1. *Java class design*

***Java Access Modifiers***

* The access modifiers control the accessibility of your class and its members outside the class and package.
* Access modifiers defined by Java are public, protected, and private. In the absence of an explicit access modifier, a member is defined with the defaultaccess level.
* The public access modifier is the least restrictive access modifier.
* Classes and interfaces defined using the public access modifier are accessible to related and unrelated classes outside the package in which they’re defined.
* The members of a class defined using the protected access modifier are accessible to classes and interfaces defined in the same package and to all derived classes, even if they’re defined in separate packages.
* The members of a class defined without using an explicit access modifier are defined with package accessibility (also called default accessibility).
* The members with package access are accessible only to classes and interfaces defined in the same package.
* A class defined using default access can’t be accessed outside its package.
* The private members of a class are only accessible to itself.
* The private access modifier is the most restrictive access modifier.
* A top-level class, interface, or enum can only be defined using the public or default access. They can’t be defined using protected or private access.
* Method parameters and local variables can never be defined using an explicit access modifier.
* Only applicable non access modifier is final.
* If accessibility of an existing Java entity or its member is decreased, it can break others’ code.

***Overloaded methods and constructors***

* Overloaded methods are methods with the same name but different method parameter lists.
* A class can overload its **own methods** and **inherited methods** from its base class.
* Overloaded methods accept different lists of arguments.
* The argument lists of overloaded methods can differ in terms of change in the number, type, or position of parameters that they accept.
* Overloaded methods are bound at compile time. Unlike overridden methods they’re not bound at runtime.
* A call to correctly overloaded methods can also fail compilation if the compiler is unable to resolve the call to an overloaded method.
* Overloaded methods might define a different return type or access or non-access modifier, but they can’t be defined with only a change in their return types or access or non-access modifiers.
* Overloaded constructors must be defined using different argument lists.
* Overloaded constructors can’t be defined by just a change in the access modifiers.
* Overloaded constructors can be defined using different access modifiers.
* A constructor can call another overloaded constructor by using the keyword this.
* A constructor can’t invoke another constructor by using its class’s name.
* If present, the call to another constructor must be the first statement in a constructor.

*Method overriding and virtual method invocation*

* Method overriding is an OOP language feature that enables a derived class to define a specific implementation of an existing base class method to extend its own behavior.
* A derived class can override an instance method defined in a base class by defining an instance method with the same method signature.
* Whenever you intend to override methods in a derived class, use the annotation @Override. It will warn you if a method can’t be overridden or if you’re actually overloading a method rather than overriding it.
* Overridden methods can define the same or covariant return types.
* A derived class can’t override a base class method to make it less accessible.
* Overriding methods must define exactly the same method parameters; the use of a subclass or parent class results in overloading methods.
* Static methods can’t be overridden. They’re not polymorphic and they’re bound at compile time.
* In a derived class, a static method with the same signature as that of a static method in its base class hides the base class method.
* A derived class can’t override the base class methods that aren’t accessible to it, such as private methods.
* Constructors cannot be overridden because a base class constructor isn’t inherited by a derived class.
* A method that can be overridden by a derived class is called a virtual method.
* Virtual method invocation is invocation of the correct method–determined using the object type and not its reference.

***Java packages***

* You can use packages to group together a related set of classes and interfaces.
* The package and sub package names are separated using a period.
* Classes and interfaces in the same package can access each other.
* An import statement allows the use of simple names for classes and interfaces defined in other packages.
* You can’t use the import statement to access multiple classes or interfaces with the same names from different packages.
* You can import either a single member or all members (classes and interfaces) of a package using the import statement.
* You can’t import classes from a sub package by using the wildcard character, an asterisk (\*), in the import statement.
* A class from the default package can’t be used in any named package, regardless of whether it’s defined within the same directory or not.
* You can import an individual static member of a class or all its static members by using an import static statement.
* An import statement can’t be placed before a package statement in a class. Any attempt to do so will cause the compilation of the class to fail.
* The members of the default package are accessible only to classes or interfaces defined in the same directory on your system.

1. *Advanced class design*

***Abstract classes***

* An *abstract class* is defined by using the keyword abstract. It defines variables to store the state of an object. It may define abstract and non-abstract methods.
* An abstract class must not necessarily define abstract methods. But if it defines even one abstract method, it must be marked as an abstract class.
* An abstract class can’t be instantiated.
* An abstract method doesn’t have any implementation. It represents a behavior that’s required by all derived classes of an abstract class. Because the base class doesn’t have enough details to implement an abstract method, the derived classes are left to implement it in their own specific manner.
* An abstract class *forces* all its non-abstract-derived classes to implement the incomplete functionality in their own unique manner.
* A base class should be defined as an abstract class so it can implement the available details but still prevent itself from being instantiated.
* An abstract class can be extended by both abstract and concrete classes. If an abstract class is extended by another abstract class, the derived abstract class *might* not implement the abstract methods of its base class.
* If an abstract class is extended by a concrete class, the derived class *must* implement all the abstract methods of its base class, or it won’t compile.
* A derived class must call its superclass’s constructor (implicitly or explicitly), irrespective of whether the superclass or derived class is an abstract class or concrete class.
* An abstract class can’t define abstract static methods. Because static methods belong to a class and not to an object, they aren’t inherited. A method that can’t be inherited can’t be implemented. Hence this combination is invalid.
* Efficient use of an abstract class lies in the identification of an abstract class in your application design so you can define common code for your objects and leave the ones that are more specific, by defining them as abstract. You can enforce the derived classes to implement these abstract methods.

***Non-access modifier—static***

* Static members (fields and methods) are common to all instances of a class, and aren’t unique to any instance of a class.
* Static members exist independently of any instances of a class, and may be accessed even when no instances of the class have been created.
* Static members are also known as *class fields* or *class methods* because they are said to belong to their class, and not to any instance of that class.
* A static variable and method can be accessed using the name of an object reference variable or the name of a class.
* A static method and variable can’t access non-static variables and methods of a class. But the reverse works: non-static variables and methods can access static variables and methods.
* Static classes and interfaces are a type of nested classes and interfaces.
* You can’t prefix the definition of a top-level class or an interface with the keyword static. A top-level class or interface is one that isn’t defined within another class or interface.

***Non-access modifier—final***

* You can’t reinitialize a final variable defined in any scope—class, instance, local, or method parameter.
* An instance final variable can be initialized either with its declaration in the initializer block or in the class’s constructor.
* A static final variable can be initialized either with its declaration or in the class’s static initializer block.
* You can’t initialize a final instance variable in an instance method because it can’t be guaranteed to execute only once. Such a method won’t compile.
* You can’t initialize a final static variable in a static method because it can’t be guaranteed to execute only once. Such a method won’t compile.
* If you don’t initialize a final local variable in a method, the compiler won’t complain, as long as you don’t use it.
* If you try to access the value of a final local variable before assigning a value to it, the code won’t compile.
* The Java compiler considers initialization of a final variable complete *only* if the initialization code will execute in *all* conditions. If the Java compiler can’t be sure of execution of code that assigns a value to your final variable, it will complain (code won’t compile) that you haven’t initialized a final variable. If an if construct uses constant values, the Java compiler can predetermine whether the then or else blocks will execute. In this case, it can predetermine whether these blocks of code will execute to initialize a final variable.
* A final instance variable defined in a base class can’t be initialized in the derived class. If you try to do so, your code won’t compile.
* Final methods defined in a base class can’t be overridden by its derived classes.
* Final methods are used to prevent a derived class from overriding the implementation of a base class’s method.
* Private final methods in a base class aren’t inherited by derived classes. A method defined using the same method signature in a derived class isn’t an overridden method, but a new method.
* A final class can’t be extended by any other class.
* A class is defined as final so that it can’t be extended by any other class. This prevents objects of derived classes from being passed on to reference variables of their base classes.
* An interface can’t be defined as final because an interface is abstract, by default. A Java entity can’t be defined both as final and abstract.

***Enumerated types***

* Enumerated types are also called *enums*.
* An enum enables you to create a *type*, which has a *fixed* set of *constants*.
* An enum can never be instantiated using the keyword new.
* Unlike a class, which is defined using the keyword class, an enumerated type is defined using the keyword enum, and can define multiple variables and methods.
* If you define a variable of an enum type, it can be assigned constant values only from that enum.
* All enums extend the abstract class java.lang.Enum, defined in the Java API.
* Because a class can extend from only one base class, an attempt to make your enum extend any other class will fail its compilation.
* The enum constants are implicit static members.
* An enum can implement any interface, but its constants should implement the relevant interface methods.
* An enum can define an abstract method. Just ensure that you override it for all your enum constants.
* You can add instance variables, class variables, instance methods, and class methods to your enums.
* An enum can’t use instance variables in the overridden methods for a particular enum constant.
* You can override non final methods from class java.lang.Enum, for individual (or all) enum constants.
* Your enums can also define constructors, which can be called from within the enum.
* You can define multiple constructors in your enums.
* Enum constants can define new methods, but these methods can’t be called on the enum constant.
* You can define an enum as a top-level enum or within another class or interface.
* You can’t define an enum local to a method.
* An enum can define a main method.

***Static nested classes***

* This class isn’t associated with any object of its outer class. Nested within its outer class, it’s accessed like any other static member of a class—by using the class name of the outer class.
* A static nested class is accessible outside the class in which it’s defined by using names of both the outer class and inner class.
* You can define both static and non-static members in a static nested class.
* A static nested class can define constructors.
* To access the static members of a static nested class, you need not create an object of this class. You need an object to access the instance members of this class.
* The accessibility of the nested static class depends on its access modifier. For example, a private static nested class can’t be accessed outside its class.
* A static nested class can access only the static members of its outer class. Similarly, the outer class can access only the static members of its nested inner class.
* An attempt to access instance members on either side will fail compilation unless it’s accessed through an instance of the outer or static nested class.
* All access levels can be used with this class—public, protected, *default*, and private.

***Inner classes***

* An inner class is an *instance member* of its outer class.
* An object of an *inner class* shares a special bond with its *outer class* and can’t exist without its instance.
* An inner class can be defined using any of the four access levels—public, protected, *default*, and private.
* Members of an inner class can refer to all variables and methods of an outer class.
* An inner class can define constructors.
* An inner class can define variables and methods with any access.
* An inner class can’t define static methods and non-final static variables.
* You can create an object of an inner class within an outer class or outside an outer class.
* Outside the outer class an inner class is instantiated using

**Outer.Inner** inner = **new** Outer().**new** Inner();

***Anonymous inner classes***

* An anonymous inner class is created when you combine object instance creation with inheriting a class or implementing an interface.
* An anonymous inner class might override none, few, or all methods of the inherited class.
* An anonymous inner class must implement all methods of the implemented interface.
* An instance of an
* Anonymous class can be assigned to any type of variable (static variable, instance variable, or local variable) or method parameter, or be returned from a method.
* The following line creates an anonymous inner class that extends Object and assigns it to a reference variable of type Object:

Object obj = new Object(){};

* The following line calls a method, say aMethod(), passing to it an instance of an anonymous class that implements Runnable:

aMethod(new Runnable() {

public void run() {}

});

* When an anonymous inner class is defined within a method, it can access only the final variables of the method in which it’s defined. This is to prevent reassignment of the variable values by the inner class.
* Though you can define variables and methods in an anonymous inner class, they can’t be accessed using the reference variable of the base class or interface, which is used to refer to the anonymous class instance.

***Method local inner classes***

* Method local inner classes are defined within a static or instance method of a class.
* A class can define multiple method local inner classes, with the same class name, but in separate methods.
* Method local inner classes can’t be defined using any explicit access modifier.
* A method local inner class can define its own constructors, variables, and methods by using any of the four access levels—public, protected, *default*, and private.
* A method local inner class can be created only within the method in which it’s defined. Also, its object creation can’t appear before its declaration.
* A method local inner class can access all variables and methods of its *outer class*, including its private members and the ones that it inherits from its base classes. It can only access the final local variables of the method in which it’s defined.
* A method local inner class can define members with the same name as its outer class. In this case, the members of the outer class can be referred to by using Outer.this.

1. *Object-oriented design principles*

***Interfaces***

* An interface is an example of separating the behavior that an object should support from its implementation. An interface is used to define behavior by defining a group of abstract methods.
* All members (variables and methods) of an interface are implicitly public.
* You declare an interface using the keyword interface. An interface can define only public, final, static variables and public, abstract methods.
* The methods of an interface are implicitly abstract and public.
* The variables of an interface are implicitly public, static, and final.
* You can declare a top-level interface only with public and default access. Valid non-access modifiers that can be applied to an interface are abstract and strictfp.
* An interface that’s defined within another interface can be defined with any access modifier.
* An interface can’t extend a class.
* An interface can extend multiple interfaces. It can’t implement another interface.
* An interface can define inner interfaces and (surprisingly) inner classes too.
* Because all the members of an interface are implicitly public, a derived interface inherits all the methods of its base interface.
* You can compare interface implementation to the signing of a contract. When a concrete class declares an implementation of an interface, it agrees to and must implement all its abstract methods.
* If you don’t implement all the methods defined in the implemented interfaces, a class can’t compile as a concrete class. A concrete class must implement all the methods from the interfaces that it implements. An abstract class might not implement all the methods from the interfaces that it implements.
* A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.
* Because the methods in an interface are implicitly public, if you try to assign a weaker access to the implemented method in a class, it won’t compile.
* A class can inherit methods with the same name from multiple interfaces. There are no compilation issues if these methods have exactly the same method signature or if these methods can coexist in the implemented class as overloaded methods. The class won’t compile if these methods coexist as incorrectly overloaded or overridden methods.

***Class inheritance versus interface inheritance***

* Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class.
* You can add new behavior to an abstract or non-abstract base class, and you may not break all the classes that subclass it.
* You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.
* Interface implementation has one major advantage of allowing a class to implement multiple interfaces, so an object of the class can be assigned to variables of multiple interface types.

***Object composition principles***

* Newcomers to programming often extend a class when they want to use a class in another class. They use inheritance in place of composition.
* You should extend a class (inheritance) when you want the objects of the derived classes to reuse the interface of their base class.
* You should define an object of another class (composition) when you want to use the functionality offered by the class.

***Singleton pattern***

* Singleton is a creational design pattern that ensures that a class is instantiated only once. The class also provides a global point of access to it.
* It is used in scenarios when you might need only one object of a class.
* A class that implements the Singleton pattern must define its constructor as private.
* A Singleton class uses a static private reference variable to refer to its sole instance.
* A Singleton class defines a static method to access its sole instance.
* You can also use enums to implement the Singleton pattern because enum instances can’t be created by any other class.
* To avoid threading issues with the creation of the sole instance of the Singleton class, you might use either of the following to create its sole instance:
* Eager initialization—instantiate the object with its declaration
* Synchronized lazy initialization—create the instance using a synchronized method or code block

***Factory pattern***

* One of the most frequently used design patterns, multiple flavors of this pattern exist: Simple Factory, Factory Method, and Abstract Factory.
* The Simple Factory pattern creates and returns objects of classes that extend a common parent class or implement a common interface. The objects are created without exposing the instantiation logic to the client. The calling class is decoupled from knowing the exact name of the instantiated class.
* The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses.
* The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object). This pattern also defines an interface for creating objects but it lets subclasses decide which class to instantiate.
* The benefits of the Factory pattern are
* Prefers method invocation over direct constructor calls
* Prevents tight coupling between a class implementation and your application
* Promotes creation of cohesive classes
* Promotes programming to an interface
* Promotes flexibility. Object instantiation logic can be changed without affecting the clients that use objects. It also allows the addition of new concrete classes.

***DAO pattern***

* The DAO pattern encapsulates all communication with a persistent store to access and manipulate the stored data.
* The DAO pattern also manages the connection to the data store to retrieve and store the data.
* An application usually defines separate DAO classes for each type of data object that should be persisted.
* The CRUD operations form the basis of the DAO pattern.
* The DAO pattern removes the direct dependency between an application and the data persistence implementation.
* The DAO pattern is frequently used with the Factory pattern.

1. *Generics and collections*

*Creating generic entities*

* You define a generic class, interface, or method by adding one or more type parameters to it.
* A class that uses a generic class uses a parameterized type, replacing the formal parameter with an actual parameter. Also, invalid casts aren’t allowed.
* A generic class can be extended by another generic or non-generic class.
* An extended class must be able to pass type arguments to its generic base class. If it doesn’t, the code won’t compile.
* When a non-generic class extends a generic class, the derived class doesn’t define any type parameters but passes arguments to all type parameters of its generic base class.
* A generic interface is defined by including one or more type parameters in its declaration.
* When a non-generic class implements a generic interface, the type parameters follow the interface name.
* When a generic class implements a generic interface, the type parameters follow both the class and the interface name.
* A generic method defines its own formal type parameters. You can define a generic method in a generic or a non-generic class.
* To define a generic method in a non-generic class or interface, you must define the type parameters with the method in its type parameter section.
* A method’s type parameter list is placed just after its access and non-access modifiers and before its return type. Because a type parameter could be used to define the return type, it should be known before the return type is used.
* You can define a generic method in a generic class or interface, defining its own type parameters.
* You can also define a generic constructor in a generic class.
* You can specify the bounds to restrict the set of types that can be used as type arguments to a generic class, interface, or method. It also enables access to the methods (and variables) defined by the bounds.
* For a bounded type parameter, the bound can be a class, an interface, or an enum, but not an array or a primitive type. All cases use the keyword extends to specify the bound. If the bound is an interface, the implements keyword isn’t used.
* A type parameter can have multiple bounds. The list of bounds consists of one class or multiple interfaces.
* For a type parameter with multiple bounds, the type argument must be a subtype of all bounds.
* The wildcard? Represents an unknown type. You can use it to declare the type of a parameter; a local, instance, or static variable; and a return value of generic types. But you can’t use it as a type argument to invoke a generic method, create a generic class instance, or for a super type.
* You can assign an instance of a subclass, say, String, to a variable of its base class, Object. But you can’t assign ArrayList<String> to a variable of type List<Object>. Inheritance doesn’t apply to type parameters.
* When you use a wildcard to declare your variables or method parameters, you lose the functionality of adding objects to a collection.
* To restrict the types that can be used as arguments in a parameterized type, you can use bounded wildcards.
* In upper-bounded wildcards, the keyword extends is used for both a class and an interface.
* For collections defined using upper-bounded wildcards, you can’t add any objects. You can iterate and read values from such collections.
* You can use final classes in upper-bounded wildcards. Although class X extends String won’t compile, <? extends String> will compile successfully.
* You can restrict the use of type arguments to a type and its super types or base types by using <? super Type>, where Type refers to a class, interface, or enum.
* Type information is erased during the compilation process; this is called type erasure.
* When a generic class is compiled, you don’t get multiple versions of the compiled class files.
* The compiler erases the type information by replacing all type parameters in generic types with Object (for unbounded parameter types) or their bounds (for bounded parameter types).
* The Java compiler might need to create additional methods, referred to as bridge methods, as part of the type erasure process.

*Using type inference*

* If you don’t specify the type of type arguments to instantiate a generic class or invoke a generic method, the Java compiler might be able to infer the argument type by examining the declaration of the generic entity and its invocation. If the type can’t be inferred, you might get a compilation warning, an error, or an exception.
* By throwing an unchecked warning, the compiler states that it can’t ensure type safety. The term *unchecked* refers to operations that might result in violating type safety. This occurs when the compiler doesn’t have enough type information to perform all type checks.
* Starting with Java 7, you can drop the type arguments required to invoke the constructor of a generic class and use a diamond—that is, <>. But an attempt to drop the diamond will result in a compilation warning.
* A Java compiler can’t infer the type parameters by using the diamond in the case of generic methods. It uses the type of the actual arguments passed to the method to infer the type parameters.

*Understanding interoperability of collections using raw types and generic types*

* Raw types exist only for generic types.
* You can assign a parameterized type to its raw type, but the reverse will give a compiler warning.
* When you assign a parameterized type to its raw type, you lose the type information.
* When you mix raw types with generic types, you might get a compiler warning or error or a runtime exception.
* You can assign an object of a subclass to reference a variable of its base class. But this subtyping rule doesn’t work when you assign a collection-of-a-derived-class object to a reference variable of a collection of a base class.
* If you declare a reference variable List<Object> to a list, whatever you assign to the list must be of generic type Object. A subclass of Object isn’t allowed.

*Working with the Collection interface*

* The Collection<E> interface represents a group of objects known as its elements.
* There’s no direct implementation of Collection; no concrete class implements it. It’s extended by more specific interfaces such as Set, List, and Queue.
* This collection is used for maximum generality—to work with methods that can accept objects of, say, Set, List, and Queue.
* All collection classes are generic.
* The Map interface doesn’t extend the core Collection interface.
* The Collection interface extends the Iterable interface, which defines method iterator(), enabling all the concrete implementations to access an Iterator<E> to iterate over all the collection objects.
* The methods of the Collection interface aren’t marked as synchronized.

*Creating and using List, Set, and Deque implementations*

* The List interface models an ordered collection of objects. It returns the objects to you in the order in which you added them. It allows you to store duplicate elements.
* In a List you can control the position where you want to store an element. This is the reason that this interface defines overloaded methods to add, remove, and retrieve elements at a particular position.
* Method listIterator() of List can be used to iterate the complete list or a part of it.
* An ArrayList is a resizable array implementation of the List interface.
* An ArrayList uses the size variable to keep track of the number of elements inserted in it. By default, an element is added to the first available position in the array. But if you add an element to an earlier location, the rest of the list elements are shifted to the right.
* If you remove an element that isn’t the last element in the list, ArrayList shifts the elements to the left.
* An ArrayList maintains a record of its size so that you can’t add elements at arbitrary locations.
* ArrayList’s method remove() sequentially searches the ArrayList to find the target object, using method equals() to compare its elements with the target object.
* If a matching element is found, remove(Object) removes the first occurrence of the match found.
* If you’re adding instances of a user-defined class as elements to an ArrayList, override its method equals() or else its method contains() or remove() might not behave as expected.
* The ArrayList methods clear(), remove(), and removeAll() offer different functionalities. clear() removes all the elements from an ArrayList. Remove (Object) removes the first occurrence of the specified element, and remove(int) removes the element at the specified position. removeAll() removes from an ArrayList all of its elements that are contained in the specified collection.
* A Deque is a double-ended queue, a queue that supports the insertion and deletion of elements at both its ends.
* As a double-ended queue, a D
* eque can work as both a queue and a stack.
* The Deque interface defines multiple methods to add, remove, and query the existence of elements from both its ends.
* Methods addFirst(), addLast(), offerFirst(), and offerLast() add and remove elements from the top and tail.
* Deque also defines methods push(), pop(), and peek() to query elements at its beginning.
* ArrayDeque and LinkedList implement the Deque interface.
* ArrayDeque is a resizable array implementation of the Deque interface.
* Deque’s method peek() only queries elements, it doesn’t remove them.
* Deque’s method remove() just removes an element.
* Deque’s method poll() returns null when Deque is empty and remove() throws a runtime exception.
* All the insertion methods (add(), addFirst(), addLast(), offer(), offer- First(), offerLast(), and push()) throw a NullPointerException if you try to insert a null element into an ArrayDeque.
* You can iterate over the elements of Deque by using an Iterator, returned by methods iterator() and descendingIterator().
* Class LinkedList implements both the List and Deque interfaces. So it’s a double-linked list implementation of the List and Deque interfaces.
* **Unlike ArrayDeque, LinkedList permits addition of null elements.**
* A LinkedList is like an ArrayList (ordered by index) but the elements are double-linked to each other. So besides the methods from List, you get a bunch of other methods to add or remove at the beginning and end of this list.
* So it’s a good choice if you need to implement a queue or a stack. A LinkedList is useful when you need fast insertion or deletion, but iteration might be slower than an ArrayList.
* Because a LinkedList implements List, Queue, and Deque, it implements methods from all these interfaces.
* The Set interface models the mathematical Set abstraction.
* The Set interface doesn’t allow duplicate elements and the elements are returned in no particular order.
* To determine the equality of objects, Set uses their method equals(). For two elements, say e1 and e2, if e1.equals(e2) returns true, Set doesn’t add both these elements.
* Set defines methods to add and remove its elements. It also defines methods to query itself for the occurrence of specific objects.
* Class HashSet implements the Set interface. It doesn’t allow the addition of duplicate elements and makes no guarantee to the order of retrieval of its elements.
* **HashSet is implemented using a HashMap.**
* To store and retrieve its elements, a HashSet uses a hashing method, accessing an object’s hashCode() value to determine the bucket in which it should be stored.
* Method hashCode() doesn’t call method equals().
* Method equals() doesn’t call method hashCode().
* Classes should override their hashCode() methods efficiently to enable collection classes like HashSet to store them in separate buckets.
* A HashSet allows storing of only one null element. All subsequent calls to storing null values are ignored.
* Class HashSet uses hashing algorithms to store, remove, and retrieve its elements. So it offers constant time performance for these operations, assuming that the hash function disperses its elements properly among its buckets.
* A LinkedHashSet offers the benefits of a HashSet combined with a LinkedList. It maintains a double-linked list running through its entries.
* As with a LinkedList, you can retrieve objects from a LinkedHashSet in the order of their insertion.
* Like a HashSet, a LinkedHashSet uses hashing to store and retrieve its elements quickly.
* A LinkedHashSet permits null values.
* LinkedHashSet can be used to create a copy of a Set with the same order as that of the original set.
* LinkedHashSet’s method addAll() accepts a Collection object. So you can add elements of an ArrayList to a LinkedHashSet. The order of insertion of objects from ArrayList to LinkedHashSet is determined by the order of objects returned by ArrayList’s iterator (ArrayList objects can be iterated in the order of their insertion).
* A TreeSet stores all its unique elements in a sorted order. The elements are ordered either on their natural order (achieved by implementing the Comparable interface) or by passing a Comparator while instantiating a TreeSet. If you fail to specify either of these, TreeSet will throw a runtime exception when you try to add an object to it.
* Unlike the other Set implementations like HashSet and LinkedHashSet, which use equals() to compare objects for equality, a TreeSet uses method compareTo() (for the Comparable interface) or compare() (for the Comparator interface) to compare objects for equality and their order.
* If two object instances are equal according to their method equals(), but not according to their method compare() or compareTo(), a Set can exhibit inconsistent behavior.
* Classes Enum and File implement the Comparable interface. The natural order of enum constants is the order in which they’re declared. Classes StringBuffer and StringBuilder don’t implement the Comparable interface.

*Map and its implementations*

* Unlike the other interfaces from the collections framework, like List and Set, the Map interface doesn’t extend the Collection interface.
* A Map defines key-values pairs, where a key can map to a 0 or 1 value.
* Map objects don’t allow the addition of duplicate keys.
* The addition of a null value as a key or value depends on a particular Map implementation. A HashMap and LinkedHashMap allow insertion of null as a key, but TreeMap doesn’t—it throws an exception.
* A HashMap is a hash-based Map that uses the hash value of its key (returned by hashCode()) to store and retrieve keys and their corresponding values. Each key can refer to a 0 or 1 value. The keys of a HashMap aren’t ordered. The Hash- Map methods aren’t synchronized, so they aren’t safe to be used in a multithreaded environment.
* You can create a HashMap by passing its constructor another Map object. Additions of new key-value pairs or deletions of existing key-value pairs in the Map object passed to the constructor aren’t reflected in the newly created HashMap.
* Because a HashMap stores objects as its keys and values, it’s common to see code that stores another collection object (like an ArrayList) as a value in a Map.
* You can call method get() on a HashMap to retrieve the value for a key.
* Methods containsKey() and containsValue() check for the existence of a key or a value in a HashMap, returning a boolean value. Methods get() and containsKey() rely on appropriate overriding of a key’s methods hashCode() and equals().
* Class String and all the wrapper classes override their methods hashCode() and equals(), so they can be correctly used as keys in a HashMap.
* HashMap uses hashing functions to add or retrieve key-value pairs. The key must override both methods equals() and hashCode() so that it can be added to a HashMap and retrieved from it.
* When objects of a class that only overrides method equals() (and not method hashCode()) are used as keys in a HashMap, containsKey() will always return false.
* If you add a key-value pair to a HashMap such that the key already exists in the HashMap, the key’s old value will be replaced with the new value.
* You can add a value with null as the key in a HashMap.
* You can use method remove(key) or clear() to remove one or all key-value pairs of a HashMap.
* Method remove() can return a null value, irrespective of whether the specified key exists in a HashMap. It might return null if matching a key isn’t present in HashMap, or if null is stored as a value for the specified key.
* For a HashMap, methods that query or search a key use the key’s methods hash- Code() and equals().
* Method remove() removes a maximum of one key-value pair from a HashMap. Method clear() clears all the entries of a HashMap. Method remove() accepts a method parameter but clear() doesn’t.
* You can use methods size() and isEmpty() to query a HashMap’s size.
* You can use method putAll() to copy all the mappings from the specified map to a HashMap.
* Method putAll() accepts an argument of type Map. It copies all the mappings from the specified map to the map that calls putAll(). For common keys, the values of map that call putAll() are replaced with the values of the Map object passed to putAll().
* The Map interface defines methods keySet(), values(), and entrySet() to access keys, values, and key-value pairs of a HashMap.
* Method values() returns a Collection object, method keySet() returns a Set object, and method entrySet() returns a Map.Entry object.
* Class Hashtable wasn’t a part of the collections framework initially. It was retrofitted to implement the Map interface in Java 2, making it a member of the Java Collection framework. But it’s considered legacy code. It’s roughly equivalent to a HashMap with some differences. The operations of a HashMap aren’t synchronized, whereas the operations of a Hashtable are synchronized.
* The LinkedHashMap IS-A HashMap with a predictable iteration order. Like a LinkedList, a LinkedHashMap maintains a double-linked list, which runs through all its entries.
* A LinkedHashMap will always iterate over its elements in their order of insertion.
* A TreeMap is sorted according to the natural ordering of its keys or as defined by a Comparator passed to its constructor.
* TreeMap implements the SortedMap interface. Like HashMap and LinkedHashMap, the operations of a TreeMap aren’t synchronized, which makes it unsafe to be used in a multithreaded environment.
* The TreeMap performs all key comparisons by using method compareTo() or compare(). Two keys are considered equal by a TreeMap if the key’s method compareTo() or compare() considers them equal.
* When you create a TreeMap object, you should specify how its keys should be ordered. A key might define its natural ordering by implementing the Comparable interface. If it doesn’t you should pass a Comparator object to specify the key’s sort order.
* The set of values that you retrieve from a TreeMap is sorted on its keys and not on its values.
* You can create a TreeMap without passing it a Comparator object or without using keys that implement a Comparable interface. But adding key-value pairs to such a TreeMap will throw a runtime exception, ClassCastException.
* When you pass a Comparator object to TreeMap constructor, the natural order of its keys is ignored.
* Because a TreeMap uses method compare() or compareTo() to determine the equality of its keys, it can access the value associated with a key, even though its key doesn’t override its method equals() or hashCode().

*Using java.util.Comparator and java.lang.Comparable*

* The Comparable interface is used to define the natural order of the objects of the class that implements it.
* Comparable is a generic interface (using T as type parameter) and defines only one method, compareTo(T object), which compares the object to the object passed to it as a method parameter.
* Method compareTo() returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.
* The Comparator interface is used to define the sort order of a collection of objects, without requiring them to implement this interface.
* The Comparator interface defines methods compare() and equals().
* You can pass Comparator to sort methods like Arrays.sort() and Collections.sort().
* A Comparator object is also passed to collection classes like TreeSet and Tree- Map that require ordered elements.
* The Comparator interface is used to specify the sort order for classes that – Don’t define a natural sort order.

*Sorting and searching arrays and lists*

* Class Arrays in the collections framework defines multiple methods to sort complete or partial arrays of primitive data types and objects.
* When method Arrays.sort() accepts from Index and to Index values to sort a partial array, the element stored at position from Index is sorted, but the element stored at position to Index isn’t.
* A space has a lower ASCII or Unicode value than lowercase or uppercase letters. When arranged in an ascending order, a String value that starts with a space is placed before the String values that don’t start with a space.
* Class Collections defines method sort() to sort objects of List.
* Classes Arrays and Collections define method binarySearch() to search a sorted array or a List for a matching value using the binary search algorithm.
* The array or List must be sorted according to the natural order of its elements or as specified by Comparator. If you pass this method an unsorted list, the results are undefined. If more than one value matches the target key value to be searched, this method can return any of these values.
* Method binarySearch() returns the index of the search key if it’s contained in the list; otherwise it returns (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or list.size() if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

*Using wrapper classes*

* All the wrapper classes are immutable.
* All the wrapper classes implement the Comparable interface. All these classes define their natural order.
* You can create objects of all the wrapper classes in multiple ways:
  + *Assignment*—By assigning a primitive to a wrapper class variable
  + *Constructor*—By using wrapper class constructors
  + *Static methods*—By calling the static method of wrapper classes, like valueOf()
* All wrapper classes (except Character) define a constructor that accepts a String argument representing the primitive value that needs to be wrapped. Watch out for exam questions that include a call to a no-argument constructor of a wrapper class. None of these classes defines a no-argument constructor.
* To get a primitive data-type value corresponding to a string value, you can use the static utility method parseDataType(), where DataType refers to the type of the return value.
* Wrapper classes Character, Byte, Short, Integer, and Long cache objects with values in the range of –128 to 127. These classes define inner static classes that store objects for the primitive values –128 to 127 in an array. If you request an object of any of these classes, from this range, method valueOf() returns a reference to a predefined object; otherwise, it creates a new object and returns its reference.
* Integer literal values are implicitly converted to Integer objects and decimal literal values are implicitly converted to Double objects.
* The objects of different wrapper classes with the same values aren’t equal.
* When arranged in natural sort order, false precedes true.

*Autoboxing and Unboxing*

* Autoboxing is the automatic conversion of a primitive data type to an object of the corresponding wrapper class (You box the primitive value). Unboxing is the reverse process (you unbox the primitive value).
* Wrapper classes are immutable. Adding a primitive value to a wrapper class variable doesn’t modify the value of the object it refers to. The wrapper class variable is assigned a new object.
* Unboxing a wrapper reference variable, which refers to null, will throw a NullPointerException.

1. *String processing*

*Regular expressions*

* You create a custom character class by enclosing a set of characters within square brackets []:
  + [fdn] can be used to find an exact match of f, d, or n.
  + [^fdn] can be used to find a character that doesn’t match either f, d, or n.
  + [a-cA-c] can be used to find an exact match of either a, b, c, A, B, or C.
* You can use these predefined character classes as follows:
  + . dot matches any character (and may or may not match line terminators).
  + \d matches any digit: [0-9].
  + \D matches a non-digit: [^0-9].
  + \s matches a whitespace character: [ (space), \t (tab), \n (new line), \x0B (end of line), \f (form feed), \r (carriage)]
  + \S matches a non-whitespace character: [^\s].
  + \w matches a word character: [a-zA-Z\_0-9].
  + \W matches a non-word character: [^\w].
* To use a regex pattern in Java code that includes a backslash, you must escape the \ by preceding it with another \. The character literal \ has a special meaning; it’s used as an escape character. To use it as a literal, it must be escaped.
* For the exam, you’ll need to know these boundary matchers:
  + \b indicates a word boundary.
  + \B indicates a non-word boundary.
  + ^ indicates the beginning of a line.
  + $ indicates the end of a line.
* The coverage of quantifiers on this exam is limited to the following greedy quantifiers:
  + X? matches X, once or not at all.
  + X\* matches X, zero or more times.
  + X+ matches X, one or more times.
  + X{min,max} matches X, within the specified range.

***Search, parse, and build strings***

* Methods indexOf() and lastIndexOf() differ in the manner that they search a target string—indexOf() searches in increasing position numbers and last- IndexOf() searches backward. Due to this difference, indexOf('a', -100) will search the complete string, but lastindexOf('a', -100) won’t. In a similar manner, because lastIndexOf() searches backwards, lastIndexOf('a', 100) will search the string, but indexOf('a', 0) or indexOf('a', -100) won’t.
* Methods indexOf() and lastIndexOf() don’t throw a compilation error or runtime exception if the search position is negative or greater than the length of this string. If no match is found, they return –1.
* Method contains() searches for an exact match in this string. Because contains() accepts a method parameter of interface CharSequence, you can pass to it both a String and a StringBuilder object.
* Methods subSequence (uppercase *S*) and substring (no uppercase letter) accept int parameters and return a substring of the target string.
* Method substring() defines overloaded versions, which accept one or two int method parameters to specify the start and end positions.
* Method subSequence() defines only one variant, the one that accepts two int method parameters for the start and end positions.
* Methods subSequence() and substring() don’t include the character at the end position in the result String. Also, unlike methods indexOf() and lastIndexOf(), they throw the runtime exception StringIndexOutOfBounds- Exception for invalid start and end positions.
* The name of methods subSequence() and substring() can be used to determine their return type. subSequence() returns CharSequence and substring() returns String.
* Methods split(String regex) and split(String regex, int limit) in class String search for a matching regex pattern and split a String into an array of string values.
* The String array returned by split() doesn’t contain the values that it matches to split the target string.
* You can limit the maximum number of tokens that you want to retrieve by using split(String regex, int limit).
* replace(char oldChar, char newChar) returns a new String resulting from finding and replacing all occurrences of the old character with the new character.
* replace(CharSequence oldVal, CharSequence newVal) returns a new String resulting from finding and replacing each substring of the string that matches the old target sequence with the specified new replacement sequence.
* replaceAll(String regex, String replacement) replaces each substring of the string that matches the given regular expression with the given replacement.
* replaceFirst(String regex, String replacement) replaces the first substring of the string that matches the given regular expression with the given replacement.
* Unlike replace(), replaceAll() doesn’t accept method parameters of type CharSequence. Watch out for the passing of objects of class StringBuilder to replaceAll().
* The combination of the replace, replaceAll, and replaceFirst overloaded methods can be confusing on the exam. Take note of the method parameters that can be passed to each of these methods.
* Scanner can be used to parse and tokenize strings.
  + If no delimiter is specified, a pattern that matches whitespace is used by default for a Scanner object.
  + You can specify a custom delimiter by calling its method useDelimiter() with a regex.
  + Method next() returns an object of type String.
  + Scanner also defines multiple nextXXX methods, where XXX refers to a primitive data type. These methods return the value as the corresponding primitive type.
  + Methods hasNext() and hasNextXxx() only return true or false but don’t advance. Only methods next() and nextXxx() advance in the input.
  + Method findInLine() matches the specified pattern with no regard to delimiters in the input.

***Formatting strings***

* Class java.util.Formatter is an interpreter for printf-style format strings.
* Formatter’s format() is used to format data.
* The format specifier takes the following form: %[argument\_index$][flags][width][.precision]conversion
* A format specification must start with a % sign and end with a conversion character:
  + b for boolean
  + c for char
  + d for int, byte, short, and long
  + f for float and double
  + s for String
* If the number of arguments exceeds the required count, the extra variables are quietly ignored by the compiler and JVM. But if the number of required arguments falls short, the JVM throws a runtime exception.
* The - indicates to left-justify this argument; you must specify width as well. Number flags (only applicable for numbers, conversion chars d and f) are as follows:
  + The + indicates to include a sign (+ or -) with this argument.
  + 0 indicates to pad this argument with zeros. Must specify width as well.
  + , indicates to use locale-specific grouping separators (for example, the comma in 123,456).
  + ( is used to enclose negative numbers in parentheses.
* The flags +, 0, (, and , can be specified only with the numeric specifiers %d and %f. If you try to use them with any other format specifier (%b, %s, or %c), you’ll get a runtime exception.
* Format specifier %b
  + You can pass any type of primitive variable or object reference to specifier %b.
  + If the target argument arg is null, then %b outputs the result as false. If arg is boolean or Boolean, the result is the String returned by String.valueOf(). Otherwise, the result is true.
* Format specifier %c
  + %c outputs the result as a Unicode character.
  + You can pass only literals and variables that can be converted to a Unicode character (char, byte, short, int, Character, Byte, Short, and Integer) to the %c specifier. Passing variables of type boolean, long, float, Boolean, Long, Float, or any other class will throw IllegalFormatConversionException.
* Format specifier %f
  + You can format decimal numbers (float, Float, double, and Double) by using the format specifier %f.
  + By default, %f prints six digits after the decimal. It also rounds off the last digit.
* Format specifier %d
  + You can format integers (byte, short, int, long, Byte, Short, Integer, Long) by using the format specifier %d.
  + If you pass literal values or variables of type float, double, Float, or Double to the format specifier %d, the code will throw a runtime exception.
* Format specifier %s
  + %s outputs the value for a primitive variable. For reference variables, it calls toString() on objects that are not null and outputs null for null values.
  + You can pass any type of primitive variable or object reference to specifier %s.
    1. *Exceptions and assertions*
* A method can throw a more specific exception subclass than the one mentioned in its throws clause, but not a more generic one (superclass).
* A method that declares a checked exception to be thrown might not throw it.
* With Java 7, you can re-throw exceptions with more inclusive type checking.

*Using the throw statement and the throws clause*

* The throws clause is part of a method declaration and lists exceptions that can be thrown by a method.
* The throws clause is used with a method declaration to specify that the method won’t handle the mentioned exception (or subclasses) and might throw it to the calling method. The calling method should handle this thrown exception appropriately or declare it to be re-thrown.
* The throw statement is used to throw an exception from a method, constructor, or an initialization block. When an exceptional condition occurs in a method, that method can handle it (by using a try statement), or throw the exception to the calling method by using the throw statement.
* A method indicates that it throws a checked exception by including its name in the throws clause, in its method declaration.
* When you use a method that throws a checked exception, you must either enclose the code within a try block or declare it to be re-thrown in the calling method’s declaration. This is also known as the handle-or-declare rule.
* A method can throw a runtime exception or error irrespective of whether its name is included in the throws clause.
* A method can throw a subclass of the exception mentioned in its throws clause but not a superclass.
* A method can handle the exception and still declare it to be thrown.
* A method can declare to throw any type of exception, checked or unchecked, even if it doesn’t. But a try block can’t define a catch block for a checked exception (other than Exception) if the try block doesn’t throw that checked exception or use a method that declares to throw that checked exception.

*Custom exceptions*

* You can create a custom exception by extending class java.lang.Exception or any of its subclasses.
* You can subclass java.lang.Exception or its subclasses (which don’t subclass RuntimeException) to create custom checked exceptions.
* You can subclass java.lang.RuntimeException or its subclasses to create custom runtime exceptions.
* You can add variables and methods to a custom exception class, like a regular class.
* The name of an exception can convey a lot of information to other developers or users, which is one of the main reasons for defining a custom exception. A custom exception can also be used to communicate a more descriptive message.
* Custom exceptions help you restrict the escalation of implementation-specific exceptions to higher layers. For example, SQLException thrown by data access code can be wrapped within a custom exception and re-thrown.
* You can throw and catch custom exceptions like the other exception classes.

*Overriding methods that throw exceptions*

* With overriding and overridden methods, it’s all about which checked exceptions an overridden method and an overriding method declare, not about the checked exceptions both actually throw.
* If a method in the base class doesn’t declare to throw any checked exception, the overriding method in the derived class can’t throw any checked exception.
* If a method in a base class declares to throw a checked exception, the overriding method in the derived class can choose not to declare to throw any checked exception.
* If a method in a base class declares to throw a checked exception, the overriding method in the derived class can declare to throw the same exception or a subclass of the exception thrown by the method in the base class. An overriding method in the derived class can’t override a method in the base class, if it declares to throw a more generic checked exception.
* Method overriding rules apply only to checked exceptions. They don’t apply to runtime (unchecked) exceptions or errors.

*try statement with multi-catch and finally clauses*

* A multi-catch handler can be used to handle more than one unrelated exception.
* To *catch* multiple exceptions in a single handler, separate the exceptions in a list by using a vertical bar (|).
* A finally block can follow a multi-catch block, like a regular catch block.
* The exceptions that you catch in a multi-catch block can’t share an inheritance relationship. If you try to do so, your code won’t compile.
* You can combine multi-catch and single-catch blocks.
* The same rules apply when combining multi-catch and single-catch blocks—that is, more specific exceptions at the top and more general ones at the bottom.
* You must define a single exception variable in the multi-catch block.
* In a multi-catch block, the variable that accepts the exception object is implicitly final.
* In a multi-catch block, the type of variable that accepts the exception object is the most specific, common super-type of all featured exception classes. Most of the time it’s likely to be Exception, but it could be more specific (for example, IOException for classes FileNotFoundException and EOFException). If the exception classes implement a common interface, then the variable is of an intersection type, with the exception class and interface as its bounds.
* Multi-catch blocks save you from duplicating code, if you need to execute the same code for handling multiple exceptions.

*Auto-close resources with try-with-resources statement*

* You can use a try-with-resources statement to define resources with a try statement that will be automatically closed after the try block completes its execution.
* The try-with-resources is a type of try statement that can declare one or more resources.
* A resource is an object such as file handlers, databases, or network connections, which *should* be closed after it’s no longer required.
* A resource must implement the java.lang.AutoCloseable interface or any of its sub interfaces (directly or indirectly) to be eligible to be declared by a try with- resources statement.
* The java.lang.AutoCloseable interface defines method close().
* If declared within the try-with-resources statement, a resource is automatically closed by calling its close() method at the end of the try block.
* If method close() throws any exception, it should be taken care of by the method that defines the try block; the method must either catch it or declare it to be thrown.
* A try-with-resources block might not be followed by a catch or a finally block. This is unlike a regular try block, which must be followed by either a catch or a finally block.
* The resource declared by try-with-resources is closed immediately after the completion of the try block. Its scope is limited to the try block, and if you try to access it outside the try block, your code won’t compile.
* The variables used to refer to resources are implicitly final variables. You must *declare* and *initialize* resources in the try-with-resources statement.
* It’s acceptable to the Java compiler to initialize the resources in a try-with resources statement to null, only as long as they aren’t being assigned a value in the try block.
* You can initialize multiple resources in a try-with-resources statement, separated by a semicolon (;). The semicolon after the last resource is optional.
* Multiple classes like FileInputStream and FileOutputStream from file I/O implement the java.io.Closeable interface, which extends the java.lang.AutoCloseable interface.

*Assertions*

* An *assertion* offers a way of asserting what should always be true.
* An assertion is implemented by using an assert statement that enables you to test your assumptions about the values assigned to variables and the flow of control in your code.
* An assert statement uses a boolean expression, which you believe to be true. If this boolean expression evaluates to false, an AssertionError is thrown (if assertions are enabled).
* Assertions are used for testing and debugging your code. They’re off by default.
* Assertions are disabled by default so they don’t become a performance liability in deployed applications.
* An assertion is defined using an assert statement that can take two forms. The simpler form uses only a boolean expression: assert <boolean expression>;.
* The longer form of an assert statement includes an expression with the boolean expression: assert <boolean expression>:<expression>;. The second expression used here must return a value (of any type).
* In the longer form of an assert statement, when the boolean expression evaluates to false, the JRE creates an object of AssertionError by passing the value of the second expression to AssertionError’s constructor.
* In the longer form of an assert statement, if you use a method with no return value (void) for the second expression, your code won’t compile.
* In the longer form of an assert statement, you can create an object for the second expression. Note that a constructor creates and returns an object, and so it satisfies the requirement that the second expression must return a value.
* You can test multiple types of invariants in your code by using assertions: internal invariants, control-flow invariants, and class invariants.
* An assertion is used to verify that code that shouldn’t execute, never executes.
* StringTokenizer class not overridden toString() method, replace() doesn’t accept a regex pattern replaceAll() will accept. replace() and replaceAll() will replace all the matches in string.