <hitle> | interface vs abstract class | <chare> | 2 | <pext> | <starting>interface vs abstract class</heading> 1. An abstract class can have a method body (non-abstract methods). 1-1. The interface has only abstract methods. 2. An abstract class can have instance variables. 2-1. An interface cannot have instance variables. 3. An abstract class can have the constructor. 3-1. The interface cannot have the constructor. 4. An abstract class can have static methods. 4-1. The interface cannot have static methods. 5. You can extend one abstract class. 5-1. You can implement multiple interfaces. 6. The abstract class can provide the implementation of the interface. 6-1. The Interface can't provide the implementation of the abstract class. 7. The abstract keyword is used to declare an abstract class. 7-1. The interface keyword is used to declare an interface. 8. An abstract class can extend another Java class and implement multiple Java interfaces. 8-1. An interface can extend another Java interface only. 9. An abstract class can be extended using keyword extends 9-1. An interface class can be implemented using keyword implements 10. A Java abstract class can have class members like private, protected, etc. 10-1. Members of a Java interface are public by default. </ending> | </end> |
| <hitle> | interface reference casting | <chare> | 2 | <pext> | <starting>interface reference casting</heading> An object reference can be cast to an interface reference  when the object implements the referenced interface. </ending> | </end> |
| <hitle> | object reference casting interface | <chare> | 2 | <pext> | <starting>object reference casting interface</heading> An object reference can be cast to an interface reference  when the object implements the referenced interface. </ending> | </end> |
| <hitle> | read-only class | <chare> | 2 | <pext> | <starting>read-only class</heading> A class can be made read-only by making all of the fields private.  The read-only class will have only getter methods  which return the private property of the class to the main method.  We cannot modify this property because there is no setter method  available in the class. </ending> | </end> |
| <hitle> | write-only class | <chare> | 2 | <pext> | <starting>write-only class</heading> A class can be made write-only by making all of the fields private.  The write-only class will have only setter methods which set the value passed from the main method to the private fields.  We cannot read the properties of the class  because there is no getter method in this class. </ending> | </end> |
| <hitle> | Encapsulation pros | <chare> | 2 | <pext> | <starting>Encapsulation pros</heading> 1. By providing only the setter or getter method,  you can make the class read-only or write-only.  In other words, you can skip the getter or setter methods. 2. It provides you the control over the data.  Suppose you want to set the value of id which should be greater than 100 only,  you can write the logic inside the setter method.  You can write the logic not to store the negative numbers  in the setter methods. 3. It is a way to achieve data hiding in Java because other  class will not be able to access the data through the private data members. 4. The encapsulate class is easy to test. So, it is better for unit testing. 5. The standard IDE's are providing the facility  to generate the getters and setters.  So, it is easy and fast to create an encapsulated class in Java. </ending> | </end> |
| <hitle> | exception types | <chare> | 2 | <pext> | <starting>exception types</heading> There are mainly two types of exceptions: checked and unchecked.  An error is considered as the unchecked exception.  According to Oracle, there are three types of exceptions: 1. Checked Exception: Checked exceptions are the one which are checked at compile-time.  For example, SQLException, ClassNotFoundException, etc. 2. Unchecked Exception:  Unchecked exceptions are the one which are handled at runtime  because they can not be checked at compile-time.  For example, ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. 3. Error: Error cause the program to exit since they are not recoverable.  For Example, OutOfMemoryError, AssertionError, etc. </ending> | </end> |
| <hitle> | Exception | <chare> | 2 | <pext> | <starting>Exception</heading> Exception Handling is a mechanism that is used to handle runtime errors.  It is used primarily to handle checked exceptions.  Exception handling maintains the normal flow of the program.  There are mainly two types of exceptions: checked and unchecked.  Here, the error is considered as the unchecked exception.  Can subclass overriding method declare an exception if parent class method doesn't throw an exception? Yes but only unchecked exception not checked. </ending> | </end> |
| <hitle> | Exception Handling | <chare> | 2 | <pext> | <starting>Exception Handling</heading> Exception Handling is a mechanism that is used to handle runtime errors. It is used primarily to handle checked exceptions. Exception handling maintains the normal flow of the program. There are mainly two types of exceptions: checked and unchecked. Here, the error is considered as the unchecked exception.  Can subclass overriding method declare an exception if parent class method doesn't throw an exception? Yes but only unchecked exception not checked. </ending> | </end> |
| <hitle> | exception classe hierarchy | <chare> | 2 | <pext> | <starting>exception classe hierarchy</heading> Throwable ---Exception ------IOException ------SQLException ------ClassNotFoundException ------RuntimeException ---------ArithmeticException ---------NullPointerException ---------NumberFormatException ---------IndexOutOfBoundsException ------------ArrayIndexOutOfBoundsException ------------StringIndexOutOfBoundsException ---Error ------StackOverflowError ------VirtualMachineError ------OutOfMemoryError </ending> | </end> |
| <hitle> | Checked Exception vs Unchecked Exception | <chare> | 2 | <pext> | <starting>Checked Exception vs Unchecked Exception</heading> 1) Checked Exception The classes that extend Throwable class except RuntimeException  and Error are known as checked exceptions,  e.g., IOException, SQLException, etc.  Checked exceptions are checked at compile-time. 2) Unchecked Exception The classes that extend RuntimeException are known as unchecked exceptions,  e.g., ArithmeticException, NullPointerException, etc.  Unchecked exceptions are not checked at compile-time. </ending> | </end> |
| <hitle> | Error and Exception base class | <chare> | 2 | <pext> | <starting>Error and Exception base class</heading> The Throwable class is the base class for Error and Exception. </ending> | </end> |
| <hitle> | finally block | <chare> | 2 | <pext> | <starting>finally block</heading> The "finally" block is used to execute the important code of the program.  It is executed whether an exception is handled or not.  In other words, we can say that finally block is the block  which is always executed.  Finally block follows try or catch block. If you don't handle the exception,  before terminating the program, JVM runs finally block, (if any).  The finally block is mainly used to place the cleanup code  such as closing a file or closing a connection.  Here, we must know that for each try block there can be zero  or more catch blocks, but only one finally block.  The finally block will not be executed if program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort). </ending> | </end> |
| <hitle> | throw vs throws | <chare> | 2 | <pext> | <starting>throw vs throws</heading> 1. The throw keyword is used to throw an exception explicitly. 1-1. The throws keyword is used to declare an exception. 2. The checked exceptions cannot be propagated with throw only. 2-1. The checked exception can be propagated with throws 3. The throw keyword is followed by an instance. 3-1. The throws keyword is followed by class. 4. The throw keyword is used within the method. 4-1. The throws keyword is used with the method signature. 5. You cannot throw multiple exceptions. 5-1. You can declare multiple exceptions,  e.g., public void method()throws IOException, SQLException. </ending> | </end> |
| <hitle> | exception propagation | <chare> | 2 | <pext> | <starting>exception propagation</heading> An exception is first thrown from the top of the stack and if it is not caught,  it drops down the call stack to the previous method,  If not caught there, the exception again drops down to the previous method,  and so on until they are caught  or until they reach the very bottom of the call stack.  This procedure is called exception propagation.  By default, checked exceptions are not propagated. </ending> | </end> |
| <hitle> | string pool | <chare> | 2 | <pext> | <starting>String pool</heading> String pool is the space reserved in the heap memory that can be used  to store the strings.  The main advantage of using the String pool is  whenever we create a string literal;  the JVM checks the "string constant pool" first.  If the string already exists in the pool,  a reference to the pooled instance is returned.  If the string doesn't exist in the pool,  a new string instance is created and placed in the pool.  Therefore, it saves the memory by avoiding the duplicacy. </ending> | </end> |
| <hitle> | immutable String | <chare> | 2 | <pext> | <starting>immutable String</heading> The simple meaning of immutable is unmodifiable or unchangeable.  In Java, String is immutable, i.e., once string object has been created,  its value can't be changed. </ending> | </end> |
| <hitle> | String objects immutable | <chare> | 2 | <pext> | <starting>String objects immutable</heading> Because Java uses the concept of the string literal.  Suppose there are five reference variables, all refer to one object.  If one reference variable changes the value of the object,  it will be affected by all the reference variables.  That is why string objects are immutable in java. </ending> | </end> |
| <hitle> | string literal | <chare> | 2 | <pext> | <starting>string literal</heading> To make Java more memory efficient  (because no new objects are created if it exists already in the string constant pool). </ending> | </end> |
| <hitle> | String vs StringBuffer | <chare> | 2 | <pext> | <starting>String vs StringBuffer</heading> 1) The String class is immutable.  1-1. The StringBuffer class is mutable. 2) The String is slow and consumes more memory  when you concat too many strings  because every time it creates a new instance. 2-1. The StringBuffer is fast and consumes less memory  when you concat strings. 3) The String class overrides the equals() method of Object class.  So you can compare the contents of two strings by equals() method.  3-1. The StringBuffer class doesn't override the equals() method of Object class. </ending> | </end> |
| <hitle> | immutable class | <chare> | 2 | <pext> | <starting>immutable class</heading> We can create an immutable class by defining a final class  having all of its members as final. </ending> | </end> |
| <hitle> | toString() | <chare> | 2 | <pext> | <starting>toString()</heading> The toString() method returns the string representation of an object.  If you print any object,  java compiler internally invokes the toString() method on the object.  So overriding the toString() method, returns the desired output,  it can be the state of an object, etc. depending upon your implementation.  By overriding the toString() method of the Object class,  we can return the values of the object,  so we don't need to write much code. </ending> | </end> |
| <hitle> | CharArray() | <chare> | 2 | <pext> | <starting>CharArray()</heading> String stays in the string pool until the garbage is collected.  If we store the password into a string,  it stays in the memory for a longer period,  and anyone having the memory-dump can extract the password as clear text. On the other hand, Using CharArray allows us to set it to blank  whenever we are done with the password.  It avoids the security threat with the string  by enabling us to control the memory. </ending> | </end> |
| <hitle> | java.util.regex | <chare> | 2 | <pext> | <starting>java.util.regex</heading> MatchResult Interface Matcher class Pattern class PatternSyntaxException class </ending> | </end> |
| <hitle> | metacharacters | <chare> | 2 | <pext> | <starting>metacharacters</heading> Metacharacters have the special meaning to the regular expression engine.  The metacharacters are ^, $, ., \*, +, etc.  The regular expression engine does not consider them  as the regular characters.  To enable the regular expression engine treating the metacharacters  as ordinary characters, we need to escape the metacharacters  with the backslash. </ending> | </end> |
| <hitle> | inner classes pros | <chare> | 2 | <pext> | <starting>inner classes pros</heading> 1. Nested classes represent a special type of relationship  that is it can access all the members of the outer class including private. 2. Nested classes are used to develop a more readable and maintainable code  because it logically groups classes and interfaces in one place only. 3. Code Optimization: It requires less code to write. </ending> | </end> |
| <hitle> | nested class | <chare> | 2 | <pext> | <starting>nested class</heading> The nested class can be defined as the class  which is defined inside another class or interface.  We use the nested class to logically group classes and interfaces in one place  so that it can be more readable and maintainable.  A nested class can access all the data members of the outer class  including private data members and methods. There are two types of nested classes,  static nested class, and non-static nested class.  The non-static nested class can also be called as inner-class </ending> | </end> |
| <hitle> | inner classes cons | <chare> | 2 | <pext> | <starting>inner classes cons</heading> 1. Inner classes increase the total number of classes used by the developer  and therefore increases the workload of JVM  since it has to perform some routine operations for those extra classes  which result in slower performance. 2. IDEs provide less support to the inner classes  as compare to the top level classes  and therefore it annoys the developers while working with inner classes. </ending> | </end> |
| <hitle> | inner classes types | <chare> | 2 | <pext> | <starting>inner classes types</heading> 1. Member Inner Class: A class created within class and outside method. 2. Anonymous Inner Class: A class created for implementing an interface  or extending class.  Its name is decided by the java compiler. 3. Local Inner Class: A class created within the method. </ending> | </end> |
| <hitle> | nested class vs inner class | <chare> | 2 | <pext> | <starting>nested class vs inner class</heading> Inner classes are non-static nested classes. In other words, we can say that inner classes are the part of nested classes. </ending> | </end> |
| <hitle> | anonymous inner class | <chare> | 2 | <pext> | <starting>anonymous inner class</heading> Anonymous inner classes are the classes that are automatically declared  and instantiated within an expression.  We cannot apply different access modifiers to them.  Anonymous class cannot be static,  and cannot define any static fields, method, or class.  In other words, we can say that it a class without the name  and can have only one object that is created by its definition. </ending> | </end> |
| <hitle> | nested interface | <chare> | 2 | <pext> | <starting>nested interface</heading> An Interface that is declared inside the interface or class  is known as the nested interface.  It is static by default.  The nested interfaces are used to group related interfaces  so that they can be easy to maintain.  The external interface or class must refer to the nested interface.  It can't be accessed directly.  The nested interface must be public if it is declared inside the interface  but it can have any access modifier if declared within the class. </ending> | </end> |
| <hitle> | interface in class | <chare> | 2 | <pext> | <starting>interface in class</heading> An interface can be defined within the class. It is called a nested interface. </ending> | </end> |
| <hitle> | inner class in interface | <chare> | 2 | <pext> | <starting>inner class in interface</heading> An Interface can have a class. they are static implicitly. </ending> | </end> |
| <hitle> | Garbage collection | <chare> | 2 | <pext> | <starting>Garbage collection</heading> Garbage collection is a process of reclaiming the unused runtime objects.  It is performed for memory management. In other words, we can say that It is the process of removing  unused objects from the memory to free up space  and make this space available for Java Virtual Machine.  Due to garbage collection java gives 0 as output to a variable  whose value is not set, i.e., the variable has been defined but not initialized. For this purpose, we were using free() function in the C language  and delete() in C++.  In Java, it is performed automatically.  So, java provides better memory management. </ending> | </end> |
| <hitle> | Garbage collection control | <chare> | 2 | <pext> | <starting>Garbage collection control</heading> Garbage collection is managed by JVM.  It is performed when there is not enough space in the memory  and memory is running low.  We can externally call the System.gc() for the garbage collection.  However, it depends upon the JVM whether to perform it or not. </ending> | </end> |
| <hitle> | object unreference | <chare> | 2 | <pext> | <starting>object unreference</heading> 1. By nulling the reference 2. By assigning a reference to another 3. By anonymous object etc. </ending> | </end> |
| <hitle> | Daemon thread | <chare> | 2 | <pext> | <starting>Daemon thread</heading> Garbage collector thread </ending> | </end> |
| <hitle> | Garbage collector thread | <chare> | 2 | <pext> | <starting>Garbage collector thread</heading> Daemon thread </ending> | </end> |
| <hitle> | finalize() | <chare> | 2 | <pext> | <starting>finalize()</heading> The finalize() method is invoked just before the object is garbage collected.  It is used to perform cleanup processing.  The Garbage collector of JVM collects only those objects  that are created by new keyword.  So if you have created an object without new, you can use the finalize method  to perform cleanup processing (destroying remaining objects).  The cleanup processing is the process to free up all the resources,  network which was previously used and no longer needed.  It is essential to remember that it is not a reserved keyword,  finalize method is present in the object class  hence it is available in every class  as object class is the superclass of every class in java.  Neither finalization nor garbage collection is guaranteed. </ending> | </end> |
| <hitle> | final, finally, finalize | <chare> | 2 | <pext> | <starting>final, finally, finalize</heading> 1) Final is used to apply restrictions on class, method, and variable.  The final class can't be inherited, final method can't be overridden,   and final variable value can't be changed.  1-1. Finally is used to place important code,  it will be executed whether an exception is handled or not. 1-2. Finalize is used to perform clean up processing  just before an object is garbage collected. 2) Final is a keyword. 2-1. Finally is a block. 3-1. Finalize is a method. </ending> | </end> |
| <hitle> | Runtime class | <chare> | 2 | <pext> | <starting>Runtime class</heading> Java Runtime class is used to interact with a java runtime environment.  Java Runtime class provides methods to execute a process, invoke GC,  get total and free memory, etc.  There is only one instance of java.lang.Runtime class is available  for one java application.  The Runtime.getRuntime() method returns the singleton instance  of Runtime class. </ending> | </end> |
| <hitle> | IO hierarchy | <chare> | 2 | <pext> | <starting>IO hierarchy </heading> OutputStream ---FileOutputStream ---ByteArrayOutputStream ---FilterOutputStream ------DataOutputStream ------BufferedOutputStream ------PrintStream ---PipedOutputStream ---ObjectOutputStream InputStream ---FileInputStream ---ByteArrayInputStream ---FilterInputStream ------DataInputStream ------BufferedInputStream ------PushBackInputStream ---PipedInputStream ---ObjectInputStream </ending> | </end> |
| <hitle> | IO stream | <chare> | 2 | <pext> | <starting>IO stream</heading> The stream is a sequence of data that flows from source to destination.  It is composed of bytes.  In Java, three streams are created for us automatically. System.out: standard output stream System.in: standard input stream System.err: standard error stream </ending> | </end> |
| <hitle> | super classe for stream | <chare> | 2 | <pext> | <starting>super classe for stream</heading> All the stream classes can be divided into two types of classes  that are ByteStream classes and CharacterStream Classes.  The ByteStream classes are further divided into InputStream classes  and OutputStream classes.  CharacterStream classes are also divided into  Reader classes and Writer classes.  The SuperMost classes for all the InputStream classes is  java.io.InputStream and for all the output stream classes is  java.io.OutPutStream.  Similarly, for all the reader classes, the super-most class is java.io.Reader,  and for all the writer classes, it is java.io.Writer. </ending> | </end> |
| <hitle> | BufferedInputStream vs BufferedOutputStream | <chare> | 2 | <pext> | <starting>BufferedInputStream / BufferedOutputStream</heading> Java BufferedOutputStream class is used for buffering an output stream.  It internally uses a buffer to store data.  It adds more efficiency than to write data directly into a stream.  So, it makes the performance fast.  Whereas, Java BufferedInputStream class is used  to read information from the stream.  It internally uses the buffer mechanism to make the performance fast. </ending> | </end> |
| <hitle> | FilterStreams | <chare> | 2 | <pext> | <starting>FilterStreams</heading> FilterStream classes are used to add additional functionalities  to the other stream classes.  FilterStream classes act like an interface which read the data from a stream,  filters it, and pass the filtered data to the caller.  The FilterStream classes provide extra functionalities  like adding line numbers to the destination file, etc. </ending> | </end> |
| <hitle> | IO filter | <chare> | 2 | <pext> | <starting>IO filter</heading> An I/O filter is an object that reads from one stream and writes to another,  usually altering the data in some way as it is passed  from one stream to another.  Many Filter classes that allow a user to make a chain  using multiple input streams.  It generates a combined effect on several filters. </ending> | </end> |
| <hitle> | serialization | <chare> | 2 | <pext> | <starting>serialization</heading> Serialization in Java is a mechanism of writing the state of an object  into a byte stream.  It is mainly used to travel object's state on the network. Serializable interface is used to perform serialization.  It is helpful when you require to save the state of a program to storage  such as the file.  At a later point of time, the content of this file can be restored  using deserialization.  It is also required to implement RMI(Remote Method Invocation).  With the help of RMI, it is possible to invoke the method of a Java object  on one machine to another machine.  How can you make a class serializable in Java? A class can become serializable by implementing the Serializable interface.  How can you avoid serialization in child class if the base class is implementing the Serializable interface? It is very tricky to prevent serialization of child class if the base class is intended to implement the Serializable interface. However, we cannot do it directly, but the serialization can be avoided by implementing the writeObject() or readObject() methods in the subclass and throw NotSerializableException from these methods.  </ending> | </end> |
| <hitle> | class serializable | <chare> | 2 | <pext> | <starting>class serializable</heading> A class can become serializable by implementing the Serializable interface. </ending> | </end> |
| <hitle> | serializable interface avoid in child | <chare> | 2 | <pext> | <starting>serializable interface avoid in child</heading> It is very tricky to prevent serialization of child class  if the base class is intended to implement the Serializable interface.  However, we cannot do it directly, but the serialization can be avoided  by implementing the writeObject() or readObject() methods in the subclass  and throw NotSerializableException from these methods. </ending> | </end> |
| <hitle> | deserialization | <chare> | 2 | <pext> | <starting>deserialization</heading> Deserialization is the process of reconstructing the object  from the serialized state.  It is the reverse operation of serialization.  An ObjectInputStream deserializes objects and primitive data  written using an ObjectOutputStream. </ending> | </end> |
| <hitle> | transient | <chare> | 2 | <pext> | <starting>transient</heading> If you define any data member as transient, it will not be serialized.  By determining transient keyword,  the value of variable need not persist when it is restored. </ending> | </end> |
| <hitle> | Externalizable | <chare> | 2 | <pext> | <starting>Externalizable</heading> The Externalizable interface is used to write the state of an object  into a byte stream in a compressed format. It is not a marker interface. </ending> | </end> |
| <hitle> | Serializable vs Externalizable | <chare> | 2 | <pext> | <starting>Serializable vs Externalizable</heading> 1. The Serializable interface does not have any method,  i.e., it is a marker interface.  1-1. The Externalizable interface contains is not a marker interface,  It contains two methods, i.e., writeExternal() and readExternal(). 2. It is used to "mark" Java classes  so that objects of these classes may get the certain capability.  2-1. The Externalizable interface provides control of the serialization logic  to the programmer. 3. Serializable, It is easy to implement but has the higher performance cost.  3-1. Externalizable, It is used to perform the serialization  and often result in better performance. 4. Serializable, No class constructor is called in serialization.  4-1. Externalizable,We must call a public default constructor  while using this interface. </ending> | </end> |
| <hitle> | reflection | <chare> | 2 | <pext> | <starting>reflection</heading> Reflection is the process of examining or modifying  the runtime behavior of a class at runtime.  The java.lang.Class class provides various methods  that can be used to get metadata,  examine and change the runtime behavior of a class.  The java.lang and java.lang.reflect packages provide classes for java reflection.  Reflection is used in: IDE (Integrated Development Environment), e.g., Eclipse, MyEclipse, NetBeans. Debugger Test Tools, etc. </ending> | </end> |
| <hitle> | autoboxing, unboxing | <chare> | 2 | <pext> | <starting>autoboxing, unboxing</heading> The autoboxing is the process of converting primitive data type  to the corresponding wrapper class object, eg., int to Integer.  The unboxing is the process of converting wrapper class object  to primitive data type.  For eg., integer to int.  Unboxing and autoboxing occur automatically in Java.  However, we can externally convert one into another  by using the methods like valueOf() or xxxValue(). It can occur whenever a wrapper class object is expected,  and primitive data type is provided or vice versa. 1. Adding primitive types into Collection like ArrayList in Java. 2. Creating an instance of parameterized classes  ,e.g., ThreadLocal which expect Type. 3. Java automatically converts primitive to object whenever one is required  and another is provided in the method calling. 4. When a primitive type is assigned to an object type. </ending> | </end> |
| <hitle> | object cloning | <chare> | 2 | <pext> | <starting>object cloning</heading> The object cloning is a way to create an exact copy of an object.  The clone() method of the Object class is used to clone an object.  The java.lang.Cloneable interface must be implemented by the class  whose object clone we want to create.  If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.  The clone() method is defined in the Object class.  The syntax of the clone() method is  protected Object clone() throws CloneNotSupportedException </ending> | </end> |
| <hitle> | object cloning pros/cons | <chare> | 2 | <pext> | <starting>object cloning pros/cons</heading> pros: 1. You don't need to write lengthy and repetitive codes.  Just use an abstract class with a 4- or 5-line long clone() method. 2. It is the easiest and most efficient way of copying objects,  especially if we are applying it to an already developed or an old project.  Just define a parent class, implement Cloneable in it,  provide the definition of the clone() method and the task will be done. 3. Clone() is the fastest way to copy the array.  cons: 1. To use the Object.clone() method,  we have to change many syntaxes to our code,  like implementing a Cloneable interface, defining the clone() method  and handling CloneNotSupportedException,  and finally, calling Object.clone(), etc. 2. We have to implement the Cloneable interface  while it does not have any methods in it.  We have to use it to tell the JVM that we can perform a clone() on our object. 3. Object.clone() is protected, so we have to provide our own clone()  and indirectly call Object.clone() from it. 4. Object.clone() does not invoke any constructor,  so we do not have any control over object construction. 5. If you want to write a clone method in a child class,  then all of its superclasses should define the clone() method in them  or inherit it from another parent class.  Otherwise, the super.clone() chain will fail. 6. Object.clone() supports only shallow copying,  but we will need to override it if we need deep cloning. </ending> | </end> |
| <hitle> | native method | <chare> | 2 | <pext> | <starting>native method</heading> A native method is a method that is implemented in a language other than Java. Natives methods are sometimes also referred to as foreign methods. </ending> | </end> |
| <hitle> | strictfp | <chare> | 2 | <pext> | <starting>strictfp</heading> Java strictfp keyword ensures that you will get the same result  on every platform if you perform operations in the floating-point variable.  The precision may differ from platform to platform that is  why java programming language has provided the strictfp keyword  so that you get the same result on every platform.  So, now you have better control over the floating-point arithmetic. </ending> | </end> |
| <hitle> | shallow copy | <chare> | 2 | <pext> | <starting>shallow copy</heading> Object cloning A shallow copy of an object is a new object whose instance variables are identical to the old object. For example, a shallow copy of a Set has the same members as the old Set and shares objects with the old Set through pointers. Shallow copies are sometimes said to use reference semantics. </ending> | </end> |
| <hitle> | singleton class | <chare> | 2 | <pext> | <starting>singleton class</heading> Singleton class is the class which can not be instantiated more than once.  To make a class singleton, we either make its constructor private  or use the static getInstance method. </ending> | </end> |
| <hitle> | Locale | <chare> | 2 | <pext> | <starting>Locale</heading> A Locale object represents a specific geographical, political, or cultural region.  This object can be used to get the locale-specific information  such as country name, language, variant, etc. </ending> | </end> |
| <hitle> | JavaBean | <chare> | 2 | <pext> | <starting>JavaBean</heading> JavaBean is a reusable software component  written in the Java programming language,  designed to be manipulated visually by a software development environment,  like JBuilder or VisualAge for Java. t.  A JavaBean encapsulates many objects into one object  so that we can access this object from multiple places.  Moreover, it provides the easy maintenance. A bean encapsulates many objects into one object  so that we can access this object from multiple places.  Moreover, it provides the easy maintenance. </ending> | </end> |
| <hitle> | multithreading | <chare> | 2 | <pext> | <starting>multithreading</heading> Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking.  It consumes less memory and gives the fast and efficient performance.  pros: 1. Threads share the same address space. 2. The thread is lightweight. 3. The cost of communication between the processes is low. </ending> | </end> |
| <hitle> | thread | <chare> | 2 | <pext> | <starting>thread</heading> A thread is a lightweight subprocess.  It is a separate path of execution  because each thread runs in a different stack frame.  A process may contain multiple threads.  Threads share the process resources, but still, they execute independently.  What do you understand by inter-thread communication? 1. The process of communication  between synchronized threads is termed as inter-thread communication. 2. Inter-thread communication is used to avoid thread polling in Java. 3. The thread is paused running in its critical section,  and another thread is allowed to enter (or lock) in the same critical section  to be executed. 4. It can be obtained by wait(), notify(), and notifyAll() methods.  Why must wait() method be called from the synchronized block? We must call the wait method otherwise it will throw  java.lang.IllegalMonitorStateException exception.  Moreover, we need wait() method for inter-thread communication  with notify() and notifyAll().  Therefore It must be present in the synchronized block for the proper  and correct communication.  Is it possible to start a thread twice? No, we cannot restart the thread, as once a thread started and executed,  it goes to the Dead state.  Therefore, if we try to start a thread twice,  it will give a runtimeException "java.lang.IllegalThreadStateException". </ending> | </end> |
| <hitle> | process vs thread | <chare> | 2 | <pext> | <starting>process vs thread</heading> 1. A Program in the execution is called the process whereas; A thread is a subset of the process 2. Processes are independent whereas threads are the subset of process. 3. Process have different address space in memory, while threads contain a shared address space. 4. Context switching is faster between the threads as compared to processes. 5. Inter-process communication is slower and expensive than inter-thread communication. 6. Any change in Parent process doesn't affect the child process whereas changes in parent thread can affect the child thread. </ending> | </end> |
| <hitle> | inter-thread communication | <chare> | 2 | <pext> | <starting>inter-thread communication</heading> 1. The process of communication between synchronized threads  is termed as inter-thread communication. 2. Inter-thread communication is used to avoid thread polling in Java. 3. The thread is paused running in its critical section,  and another thread is allowed to enter (or lock) in the same critical section  to be executed. 4. It can be obtained by wait(), notify(), and notifyAll() methods. </ending> | </end> |
| <hitle> | wait() | <chare> | 2 | <pext> | <starting>wait()</heading> The wait() method is provided by the Object class in Java.  This method is used for inter-thread communication in Java.  The java.lang.Object.wait() is used to pause the current thread,  and wait until another thread does not call the notify() or notifyAll() method. </ending> | </end> |
| <hitle> | multithreading pros | <chare> | 2 | <pext> | <starting>multithreading pros</heading> 1. Multithreading allows an application/program  to be always reactive for input, even already running  with some background tasks 2. Multithreading allows the faster execution of tasks,  as threads execute independently. 3. Multithreading provides better utilization of cache memory  as threads share the common memory resources. 4. Multithreading reduces the number of the required server  as one server can execute multiple threads at a time. </ending> | </end> |
| <hitle> | thread lifecycle states | <chare> | 2 | <pext> | <starting>thread lifecycle states</heading> 1. New: In this state, a Thread class object is created using a new operator,  but the thread is not alive.  Thread doesn't start until we call the start() method. 2. Runnable: In this state, the thread is ready  to run after calling the start() method.  However, the thread is not yet selected by the thread scheduler. 3. Running: In this state, the thread scheduler picks the thread  from the ready state, and the thread is running. 4. Waiting/Blocked: In this state, a thread is not running but still alive,  or it is waiting for the other thread to finish. 5. Dead/Terminated: A thread is in terminated or dead state  when the run() method exits. </ending> | </end> |
| <hitle> | preemptive scheduling vs time slicing | <chare> | 2 | <pext> | <starting>preemptive scheduling vs time slicing</heading> Under preemptive scheduling, the highest priority task executes  until it enters the waiting or dead states  or a higher priority task comes into existence.  Under time slicing, a task executes for a predefined slice of time  and then reenters the pool of ready tasks.  The scheduler then determines which task should execute next,  based on priority and other factors. </ending> | </end> |
| <hitle> | context switching | <chare> | 2 | <pext> | <starting>context switching</heading> In Context switching the state of the process (or thread) is stored  so that it can be restored  and execution can be resumed from the same point later.  Context switching enables the multiple processes to share the same CPU. </ending> | </end> |
| <hitle> | Thread class vs Runnable interface | <chare> | 2 | <pext> | <starting>Thread class vs Runnable interface</heading> 1. By extending the Thread class, we cannot extend any other class,  as Java does not allow multiple inheritances  while implementing the Runnable interface;  we can also extend other base class(if required). 2. By extending the Thread class,  each of thread creates the unique object and associates with it  while implementing the Runnable interface;  multiple threads share the same object 3. Thread class provides various inbuilt methods  such as getPriority(), isAlive and many more  while the Runnable interface provides a single method, i.e., run(). </ending> | </end> |
| <hitle> | join() | <chare> | 2 | <pext> | <starting>join()</heading> The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing  until the thread it joins with completes its task.  Join method is overloaded in Thread class in the following ways. public void join()throws InterruptedException public void join(long milliseconds)throws InterruptedException </ending> | </end> |
| <hitle> | sleep() | <chare> | 2 | <pext> | <starting>sleep()</heading> The sleep() method in java is used to block a thread for a particular time,  which means it pause the execution of a thread for a specific time.  There are two methods of doing so. When we call the sleep() method, it pauses the execution  of the current thread for the given time and gives priority to another thread. Moreover, when the waiting time completed then again previous thread  changes its state from waiting to runnable and comes in running state,  and the whole process works so on till the execution doesn't complete. </ending> | </end> |
| <hitle> | wait() vs sleep() | <chare> | 2 | <pext> | <starting>wait() vs sleep()</heading> 1) The wait() method is defined in Object class.  1-1. The sleep() method is defined in Thread class. 2) The wait() method releases the lock.  2-1. The sleep() method doesn't release the lock. </ending> | </end> |
| <hitle> | daemon threads | <chare> | 2 | <pext> | <starting>daemon threads</heading> The daemon threads are the low priority threads  that provide the background support and services to the user threads. Daemon thread gets automatically terminated by the JVM  if the program remains with the daemon thread only,  and all other user threads are ended/died.  There are two methods for daemon thread available in the Thread class: public void setDaemon(boolean status): It used to mark the thread daemon thread or a user thread. public boolean isDaemon(): It checks the thread is daemon or not. </ending> | </end> |
| <hitle> | shutdown hook | <chare> | 2 | <pext> | <starting>shutdown hook</heading> The shutdown hook is a thread that is invoked implicitly  before JVM shuts down.  So we can use it to perform clean up the resource or save the state  when JVM shuts down normally or abruptly.  We can add shutdown hook by using the following method: Some important points about shutdown hooks are : 1. Shutdown hooks initialized but can only be started  when JVM shutdown occurred. 2. Shutdown hooks are more reliable than the finalizer()  because there are very fewer chances that shutdown hooks not run. 3. The shutdown hook can be stopped  by calling the halt(int) method of Runtime class. </ending> | </end> |
| <hitle> | thread interrupt | <chare> | 2 | <pext> | <starting>thread interrupt</heading> We should interrupt a thread  when we want to break out the sleep or wait state of a thread.  We can interrupt a thread by calling the interrupt()  throwing the InterruptedException. </ending> | </end> |
| <hitle> | synchronization | <chare> | 2 | <pext> | <starting>synchronization</heading> Synchronization is the capability to control the access of multiple threads  to any shared resource.  It is used: 1. To prevent thread interference. 2. To prevent consistency problem. When the multiple threads try to do the same task,  there is a possibility of an erroneous result, hence to remove this issue,  Java uses the process of synchronization which allows only one thread  to be executed at a time.  Synchronization can be achieved in three ways: 1. by the synchronized method 2. by synchronized block 3. by static synchronization </ending> | </end> |
| <hitle> | Synchronized block | <chare> | 2 | <pext> | <starting>Synchronized block</heading> The Synchronized block can be used to perform synchronization  on any specific resource of the method.  Only one thread at a time can execute on a particular resource,   and all other threads which attempt to enter the synchronized block  are blocked. 1. Synchronized block is used to lock an object for any shared resource. 2. The scope of the synchronized block is limited to the block  on which, it is applied. Its scope is smaller than a method.  Java object can be locked down for exclusive use by a given thread. You can lock an object by putting it in a "synchronized" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it. </ending> | </end> |
| <hitle> | static synchronization | <chare> | 2 | <pext> | <starting>static synchronization</heading> If you make any static method as synchronized,  the lock will be on the class not on the object.  If we use the synchronized keyword before a method  so it will lock the object (one thread can access an object at a time)  but if we use static synchronized so it will lock a class  (one thread can access a class at a time). </ending> | </end> |
| <hitle> | notify() vs notifyAll() | <chare> | 2 | <pext> | <starting>notify() vs notifyAll()</heading> The notify() is used to unblock one waiting thread  whereas notifyAll() method is used to unblock all the threads in waiting state. </ending> | </end> |
| <hitle> | deadlock | <chare> | 2 | <pext> | <starting>deadlock</heading> Deadlock is a situation in which every thread is waiting for a resource  which is held by some other waiting thread.  In this situation, Neither of the thread executes nor it gets the chance to be executed.  Instead, there exists a universal waiting state among all the threads.  Deadlock is a very complicated situation which can break our code at runtime. </ending> | </end> |
| <hitle> | deadlock condition | <chare> | 2 | <pext> | <starting>deadlock condition</heading> We can detect the deadlock condition by running the code on cmd  and collecting the Thread Dump, and if any deadlock is present in the code,  then a message will appear on cmd. Ways to avoid the deadlock condition in Java: 1. Avoid Nested lock: Nested lock is the common reason for deadlock  as deadlock occurs when we provide locks to various threads  so we should give one lock to only one thread at some particular time. 2. Avoid unnecessary locks: we must avoid the locks which are not required. 3. Using thread join: Thread join helps to wait for a thread  until another thread doesn't finish its execution  so we can avoid deadlock by maximum use of join method. </ending> | </end> |
| <hitle> | Thread Scheduler | <chare> | 2 | <pext> | <starting>Thread Scheduler</heading> In Java, when we create the threads, they are supervised  with the help of a Thread Scheduler, which is the part of JVM.  Thread scheduler is only responsible for deciding  which thread should be executed.  Thread scheduler uses two mechanisms for scheduling the threads: Preemptive and Time Slicing. Java thread scheduler used for: 1. It selects the priority of the thread. 2. It determines the waiting time for a thread 3. It checks the Nature of thread </ending> | </end> |
| <hitle> | thread safety | <chare> | 2 | <pext> | <starting>thread safety</heading> If a method or class object can be used by multiple threads  at a time without any race condition, then the class is thread-safe.  Thread safety is used to make a program safe  to use in multithreaded programming.  How do do it? 1. Synchronization 2. Using Volatile keyword 3. Using a lock based mechanism 4. Use of atomic wrapper classes </ending> | </end> |
| <hitle> | race-condition | <chare> | 2 | <pext> | <starting>race-condition</heading> A Race condition is a problem which occurs in the multithreaded programming  when various threads execute simultaneously accessing a shared resource  at the same time.  The proper use of synchronization can avoid the Race condition. </ending> | </end> |
| <hitle> | volatile | <chare> | 2 | <pext> | <starting>volatile</heading> Volatile keyword is used in multithreaded programming  to achieve the thread safety, as a change in one volatile variable is visible  to all other threads so one variable can be used by one thread at a time. </ending> | </end> |
| <hitle> | thread pool | <chare> | 2 | <pext> | <starting>thread pool</heading> 1. Java Thread pool represents a group of worker threads,  which are waiting for the task to be allocated. 2. Threads in the thread pool are supervised by the service provider  which pulls one thread from the pool and assign a job to it. 3. After completion of the given task, thread again came to the thread pool. 4. The size of the thread pool depends on the total number of threads  kept at reserve for execution.  The advantages of the thread pool are : 1. Using a thread pool, performance can be enhanced. 2. Using a thread pool, better system stability can occur. </ending> | </end> |
| <hitle> | concurrency API | <chare> | 2 | <pext> | <starting>concurrency API</heading> Concurrency API can be developed using the class and interfaces of  java.util.Concurrent package.  Classes and interfaces in java.util.Concurrent package are: Executor FarkJoinPool ExecutorService ScheduledExecutorService Future TimeUnit(Enum) CountDownLatch CyclicBarrier Semaphore ThreadFactory BlockingQueue DelayQueue Locks Phaser </ending> | </end> |
| <hitle> | Executor interface | <chare> | 2 | <pext> | <starting>Executor interface</heading> The Executor Interface provided by the package java.util.concurrent  is the simple interface used to execute the new task.  The execute() method of Executor interface is used  to execute some given command. </ending> | </end> |
| <hitle> | BlockingQueue | <chare> | 2 | <pext> | <starting>BlockingQueue</heading> The java.util.concurrent.BlockingQueue is the subinterface of Queue  that supports the operations such as waiting for the space availability  before inserting a new value or waiting for the queue  to become non-empty before retrieving an element from it. BlockingQueue is an interface which extends the Queue interface.  It provides concurrency in the operations like retrieval, insertion, deletion.  While retrieval of any element, it waits for the queue to be non-empty.  While storing the elements, it waits for the available space.  BlockingQueue cannot contain null elements,  and implementation of BlockingQueue is thread-safe. </ending> | </end> |
| <hitle> | Callable interface vs Runnable interface | <chare> | 2 | <pext> | <starting>Callable interface vs Runnable interface</heading> The Callable interface and Runnable interface both are used by the classes  which wanted to execute with multiple threads.  However, there are two main differences between the both : 1. A Callable <V> interface can return a result,  whereas the Runnable interface cannot return any result. 2. A Callable <V> interface can throw a checked exception,  whereas the Runnable interface cannot throw checked exception. 3. A Callable <V> interface cannot be used before the Java 5  whereas the Runnable interface can be used. </ending> | </end> |
| <hitle> | Atomic action | <chare> | 2 | <pext> | <starting>Atomic action</heading> 1. The Atomic action is the operation which can be performed  in a single unit of a task without any interference of the other operations. 2. The Atomic action cannot be stopped in between the task.  Once started it fill stop after the completion of the task only. 3. An increment operation such as a++ does not allow an atomic action. 4. All reads and writes operation for the primitive variable  (except long and double) are the atomic operation. 5. All reads and writes operation for the volatile variable  (including long and double) are the atomic operation. 6. The Atomic methods are available in java.util.Concurrent package. </ending> | </end> |
| <hitle> | lock interface | <chare> | 2 | <pext> | <starting>lock interface</heading> The java.util.concurrent.locks.Lock interface is used  as the synchronization mechanism.  It works similar to the synchronized block.  Differences between the lock and synchronized block are: 1. Lock interface provides the guarantee of sequence  in which the waiting thread will be given the access,  whereas the synchronized block doesn't guarantee it. 2. Lock interface provides the option of timeout if the lock is not granted  whereas the synchronized block doesn't provide that. 3. The methods of Lock interface, i.e., Lock() and Unlock() can be called  in different methods  whereas single synchronized block must be fully contained in a single method. </ending> | </end> |
| <hitle> | ExecutorService | <chare> | 2 | <pext> | <starting>ExecutorService</heading> The ExecutorService Interface is the subinterface of Executor interface  and adds the features to manage the lifecycle. </ending> | </end> |
| <hitle> | Synchronous vs Asynchronous | <chare> | 2 | <pext> | <starting>Synchronous vs Asynchronous</heading> Synchronous programming: In Synchronous programming model,  a thread is assigned to complete a task and hence thread started working on it,  and it is only available for other tasks once it will end the assigned task. Asynchronous Programming: In Asynchronous programming,  one job can be completed by multiple threads and hence  it provides maximum usability of the various threads. </ending> | </end> |
| <hitle> | Callable vs Future | <chare> | 2 | <pext> | <starting>Callable vs Future</heading> Java Callable interface: In Java5 callable interface was provided by the package  java.util.concurrent.  It is similar to the Runnable interface but it can return a result,  and it can throw an Exception.  It also provides a run() method for execution of a thread.  Java Callable can return any object as it uses Generic. public interface Callable<V>  Java Future interface:  Java Future interface gives the result of a concurrent process.  The Callable interface returns the object of java.util.concurrent.Future. Java Future provides following methods for implementation: 1. cancel(boolean mayInterruptIfRunning):  It is used to cancel the execution of the assigned task. 2. get(): It waits for the time if execution not completed  and then retrieved the result. 3. isCancelled(): It returns the Boolean value as it returns true  if the task was canceled before the completion. 4. isDone(): It returns true if the job is completed successfully  else returns false. </ending> | </end> |
| <hitle> | ScheduledExecutorService vs ExecutorService interface | <chare> | 2 | <pext> | <starting>ScheduledExecutorService vs ExecutorService interface</heading> ExecutorServcie and ScheduledExecutorService both are the interfaces  of java.util.Concurrent package but scheduledExecutorService provides  some additional methods to execute the Runnable and Callable tasks  with the delay or every fixed time period. </ending> | </end> |
| <hitle> | FutureTask | <chare> | 2 | <pext> | <starting>FutureTask</heading> Java FutureTask class provides a base implementation of the Future interface. The result can only be obtained if the execution of one task is completed,  and if the computation is not achieved then get method will be blocked.  If the execution is completed, then it cannot be re-started  and can't be canceled. </ending> | </end> |
| <hitle> | Collection framework | <chare> | 2 | <pext> | <starting>Collection framework</heading> Collection Framework is a combination of classes and interface,  which is used to store and manipulate the data in the form of objects.  It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc.  and interfaces such as List, Queue, Set, etc. for this purpose. </ending> | </end> |
| <hitle> | array vs collection | <chare> | 2 | <pext> | <starting>array vs collection</heading> Array and Collection are somewhat similar  regarding storing the references of objects and manipulating the data,  but they differ in many ways.  The main differences are: 1. Arrays are always of fixed size,  i.e., a user can not increase or decrease the length of the array  according to their requirement or at runtime,  but In Collection, size can be changed dynamically as per need. 2. Arrays can only store homogeneous or similar type objects,  but in Collection, heterogeneous objects can be stored. 3. Arrays cannot provide the ?ready-made? methods for user requirements  as sorting, searching, etc.  but Collection includes readymade methods to use. </ending> | </end> |
| <hitle> | Collection | <chare> | 2 | <pext> | <starting>Collection</heading> Collection (java.util.Collection) is the primary interface,  and every collection must implement this interface. </ending> | </end> |
| <hitle> | List | <chare> | 2 | <pext> | <starting>List</heading> List interface extends the Collection interface,  and it is an ordered collection of objects.  It contains duplicate elements.  It also allows random access of elements. </ending> | </end> |
| <hitle> | Set | <chare> | 2 | <pext> | <starting>Set</heading> Set (java.util.Set) interface is a collection  which cannot contain duplicate elements.  It can only include inherited methods of Collection interface </ending> | </end> |
| <hitle> | Queue | <chare> | 2 | <pext> | <starting>Queue</heading> Queue (java.util.Queue) interface defines queue data structure,  which stores the elements in the form FIFO (first in first out). </ending> | </end> |
| <hitle> | Dequeue | <chare> | 2 | <pext> | <starting>Dequeue</heading> Dequeue interface: it is a double-ended-queue.  It allows the insertion and removal of elements from both ends.  It implants the properties of both Stack and queue  so it can perform LIFO (Last in first out) stack  and FIFO (first in first out) queue, operations. </ending> | </end> |
| <hitle> | Map java | <chare> | 2 | <pext> | <starting>Map</heading> Map interface: A Map represents a key, value pair storage of elements.  Map interface does not implement the Collection interface.  It can only contain a unique key but can have duplicate elements.  There are two interfaces which implement Map in java  that are Map interface and Sorted Map. </ending> | </end> |
| <hitle> | ArrayList vs Vector | <chare> | 2 | <pext> | <starting>ArrayList vs Vector</heading> 1) ArrayList is not synchronized. / Vector is synchronized. 2) ArrayList is not a legacy class. / Vector is a legacy class. 3) ArrayList increases its size by 50% of the array size.  3-1. Vector increases its size by doubling the array size. 4) ArrayList is not thread-safe as it is not synchronized.  4-1. Vector list is thread-safe as it's every method is synchronized. </ending> | </end> |
| <hitle> | ArrayList vs LinkedList | <chare> | 2 | <pext> | <starting>ArrayList vs LinkedList</heading> 1) ArrayList uses a dynamic array.  1-1. LinkedList uses a doubly linked list. 2) ArrayList is not efficient for manipulation because too much is required.  2-1. LinkedList is efficient for manipulation. 3) ArrayList is better to store and fetch data.  3-1. LinkedList is better to manipulate data. 4) ArrayList provides random access.  3-1. LinkedList does not provide random access. 5) ArrayList takes less memory overhead as it stores only object.  5-1. LinkedList takes more memory overhead,  as it stores the object as well as the address of that object. </ending> | </end> |
| <hitle> | Iterator vs ListIterator | <chare> | 2 | <pext> | <starting>Iterator vs ListIterator</heading> Iterator traverses the elements in the forward direction only  whereas ListIterator traverses the elements  into forward and backward direction. 1) The Iterator traverses the elements in the forward direction only.  1-1. ListIterator traverses the elements in backward  and forward directions both. 2) The Iterator can be used in List, Set, and Queue.  2-1. ListIterator can be used in List only. 3) The Iterator can only perform remove operation  while traversing the collection.  3-1. ListIterator can perform ?add,? ?remove,? and ?set? operation  while traversing the collection. </ending> | </end> |
| <hitle> | Iterator vs Enumeration | <chare> | 2 | <pext> | <starting>Iterator vs Enumeration</heading> 1) The Iterator can traverse legacy and non-legacy elements.  1-1. Enumeration can traverse only legacy elements. 2) The Iterator is fail-fast.  2-1 Enumeration is not fail-fast. 3) The Iterator is slower than Enumeration.  3-1. Enumeration is faster than Iterator. 4) The Iterator can perform remove operation while traversing the collection.  4-1. The Enumeration can perform only traverse operation on the collection. </ending> | </end> |
| <hitle> | List vs Set | <chare> | 2 | <pext> | <starting>List vs Set</heading> The List and Set both extend the collection interface.  However, there are some differences between the both which are listed below. 1. The List can contain duplicate elements whereas Set includes unique items. 2. The List is an ordered collection which maintains the insertion order  whereas Set is an unordered collection  which does not preserve the insertion order. 3. The List interface contains a single legacy class which is Vector class  whereas Set interface does not have any legacy class. 4. The List interface can allow n number of null values  whereas Set interface only allows a single null value. </ending> | </end> |
| <hitle> | HashSet vs TreeSet | <chare> | 2 | <pext> | <starting>HashSet vs TreeSet</heading> The HashSet and TreeSet, both classes, implement Set interface.  The differences between the both are listed below. 1. HashSet maintains no order whereas TreeSet maintains ascending order. 2. HashSet impended by hash table  whereas TreeSet implemented by a Tree structure. 3. HashSet performs faster than TreeSet. 4. HashSet is backed by HashMap  whereas TreeSet is backed by TreeMap. </ending> | </end> |
| <hitle> | Set vs Map | <chare> | 2 | <pext> | <starting>Set vs Map</heading> 1. Set contains values only whereas Map contains key and values both. 2. Set contains unique values  whereas Map can contain unique Keys with duplicate values. 3. Set holds a single number of null value  whereas Map can include a single null key with n number of null values. </ending> | </end> |
| <hitle> | HashSet vs HashMap | <chare> | 2 | <pext> | <starting>HashSet vs HashMap</heading> 1. HashSet contains only values whereas HashMap includes the entry.  HashSet can be iterated, but HashMap needs to convert into Set to be iterated. 2. HashSet implements Set interface  whereas HashMap implements the Map interface 3. HashSet cannot have any duplicate value  whereas HashMap can contain duplicate values with unique keys. 4. HashSet contains the only single number of null value  whereas HashMap can hold a single null key with n number of null values. </ending> | </end> |
| <hitle> | HashMap vs TreeMap | <chare> | 2 | <pext> | <starting>HashMap vs TreeMap</heading> 1. HashMap maintains no order, but TreeMap maintains ascending order. 2. HashMap is implemented by hash table  whereas TreeMap is implemented by a Tree structure. 3. HashMap can be sorted by Key or value  whereas TreeMap can be sorted by Key. 4. HashMap may contain a null key with multiple null values  whereas TreeMap cannot hold a null key but can have multiple null values. </ending> | </end> |
| <hitle> | HashMap vs Hashtable | <chare> | 2 | <pext> | <starting>HashMap vs Hashtable</heading> 1) HashMap is not synchronized.  1-1. Hashtable is synchronized. 2) HashMap can contain one null key and multiple null values.  2-1 Hashtable cannot contain any null key or null value. 3) HashMap is not ?thread-safe,? so it is useful for non-threaded applications. 3-1. Hashtable is thread-safe, and it can be shared between various threads. 4) HashMap inherits the AbstractMap class. 4-1. Hashtable inherits the Dictionary class. </ending> | </end> |
| <hitle> | Collection vs Collections | <chare> | 2 | <pext> | <starting>Collection vs Collections</heading> 1. The Collection is an interface whereas Collections is a class. 2. The Collection interface provides the standard functionality of data structure  to List, Set, and Queue.  However, Collections class is to sort and synchronize the collection elements. 3. The Collection interface provides the methods  that can be used for data structure  whereas Collections class provides the static methods  which can be used for various operation on a collection. </ending> | </end> |
| <hitle> | Comparable vs Comparator | <chare> | 2 | <pext> | <starting>Comparable vs Comparator</heading> 1) Comparable provides only one sort of sequence.  1-1. The Comparator provides multiple sorts of sequences. 2) It provides one method named compareTo().  2-1. It provides one method named compare(). 3) It is found in java.lang package.  3-1. It is located in java.util package. 4) If we implement the Comparable interface, The actual class is modified.  4-1 The actual class is not changed. </ending> | </end> |
| <hitle> | hashCode() | <chare> | 2 | <pext> | <starting> hashCode()</heading> The hashCode() method returns a hash code value (an integer number). The hashCode() method returns the same integer number if two keys  (by calling equals() method) are identical. However, it is possible that two hash code numbers  can have different or the same keys. If two objects do not produce an equal result by using the equals() method,  then the hashcode() method will provide the different integer result  for both the objects. </ending> | </end> |
| <hitle> | equals() | <chare> | 2 | <pext> | <starting>equals()</heading> The equals method is used to check whether two objects are the same or not. It needs to be overridden if we want to check the objects  based on the property. For example, Employee is a class  that has 3 data members: id, name, and salary.  However, we want to check the equality of employee object by the salary. Then, we need to override the equals() method. </ending> | </end> |
| <hitle> | synchronize List, Set, Map | <chare> | 2 | <pext> | <starting>synchronize List, Set, Map</heading> Collections class provides methods to make List,  Set or Map elements as synchronized: public static List synchronizedList(List l){} public static Set synchronizedSet(Set s){} public static SortedSet synchronizedSortedSet(SortedSet s){} public static Map synchronizedMap(Map m){} public static SortedMap synchronizedSortedMap(SortedMap m){} </ending> | </end> |
| <hitle> | generic collection pros | <chare> | 2 | <pext> | <starting>generic collection pros</heading> 1. If we use the generic class, we don't need typecasting. 2. It is type-safe and checked at compile time. 3. Generic confirms the stability of the code  by making it bug detectable at compile time. </ending> | </end> |
| <hitle> | hash-collision | <chare> | 2 | <pext> | <starting>hash-collision</heading> Two different keys with the same hash value are known as hash-collision.  Two separate entries will be kept in a single hash bucket to avoid the collision.  There are two ways to avoid hash-collision. Separate Chaining Open Addressing </ending> | </end> |
| <hitle> | Dictionary | <chare> | 2 | <pext> | <starting>Dictionary</heading> The Dictionary class provides the capability to store key-value pairs. </ending> | </end> |
| <hitle> | fail-fast | <chare> | 2 | <pext> | <starting>fail-fast</heading> The Iterator in java which immediately throws  ConcurrentmodificationException, if any structural modification occurs in,  is called as a Fail-fast iterator.  Fail-fats iterator does not require any extra space in memory. </ending> | </end> |
| <hitle> | Array vs ArrayList | <chare> | 2 | <pext> | <starting>Array vs ArrayList</heading> 1 The Array is of fixed size, means we cannot resize the array as per need.  ArrayList is not of the fixed size we can change the size dynamically. 2 Arrays are of the static type. ArrayList is of dynamic size. 3 Arrays can store primitive data types as well as objects.  ArrayList cannot store the primitive data types it can only store the objects. </ending> | </end> |
| <hitle> | ArrayList remove duplicates | <chare> | 2 | <pext> | <starting>ArrayList remove duplicates</heading> There are two ways to remove duplicates from the ArrayList. 1. Using HashSet: By using HashSet we can remove the duplicate element  from the ArrayList, but it will not then preserve the insertion order. 2. Using LinkedHashSet: We can also maintain the insertion order  by using LinkedHashSet instead of HashSet.  The Process to remove duplicate elements from ArrayList using the LinkedHashSet: 1. Copy all the elements of ArrayList to LinkedHashSet. 2. Empty the ArrayList using clear() method,  which will remove all the elements from the list. 3. Now copy all the elements of LinkedHashset to ArrayList. </ending> | </end> |
| <hitle> | ArrayList synchronize | <chare> | 2 | <pext> | <starting>ArrayList synchronize</heading> 1. Using Collections.synchronizedList() method 2. Using CopyOnWriteArrayList<T> </ending> | </end> |
| <hitle> | Java data type | <chare> | 2 | <pext> | Data types specify the different sizes and values that can be stored  in the variable.  There are two types of data types in Java:  1. Primitive data types: The primitive data types  include boolean, char, byte, short, int, long, float and double.  2. Non-primitive data types:  Include String, Classes, Interfaces, and Arrays. | </end> |
| <hitle> | OOPs | <chare> | 2 | <pext> | OOP is an abbreviation of Object-Oriented Programming.  OOP is a paradigm that provides many concepts,  such as inheritance, data binding, polymorphism, etc.  Object means a real-world entity such as a car, a train, etc.  Object-Oriented Programming is a methodology or paradigm  to design a program using classes and objects.  OOP simplifies software development and maintenance  by providing some concepts:  1. Object  2. Class  3. Inheritance  4. Polymorphism  5. Abstraction  6. Encapsulation  Apart from these concepts, there are some other terms  which are used in Object-Oriented design:  1. Coupling  2. Cohesion  3. Association  4. Aggregation  5. Composition | </end> |
| <hitle> | Inheritance | <chare> | 3 | <pext> | When one object acquires all the properties and behaviors  of a parent object, it is known as inheritance.  It provides code reusability.  It is used to achieve runtime polymorphism. | </end> |
| <hitle> | Polymorphism | <chare> | 3 | <pext> | If one task is performed in different ways,  it is known as polymorphism.  For example: to convince the customer differently,  to draw something, for example, shape, triangle, rectangle, etc.  In Java, we use method overloading and method overriding  to achieve polymorphism. | </end> |
| <hitle> | Types of polymorphism Kotlin | <chare> | 3 | <pext> | Polymorphism is mainly divided into two types:  1. Compile-time Polymorphism  2. Runtime Polymorphism  1. Compile-time Polymorphism  In compile-time polymorphism, the name functions, that is,  the signature remains the same but parameters  or return type is different.  At compile time, the compiler then resolves  which functions we are trying to call based  on the type of parameters and more.  fun main (args: Array<String>) {  println(doubleof(4))  println(doubleof(4.3))  println(doubleof(4.323))  }  fun doubleof(a: Int):Int {  return 2\*a  }  fun doubleOf(a:Float):Float {  return 2\*a  }  fun doubleof(a:Double):Double {  return 2.00\*a  }  2. Runtime Polymorphism  In run-time polymorphism, the compiler resolves a call  to overridden/overloaded methods at runtime.  We can achieve run-time polymorphism using method overriding.  We can see an example where we extend a superclass and override  one of its member methods.  fun main(args: Array<string>){  var a = Sup()  a.method1()  a.method2()  var b = Sum()  b.method1()  b.method2()  }  open class Sup{  open fun method1(){  println("printing method 1 from inside Sup")  }  fun method2(){  println("printing method 2 from inside Sup")  }  }  class Sum:Sup(){  override fun method1(){  println("printing method 1 from inside Sum")  }  } | </end> |
| <hitle> | Abstraction | <chare> | 3 | <pext> | Hiding internal details and showing functionality  is known as abstraction.  For example phone call, we don't know the internal processing.  In Java, we use abstract class and interface to achieve abstraction. | </end> |
| <hitle> | Abstraction | <chare> | 3 | <pext> | Hiding internal details and showing functionality  is known as abstraction.  For example phone call, we don't know the internal processing.  In Java, we use abstract class and interface to achieve abstraction. | </end> |
| <hitle> | Encapsulation | <chare> | 3 | <pext> | Binding or wrapping code and data together into a single unit  are known as encapsulation.  For example, a capsule, it is wrapped with different medicines.  A java class is the example of encapsulation.  Java bean is the fully encapsulated class because  all the data members are private here. | </end> |
| <hitle> | Coupling | <chare> | 3 | <pext> | Coupling refers to the knowledge or information or dependency  of another class.  It arises when classes are aware of each other.  If a class has the details information of another class,  there is strong coupling.  In Java, we use private, protected, and public modifiers to display  the visibility level of a class, method, and field.  We can use interfaces for the weaker coupling because  there is no concrete implementation. | </end> |
| <hitle> | Cohesion | <chare> | 3 | <pext> | Cohesion refers to the level of a component which performs  a single well-defined task.  A single well-defined task is done by a highly cohesive method.  The weakly cohesive method will split the task into separate parts.  The java.io package is a highly cohesive package because  it has I/O related classes and interface.  However, the java.util package is a weakly cohesive package  because it has unrelated classes and interfaces. | </end> |
| <hitle> | Association | <chare> | 3 | <pext> | Association represents the relationship between the objects.  Here, one object can be associated with one object or many objects.  There can be four types of association between the objects:  1. One to One  2. One to Many  3. Many to One, and  4. Many to Many  Association can be undirectional or bidirectional.  Let's understand the relationship with real-time examples.  For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many).  Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many). | </end> |
| <hitle> | Aggregation | <chare> | 3 | <pext> | Aggregation is a way to achieve Association.  Aggregation represents the relationship where one object  contains other objects as a part of its state.  It represents the weak relationship between objects.  It is also termed as a has-a relationship in Java.  Like, inheritance represents the is-a relationship.  It is another way to reuse objects. | </end> |
| <hitle> | Composition | <chare> | 3 | <pext> | The composition is also a way to achieve Association.  The composition represents the relationship  where one object contains other objects as a part of its state.  There is a strong relationship between the containing object  and the dependent object.  It is the state where containing objects do not have  an independent existence.  If you delete the parent object,  all the child objects will be deleted automatically. | </end> |
| <hitle> | object-oriented vs object-based oop vs obp programming | <chare> | 3 | <pext> | Object-based programming language follows all the features  of OOPs except Inheritance.  JavaScript and VBScript are examples of  object-based programming languages. | </end> |
| <hitle> | Final keyword in java | <chare> | 2 | <pext> | The final keyword is a non-access modifier used for  classes, attributes and methods, which makes them non-changeable  and impossible to inherit or override.  The final keyword is useful when we want a variable  to always store the same value, like PI (3.14159...).  The final keyword is called a "modifier". | </end> |
| <hitle> | Access Modifiers in Java | <chare> | 2 | <pext> | 1. Private:  We can access the private modifier only within the same class  and not from outside the class.  2. Default:  We can access the default modifier only within the same package  and not from outside the package.  And also, if we do not specify any access modifier  it will automatically consider it as default.  3. Protected:  We can access the protected modifier within the same package  and also from outside the package  with the help of the child class.  If we do not make the child class, we cannot access it  from outside the package.  So inheritance is a must for accessing it  from outside the package.  4. Public:  We can access the public modifier from anywhere.  We can access public modifiers from within the class  as well as from outside the class and also within the package  and outside the package. | </end> |
| <hitle> | Static pros/cons in Java | <chare> | 2 | <pext> | 1. Advantages:  Static members/methods are used as in helper classes  say like Math or in constants classes which helps other objects  to utilize Strings or useful functions for which we do not need  to create object but invoked using Class name.  For example, singleton objects are invoked using a static function.  2. Disadvantages:  Static members are part of class and thus remain in memory  till application terminates and can’t be ever garbage collected.  Using excess of static members sometime predicts that we fail  to design our product and trying to cop of  with static / procedural programming.  It denotes that object oriented design is compromised.  This can result in memory over flow.  Also there are certain disadvantages  if we make any method static in Java for example  we can not override any static method in Java  so it makes testing harder  we can not replace that method with mock.  Since static method maintains global state  they can create subtle bug in concurrent environment  which is hard to detect and fix. | </end> |
| <hitle> | Array and List in concept | <chare> | 2 | <pext> | An array is a container object that holds a fixed number of values  of a single type.  The length of an array is established when the array is created.  After creation, its length is fixed.  Each item in an array is called an element, and each element is accessed by its numerical index.  Also lists are containers for elements having differing data types but arrays are used as containers for elements of the same data type. | </end> |
| <hitle> | One Dimensional Array | <chare> | 2 | <pext> | A One-Dimensional Array in Java programming  is a special type of variable that can store  multiple values of only a single data type  such as int, float, double, char, etc. at a contagious location  in computer memory.  Contagious location means at a fixed gap in computer memory. | </end> |
| <hitle> | One Dimensional Array | <chare> | 2 | <pext> | Multidimensional Arrays can be defined in simple words  as array of arrays.  A two-dimensional array is represented as an array of  one-dimensional arrays of the same type.  Mostly, it is used to represent a table of values with rows and columns. | </end> |