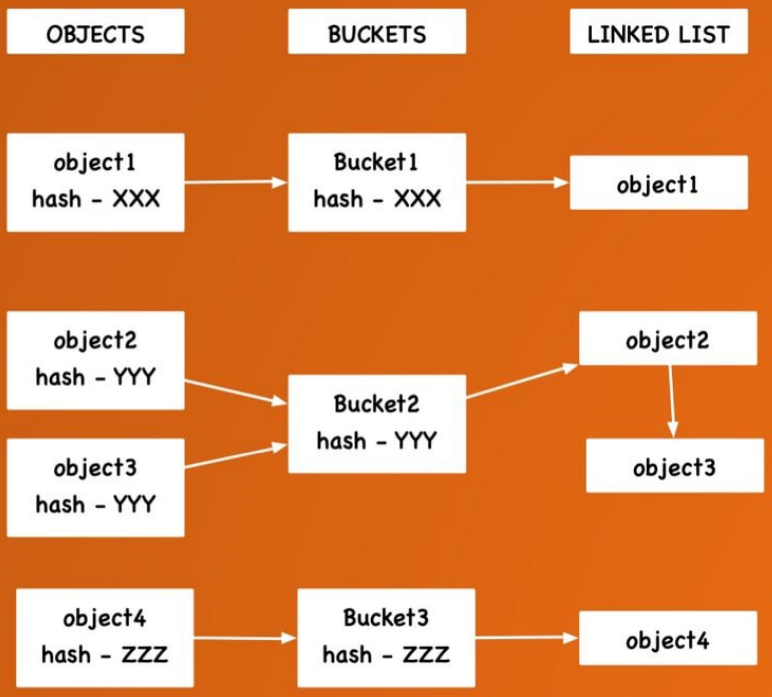
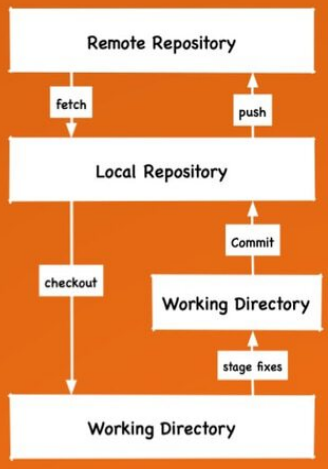
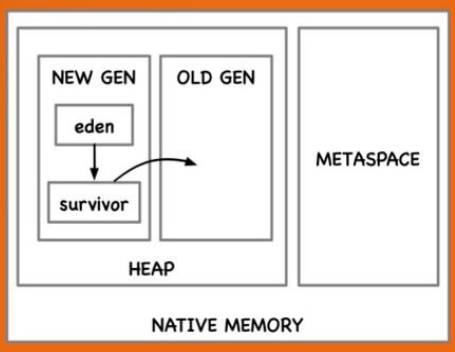
PREPARE FOR YOUR JAVA INTERVIEW IN A DAY!!!

JAVA INTERVIEW COURSE



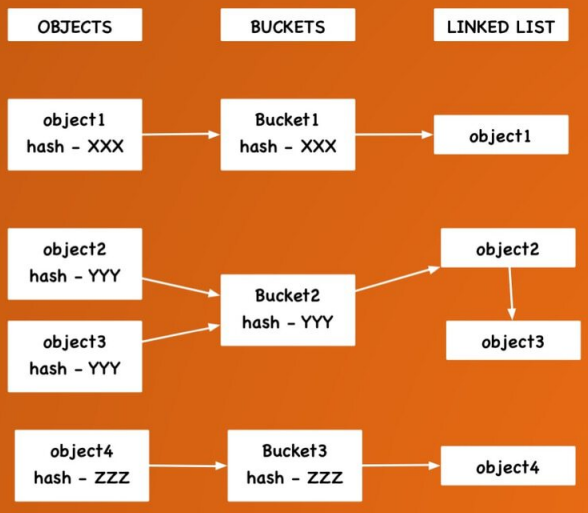


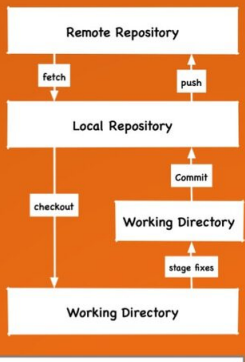


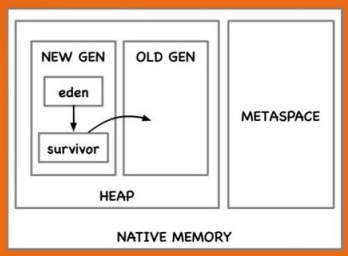
JOLLYM

PREPARE FOR YOUR JAVA INTERVIEW IN A DAY!!!

JAVA INTERVIEW COURSE







JOLLYM

JAVA INTERVIEW COURSE

Jolly M

ISBN-13: 9781234567890
ISBN-10: 1477123456

Cover design by: Art Painter
Library of Congress Control Number: 2018675309
Printed in the United States of America

CONTENTS

String Pool Memory Management
Immutability - Security Issue
Circumvent String Immutability
StringBuilder vs StringBuffer
Unicode
INNER CLASSES
Inner Classes
Static Member Nested Class
Local Inner Class
Non-Static Nested Class
Anonymous Inner Class
FUNCTIONAL PROGRAMMING
Functional Interface
Lambda Expression
Pure Functions
Fluent Interface
GENERICS
Generics
Type Wildcards
Generic Method
Java Generics vs Java Array
Java Array
Type Erasure
Co-variance
Contra-variance
Co-variance vs Contra-variance
COLLECTIONS
Collections
Collection Fundamentals
Collection Interfaces

INTRODUCTION

Java Interview Course book is written based on the experience of many professional Software Architects who have conducted hundreds of technical interviews for their organization.

Java Interview Course covers most of the Java concepts that applicants are expected to have hands-on experience and must have thorough conceptual understating. This book also includes lots of snippets of code and figures to explain the concepts.

A clear understanding of the principles of software design and language is important as the same questions can be discussed during an interview in several different ways.

Interviewers are interested not only in hearing the correct response but also in your opinion and thoughts on the issue. Your approach towards interpreting the questions and articulating your thoughts plays a crucial role in your success.

To build the courage to face such an interview, you need to spend a decent deal of time coding, reading technical topics, and talking to peers about it.

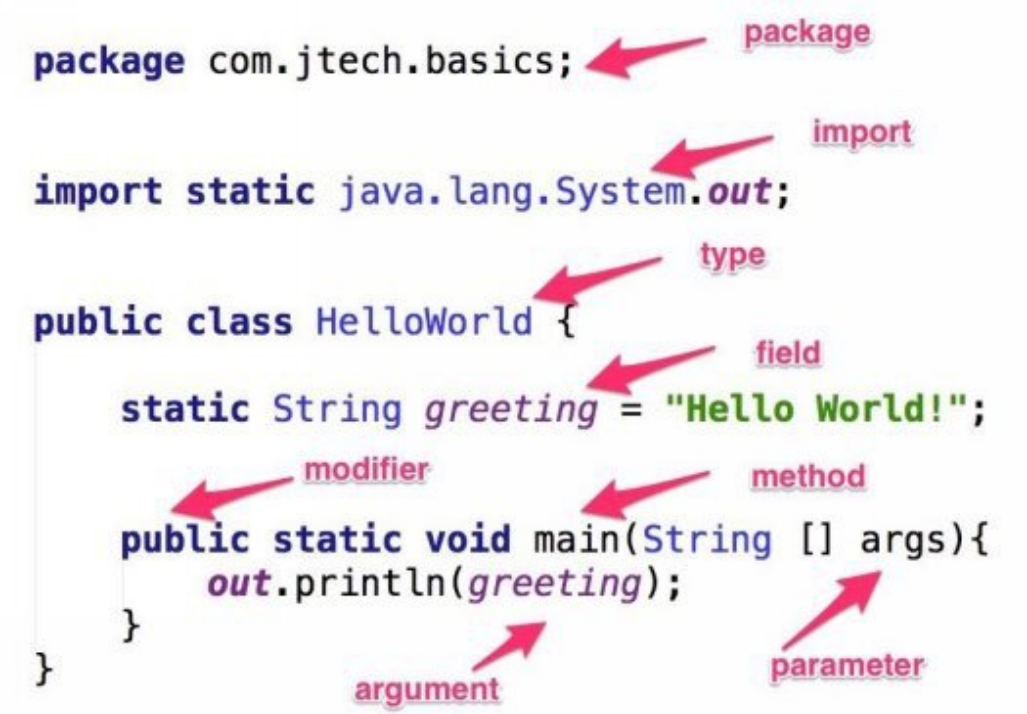
I hope this book helps you plan not just for your next interview but also for your daily task of developing software.

All the best!!!

JAVA FUNDAMENTALS

JAVA PROGRAM ANATOMY

The code snippet below depicts the anatomy of a simple Java program.



- Package - Represents logical grouping of similar types into namespaces. It prevents naming collisions and access protection.
- Import - Imports the package, so that classes can be used in the code by their unqualified names.
- Class - Represents a type template having properties and methods.
- Field - Represents a member used for holding values.
- Method - Represents an operation/behavior of the class.

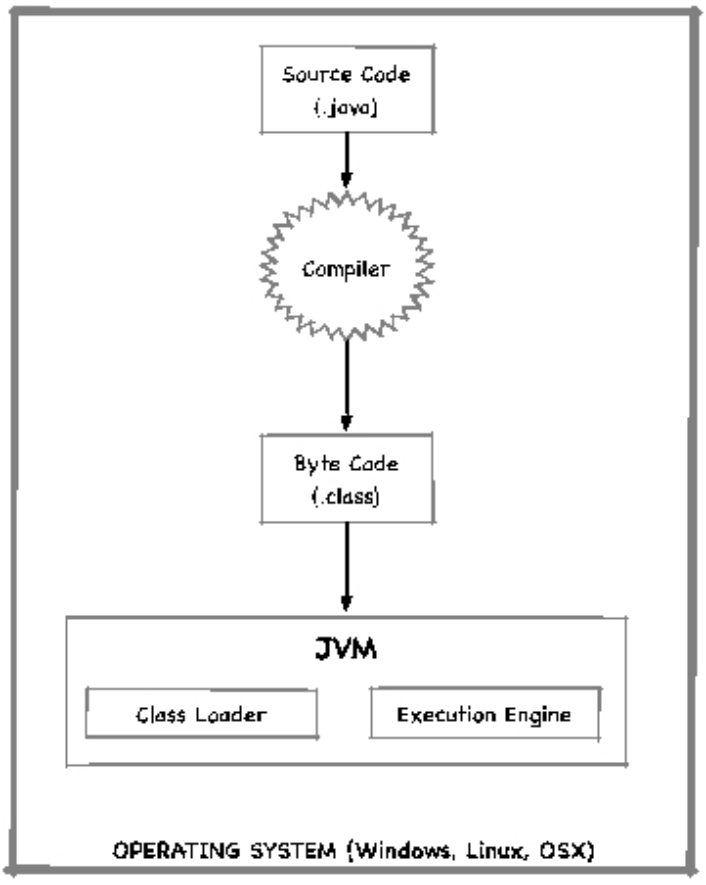
- Modifier - Specifies the access control level of a class and or its members.
- Parameter - Specifies the variable declared in the method definition.
- Argument - Specifies the data passed to the method parameters.

Questions

- What is a package?
- Why do you need to import packages?
- Why do you need to specify access modifiers?
- What is a static import?
    - *static import enables access to static members of a class without the need to qualify it by the class name.*
- What is the difference between argument and parameter?

COMPILING AND EXECUTING JAVA CODE IN JVM

Java program compilation and execution steps



1. Java compiler compiles the Java source code (.java file) into a binary format known as bytecode (.class file). Java bytecode is a platform-independent instruction set, which contains instructions (opcode) and parameter information.

2. Bytecode is translated by the operating system specific Java Virtual Machine (JVM) into the platform-specific machine code.
3. The Class loader in JVM loads the binary representation of the classes into memory.
4. The Execution engine in JVM executes the byte code and generates operating system specific machine instructions. These machine code instructions are executed directly by the central processing unit (CPU).

Questions

- Explain the process of Java code compilation and execution?
- What is bytecode?
- What is the difference between bytecode and source code?
- What is the machine code?
- What is the difference between bytecode and machine code?
- What is JVM? Is it different or the same for different operating systems?
- What are the major components of JVM?
- What is the role of the class loader in JVM?
- What is the role of the execution engine in JVM?
- What are machine instructions?

DATA TYPES

Primitive Types

- Primitive data types are byte , boolean , char , short , int , float , long and double .
- Primitive types always have value; if not assigned, it will have a default value.
- A long value is suffixed with L (or l) to differentiate it from int .
- A float value is suffixed with F (or f) to differentiate it from double . Similarly double is suffixed with D (or d)
- A char is unsigned and represent Unicode values.
- When a primitive type is assigned to another variable, a copy is created.

Reference Types

- All non-primitive types are reference types.
- Reference types are also usually known as objects . Though reference types refer to an object in memory.
- An unassigned object of reference type will have null as the default value.
- Objects have variables and methods, which define their state and behavior.
- When a reference is assigned to another reference, both points to the same object.

Questions

- What are primitive data types?

- If a variable of primitive data type is not assigned, what does it contain?
- Why do we suffix L with long, F with Float and D with a double?
- What happens when you assign a variable of a primitive data type to another variable of the same type?
- What are the reference data types?
- What happens when you assign a variable of the reference data type to another variable of the same reference type?
- What are the differences between the primitive data types and the reference data types?
- What is the purpose of variables and methods in a reference type?
- If a variable of reference data type is not assigned, what does it contain?

NAMING CONVENTION

Camel Case vs Pascal Case

Camel Case is the practice of writing composite words such that the first letter in each word is capitalized, like BorderLength; it is also known as Pascal Case or upper Camel Case . But in the programming world, Camel case generally starts with the lower case letter, like borderLength; it is also known as lower Camel Case . For this discussion, let's consider the format BorderLength as Pascal Case and the format borderLength as Camel Case .

The Naming convention is a set of rules that govern the naming for the identifiers, which represents interface, class, method, variable, and other entities. Although the choice and implementation of the naming conventions often become a matter of debate.

Standard naming convention improves the code readability, review, and overall understanding.

Interface

- The name should be Pascal Case .
- The name should be an adjective if it defines behavior, otherwise noun.

```
public interface Runnable
```

Class

- The name should be Pascal Case .
- The name should be a noun, as a class represents some real-world object.

```
public class ArrayList
```

Method

- The name should be Camel Case .

```
public boolean isEmpty()
```

Variable

- The name should be Camel Case .

```
private long serialVersion = 1234L ;
```

Constants

- The name should be all uppercase letter. Compound words should be separated by underscores.

```
private int DEFAULT_CAPACITY = 10 ;
```

Enum

- Enum set names should be all uppercase letters.

```
public enum Duration {
    SECOND , MINUTE , HOUR
}
```

Acronyms

- Even though acronyms are generally represented by all Upper Case but in Java, only the first letter of acronyms should be upper case and rest lower case.

```
public void parseXml(){}
```

Questions

- What is the meaning of naming convention?
- Why do we need a naming convention?
- What is the difference between Camel Case and Pascal Case?
- What is the difference between Upper Camel Case and Lower Camel Case?
- Explain naming convention for Interface, Class, Method, Variable, Constant, Enum, and Acronyms?

OBJECT CLASS

Every Java class is inherited, directly or indirectly, from java.lang.Object class, which also means that a variable of Object class can reference the object of any class.

Because of inheritance, following Object class methods are available for overriding with class-specific code.

- hashCode() - returns hash-code value for the object.
- equals() - compares two objects for equality using identity (==) operator.
- clone() - creates copy of object. An Overriding class should inherit the Cloneable interface and implement a clone() method to define the meaning of copy.
- toString() - returns string representation of the object.
- f inalize() - called by Garbage Collector to clean up resources. Object's implementation of finalize() does nothing.

Questions

- What is the base class for all Java classes?
- What are the different methods of Object class available for overriding in the derived class?
- What happens if your class does not override equals method from the Object class?
    - *The equals() method in Object class compares whether object references are the same and not the content. To*

compare content, you need to override the equals() method.

- What is the purpose of the clone () method?
- Why should the overriding class define its implementation of the clone() method?
- What happens if the overriding class does not override the clone() method?
    - *In case if an object contains references to an external object, any change made to the referenced object will be visible in the cloned too.*

ACCESS MODIFIERS

The access modifiers determine the rules about whether other classes can access a variable or invoke a method.

At the class level, you can either use a public modifier or no modifier.

For class members, you can use one of the following access modifiers.

private - External classes cannot access the member.
protected - Only sub-classes can access the member.
public - All classes in the application can access the member.
no modifier - All classes within the package can access this member.

The access modifier in the overriding methods should be the same or less restrictive than the overridden method.

Optional static and final keywords are frequently used along with the access modifiers.

Questions

- What is the purpose of the access modifier?
- Is there any difference between the list of access modifiers available for a class and for its members?
- What is the scope of private, protected, and public access modifiers?
- What happens when no access modifier is specified with the class?

- If sub-class exists in a different package, can it still have visibility to the protected members of the super-class?
- Why should the member access modifier in the derived class be less restrictive than the base?
    - *As per the concepts of Inheritance, you should be able to use sub-class objects with super-class reference. This will not be possible if sub-class member is decelerated with more restrictive access modifier.*
- What should be the criteria to decide an access modifier for your class?
    - *You should use the most restrictive access modifier to ensure security and to prevent any misuse.*

STATIC

static class

Only nested/inner classes can be defined as static and not the outer class.

static variable and method

When a static keyword is used with the variables and the methods, it signifies that these members belong to the class and these members are shared by all the objects of the class. Static members do not have a copy and are stored only at a single location in memory. These members should be accessed using the class name.

A static method does not have access to instance methods or properties, because static members belong to the class and not the class instances.

Questions

- What are the static classes?
- Can any class be declared as a static class?
- What are the static methods?
- What are the static variables?
- Who owns the static class members? How is that different for non-static classes?
- How should you access class members that are static?
- Can a static method have access to an instance member? Why?

FINAL

final Class

A final class cannot be extended, which makes the class secure and efficient.

final Method

A final method cannot be overridden, which prevents any possibility of introducing any unexpected behavior to the class.

final Variable

A final variable reference cannot be changed, but the content of the mutable object , that the final variable is referencing, can be changed.

blank final variable – a variable that is not initialized at the point of declaration.

Notes

- A blank final variable needs to be initialized in the constructor of the class.
- The final variables are like immutable variables, so computations related to final variables can be cached for optimization.

Questions

- Explain the final class?
- What are the benefits of declaring a class final?
- Explain the final method?

- What are the benefits of declaring a method final?
- Explain the final variable?
- What are the benefits of declaring a variable final?
- When you declare a variable final, can you change the content of the object it's referencing?
- When you declare a variable final, can you change it to reference another object?
- What is the blank final variable?
- How does declaring a variable as final helps with optimization?

STATIC INITIALIZATION BLOCK

- A static initialization block is generally used to ensure that all the required class resources (like drivers, connection strings, etc.) are available before any object of the class is used.
- A static block does not have access to the instance members.
- A static block i s called only once for a class.
- A class can define any number of initializing blocks, which gets called in order of their definition in the class.
- You can only throw an unchecked exception from a static block .

In this code example, static initialization block creates connection string once for the class.

```
private static  String connectionString ;
static {
        connectionString = getConnectionSting();
}
```

Questions

- What is a static initialization block?
- What is the primary purpose of a static initialization block? What are the types of things you should do in a static block?
- Can you access instance members from static initialization block? Why?

- Does the static initialization block gets called whenever an instance of the class is created?
- How many static blocks can be defined in a class?
- When multiple static blocks are defined, what is the criterion about their order of execution?
- Can you throw an exception from the static initialization block? What type?

FINALLY

T he primary purpose of a finally block is to ensure that the application is brought back to a consistent state, after the operations performed in the try block. Within the finally block, usually, the resources like streams and database connections can be closed to prevent leaks.

```
InputStream is = null ;
try {
   is = new FileInputStream("input.txt" );
}
finally {
   if (is != null ) {
      is .close();
   }
}
```

finally block execution

The compiler does all in its power to execute the finally block, except in the following conditions:

- If System.exit() is called.
- If the current thread is interrupted.
- If JVM crashes.

Return from finally

You must never return from within the finally block. If there is a return statement present in the finally block, it will immediately return, ignoring any other return present in the function.

Questions

- How do you guarantee that a block of code is always executed?

- What are the things that you should do in a finally block?
- What are the things that you should do in a catch block?
- Does finally block always execute? What are the conditions when finally block does not execute?
- Should you ever return from the finally block? Why?

FINALIZE()

When the Garbage Collector determines that there is no reference to an object exists, it calls f inalize() on that object; just before removing that object from memory.

The finalize() method will not be called if an object does not become eligible for garbage collection, or if JVM stops before garbage collector gets a chance to run.

The finalize() method could be overridden to release the resources like file handles, database connections, etc.; but you must not rely on *finalize* to do so, and release such resources explicitly.

There is no guarantee that finalize() will be called by the JVM , and you should treat finalize only as a backup mechanism for releasing resources. Where possible, use t ry-with-resource construct to automatically release the resources; as soon as their usage is finished.

If an uncaught exception is thrown by the finalize() method, the exception is ignored before terminating the finalization.

Questions

- What is the finalize method in Java?
- When does the finalize method gets called?
- Who calls the finalize method?
- What are the things that can be done in the finalize method?
- Should you explicitly call the finalize method to release resources? Why?

- What are some alternate mechanisms that can be used to release system resources?
- What happens if an unhandled exception is thrown from the finalize method?

WIDENING VS NARROWING CONVERSIONS

Widening Conversions

Widening conversions deals with assigning an object of sub-class (derived class) to an object of super-class (base class). In the example below, Car is derived from Vehicle .

```
Car car = new Car ();
Vehicle vehicle = car ;
```

Narrowing Conversions

Narrowing conversions deals with assigning an object of super-class (base class) to an object of sub-class (derived class). An explicit cast is required for conversion. In the example below, Bike is derived from Vehicle .

```
Vehicle vehicle = new Vehicle ();
Bike bike = ( Bike ) vehicle ;
```

Questions

- What is widening conversion?
- What is narrowing conversion?
- Is there any possibility of a loss of data in narrowing conversion?

GETTERS AND SETTERS

The following code demonstrates the usage of getter and setter .

```
public class Person {

    private String name ;

    public String getName() {
     return StringUtils .capitalize( name );
  }

    public void setName( String name) {
     if (name.isEmpty()){
        System . out .println( "Name string is empty" );
        //throw exception
     }
     this . name = name;
  }
}
```

Benefits of using getter and setter

- Validations can be performed in the setter or can be added later when required.
- Value can have alternative representation, based on internal (storage) or external (caller's) requirement.
- It hides the internal data structure used to store the value.
- Internal fields can be changed, without requiring changing any users of the code.
- It encapsulates the internal complexity while retrieving or calculating the value.

- It provides the ability to specify different access modifiers for getter and setter.
- It provides the ability to add debugging information.
- It can be passed around as Lambda expressions.
- Many libraries like mocking, serialization, etc. expect getters/setters for operating on the objects.

Questions

- Why do you need getters and setters when you can directly expose the class field?
- Explain a few benefits of using getters and setters?

VARARGS VS OBJECT ARRAY

The varargs argument allows zero or more parameters to be passed to the method; whereas, an object array argument cannot be called with zero parameters.

varargs

```
public static int getCumulativeValue( int ... values){
    int sum = 0 ;
    for ( int value : values){
       sum += value ;
   }
    return sum ;
}
```

object array

```
public static int getCumulativeValues( int [] values){
    int sum = 0 ;
    for ( int value : values){
       sum += value ;
   }
    return sum ;
}
```

- The varargs can only be the last argument in the method; whereas, an object array can be defined in any order.
- Both varargs and object array are handled as an array within a method.
- Though varargs are not very popular, it can be used in any place where you have to deal with an indeterminate number of

parameters.

Questions

- What is varargs?
- What is the difference between varargs and object array?
- Can you call a method with zero arguments, that expects a varargs as a parameter?
- Can you overload a method that takes an int array, to take an int varargs?
- What are the different scenarios where you can use varargs?

DEFAULT INTERFACE METHOD

- Default interface methods are directly added to an Interface to extend its capabilities.
- The Default interface method can be added to an Interface that is not even under your control.
- Default interface method does not break any existing implementation of an interface it is added to.
- Implementing class can override the default methods defined in the interface.
- The Default method is also known as the Defender or Virtual extension method.

In this code example default Interface method , getAdditonSymbol() , is added to an existing interface Calculator .

```
public interface Calculator {
    public < T > T   add( T num1, T num2);
    default public String getAdditionSymbol(){
       return "+" ;
   }
}
```

Limitations with the Default method

- If the class inherits multiple interfaces having default methods with the same signature, then the implementing class has to provide the implementation for that default method.
- If any class in the inheritance hierarchy has a method with the same signature, then default methods become irrelevant.

Default method vs Abstract method

Following are a couple of minor differences:

- Abstract methods allow defining constructor.
- Abstract methods can have a state associated.

With Default method - Abstract class vs Interface

With the introduction of default methods , now even the Interfaces can be extended to add more capabilities, without breaking the classes that inherit from the Interface.

Questions

- What are default interface methods?
- What are the benefits of default interface methods?
- Can you add default interface methods to an interface that is not directly under your control?
- Can you override the default interface methods to provide different implementation?
- What happens when a class inherits two interfaces and both define a default method with the same signature?
- How defining a default method in an interface is different from defining the same method in an abstract class?

STATIC INTERFACE METHOD

- Static Interface methods are directly added to an interface to extend its capabilities.
- Static Interface methods are generally used to implement utility functions like validations, sorting, etc.
- Static interface methods are also used when you want to enforce specific behavior in the classes inheriting the Interface.
- Implementing class cannot override the static methods defined in the interface it is inheriting.
- The static Interface method can even be added to an interface that is not under your control.
- Similar to the default Interface method, even the static interface method does not break any existing implementation of the interface it is added to.

In this code example static Interface method, getUtcZonedDateTime(), is added to an existing interface DBWrapper.

```
public interface DBWrapper {
    static ZonedDateTime getUTCZonedDateTime(
            Instant date ){
      ZoneId zoneId =
            TimeZone.getTimeZone( "UTC" ).toZoneId();
        ZonedDateTime zonedDateTime =
              ZonedDateTime.ofInstant(date, zoneId );
       return zonedDateTime ;
   }
}
```

Questions

- What is static interface method?
- Where can you use static interface methods?
- Can you override static interface methods?
- What is the difference between static and default interface methods?
- Can you add a static interface method to an interface which is not directly under your control?
- What happens if a class inherits two interfaces and both define a static interface method with the same signature?

ANNOTATIONS

An annotation associates metadata to different program elements. Annotations may be directed at the compiler or at runtime processing.

Annotations metadata can be used for documentation, generating boilerplate code, performing compiler validation, runtime processing, etc. Annotations do not have any direct effect on the code piece they annotate.

We can apply annotations to a field, a variable, a method, a parameter, a class, an interface, an enum, a package, an annotation itself, etc.

Usage

User-defined annotations are directly placed before the item to be annotated.

```
@Length (max= 10 , min= 5 )
public class ParkingSlot {
// Code goes here
}
```

Built-in annotations

- @Deprecated - signifies that the method is obsoleted.
- @Override - signifies that a superclass method is overridden.
- @SupressWarnings - used to suppress warnings.

Questions

- What are annotations?
- Where can you use annotations?

- Where you can apply annotations?

PREFERENCES

In Java, the Preferences class is used for storing user preferences in hierarchical form. Preferences class abstracts out the process of storage and stores the preferences in an operating system specific way: preferences file on Mac, or the registry on Windows systems. Though the keys in preferences are Strings value can belong to any primitive type.

Applications use Preferences class to store and retrieve user and system preferences and configuration data.

Questions

- What are the Preferences?
- What are the types of information that can be stored with the Preferences?
- While using the Preference class, should you worry about the internal format used by the operating system?

PASS BY VALUE OR PASS BY REFERENCE

In Java - method arguments, whether primitive or object reference, are always passed by value, and access to the objects is allowed only through the reference. While passing an object to a method, it's the copy of the reference that is passed and not the object itself. Any changes done to the object reference changes the object content and not the value of the reference.

Questions

- What is the difference between pass-by-value and pass-by-reference?
- How is the reference type argument passed in Java; by reference or by value?
- If a copy of the reference is passed by value, how can the method get access to the object that reference is pointing to?
- If a copy of the reference is passed by value, can you change the value of reference?

OBJECT ORIENTED PROGRAMMING

POLYMORPHISM

Polymorphism is an ability of a class instance to take different forms based on the instance it's acting upon.

Polymorphism is primarily achieved by sub-classing or by implementing an interface. The derived classes can have their own unique implementation for a certain feature and yet share some of the functionality through inheritance.

The behavior of objects depends specifically on its position in the class hierarchy.

Consider you have a Furniture class that has addLegs() method. A Chair and a Table class extend Furniture class; having their implementation of addLegs(). In the above situation, the runtime determines which implementation of addLegs() method gets called, based on whether you have a Chair or a Table class instance.

```
public abstract class Furniture {
    public abstract void addLegs();
    public void print( String message){
       System . out .println(message);
   }
}

class Chair extends Furniture {
    @Override
    public void addLegs() {
      print( "Chair Legs Added" );
   }
}
```

```
class Table extends Furniture{
    @Override
    public void addLegs() {
      print( "Table Legs Added" );
   }
}

Furniture furniture = new Chair ();
// This prints "Chair Legs Added"
furniture .addLegs();

furniture = new Table ();
// This prints "Table Legs Added"
furniture .addLegs();
```

Benefits of polymorphism

The real power and benefit of polymorphism can be achieved when you can code to an abstract base class or an interface. Based on the context, polymorphism enables the selection of most appropriate class implementation. Not only in production code, but it also paves the way to have an alternate implementation for testing.

Questions

- What is Polymorphism?
- What are the different ways to achieve polymorphism?
- How is inheritance useful to achieve polymorphism?
- What are the benefits of polymorphism?
- How is the polymorphism concept useful for unit testing?

Parametric polymorphism

In Java, Generics provides an implementation of Parametric Polymorphism , which enables the use of the same code implementation with the values of different types, without compromising on compile-time type safety check.

In the example below, we added an upper bound to type parameter T such that it implements an interface that guarantees getWheelsCount() method in the type T.

```
interface Vehicle {
    int getWheelsCount();
}


class Car < T extends Vehicle> {
    private T vehicle ;
    public Car( T vehicle) {
       this . vehicle = vehicle;
   }
    public int getWheelsCount() {
       return vehicle .getWheelsCount();
   }
}
```

It can take a list of Vehicle of type T and returns the count of wheels in the Vehicle , without worrying about what type T actually is.

Questions

- What is Parametric Polymorphism?
- How *Generics* is used to achieve Parametric Polymorphism?
- How are the *Type Wildcards* used to achieve Parametric Polymorphism?
- Can you achieve Parametric Polymorphism without Generics?

Subtype polymorphism

In Subtype polymorphism, also known as inclusion polymorphism , the parameter definition of a function supports any argument of a type or its subtype.

So if the parameters of a function have subtypes, then that function exhibits subtype polymorphism . Java code below illustrates the use of this kind of polymorphism.

```
abstract class Vehicle{
    public abstract int getWheelsCount();
}


class Car extends Vehicle{
    @Override
    public int getWheelsCount() {
       return 4 ;
   }
}


class Bike extends Vehicle{
    @Override
    public int getWheelsCount() {
       return 2 ;
   }
}


public void printWheelsCount(Vehicle vehicle) {
   print(vehicle.getWheelsCount());
}


public void main( String [] args) {
   printWheelsCount( new Car ());
```

```
    printWheelsCount( new Bike ());
}
```

In the code above, the method printWheelsCount() takes a Vehicle as parameter and prints count of wheels in the Vehicle. The main method shows subtype polymorphic calls, taking objects of Car and Bike as arguments. Any place that expects a type as a parameter will also accept subclass of that type as a parameter.

Questions

- What is Subtype Polymorphism?
- What is Inclusion Polymorphism?
- What is the difference between Parametric Polymorphism and SubType Polymorphism?
- Can you achieve SubType polymorphism using Generics?

OVERRIDING

- Method overriding is redefining the base class method to behave in a different manner than its implementation in the base class.
- Method overriding is an example of dynamic or runtime polymorphism.
- In dynamic polymorphism , the runtime takes the decision to call an implementation, as the compiler does not know what to bind at compile time.

Rules for method overriding

- Method arguments and their order must be the same in the overriding method.
- Overriding methods can have the same return type or subtype of the base class method's return type.
- The access modifier of the overridden method cannot be more restrictive than its definition in the base class.
- Constructor, static, and final method cannot be overridden.
- The overridden method cannot throw checked exceptions if its definition in base class doesn't, though the overridden method can still throw an unchecked exception .

Questions

- What is method overriding?
- What is dynamic polymorphism?
- Why can't you override static methods defined in super-class or interface?

- Can you override a final method defined in super-class?
- Can you override a public method in super-class and mark it protected?
- Why can't you override constructor of super-class?
- Can an overridden method throw checked exception; when the method in super-class does not? Why?
- What are the benefits of method overriding?

@Override

The @Override annotation is a way to explicitly declare the intention of overriding a method implementation defined in the base class. Java performs compile-time checking for all such annotated methods. It provides an easy way to mistake-proof against accidentally writing the wrong method signature when you want to override from the base class.

If a derived class defines a method having the same signature as a method in the base class, the method in the derived class hides the one in the base class. By prefixing a subclass's method header with the @Override annotation, you can detect if an inadvertent attempt is made to overload instead of overriding a method.

Questions

- What is the purpose of the @Override annotation?
- What happens if you define a method with the same signature as defined in the super-class and without using the @Override annotation?
- What are the benefits of @Overrode annotation?

OVERLOADING

- The method overloading is about defining more than one method with the same name, but with different parameters.
- Method overloading is an example of static or compile-time polymorphism.
- In static polymorphism , it's while writing the code the decision is made to call a specific implementation.

Rules for method overloading

- The method can be overloaded by defining method with the same name as an existing one but having
    - A different number of arguments.
    - The different datatype of arguments.
    - A different order of arguments.
- The return type of the overloaded method can be different.
- A method with the same name and the same parameter cannot be defined when they differ only by return type.
- Overloading methods are not required to throw the same exception as the methods it’s overloading.

Operator Overloading

Operator overloading is an ability to enhance the definition of the language-dependent operators. For example, you can use + operator to add two integers and also to concat two strings.

Questions

- Explain method overloading?

- What is static polymorphism?
- What is the difference between static and dynamic polymorphism?
- Can you override a method such that all the parameters are the same with the difference only in return type?
- What is operator overloading?
- What are the benefits of method overloading?
- What is the difference between overriding and overloading?

ABSTRACTION

Abstraction helps to move the focus from the internal details of the concrete implementation to the types and its behavior. Abstraction is all about hiding details about data, its internal representation, and its implementation.

The other equally important object-oriented concept is an encapsulation that could be used to abstract the complexities and the internal implementation of a class.

Abstraction also helps to make the software maintainable, secure, and provides an ability to change the implementation without breaking any client.

Questions

- What is abstraction?
- How abstraction is different from encapsulation?
- What are the benefits of abstraction?
- Can you achieve abstraction without encapsulation?

INHERITANCE

Inheritance is an object-oriented design concept that deals with reusing an existing class definition (known as super-class) and defining more special categories of class (known as sub-class) by inheriting that class. It focuses on establishing the IS-A relationship between sub-class and its super-class. Inheritance is also used as a technique to implement polymorphism ; when a derived type implements a method defined in the base type.

Rules for Inheritance

- There can be a multiple levels of inheritance, based on the requirements to create specific categories.
- Only single class inheritance is allowed in Java, as multiple inheritances comes with its share of complexity; check Diamond Problem .
- A class declared final cannot be extended.
- A class method declared final cannot be overridden.
- Constructors and private members of the base class are not inherited.
- The constructor of the base class can be called using super() .
- The base class's overridden method can be called using super keyword, otherwise, you will end up calling an overriding method recursively.

Questions

- Explain inheritance?
- What is the purpose of inheritance?

- What should be the criteria to decide the inheritance relation between two classes?
- How inheritance plays an important role in polymorphism?
- Can you inherit the final class?
- What happens if you don't use the super keyword to call an overridden member?
- Why can't you inherit static members defined in the super-class?
- What are the challenges one can face if multiple inheritances are possible in Java?

COMPOSITION

The composition is an object-oriented design concept that is closely related to inheritance , as it also deals with reusing classes; but it focuses on establishing a HAS- A relationship between classes. So unlike Inheritance , which deals with extending features of a class, composition reuses a class by composing it. The composition is achieved by storing reference of another class as a member.

Inheritance vs Composition

Primary issue with inheritance is that it breaks encapsulation as the derived class becomes tightly coupled to implementation of the base class. The problem becomes complex when a class is not designed keeping future inheritance scope and you have no control over the base class. There is possibility of breaking a derived class because of changes in the base class.

So, inheritance must be used only when there is perfect IS-A relationship between the base and the derived class definitions; and in case of any confusion prefer composition over inheritance .

Questions

- Explain composition?
- What is the difference between inheritance and composition?
- What should be the criteria to decide composition relation between two classes?
- Explain few problems with inheritance that can be avoided by using composition?

- When should you prefer composition over inheritance and vice versa?

FUNDAMENTAL DESIGN CONCEPTS

DEPENDENCY INJECTION VS INVERSION OF CONTROL

The Dependency Injection and the Inversion of Control promotes modular software development by loosely coupling the dependencies. Independent modular objects are also more maintainable and testable.

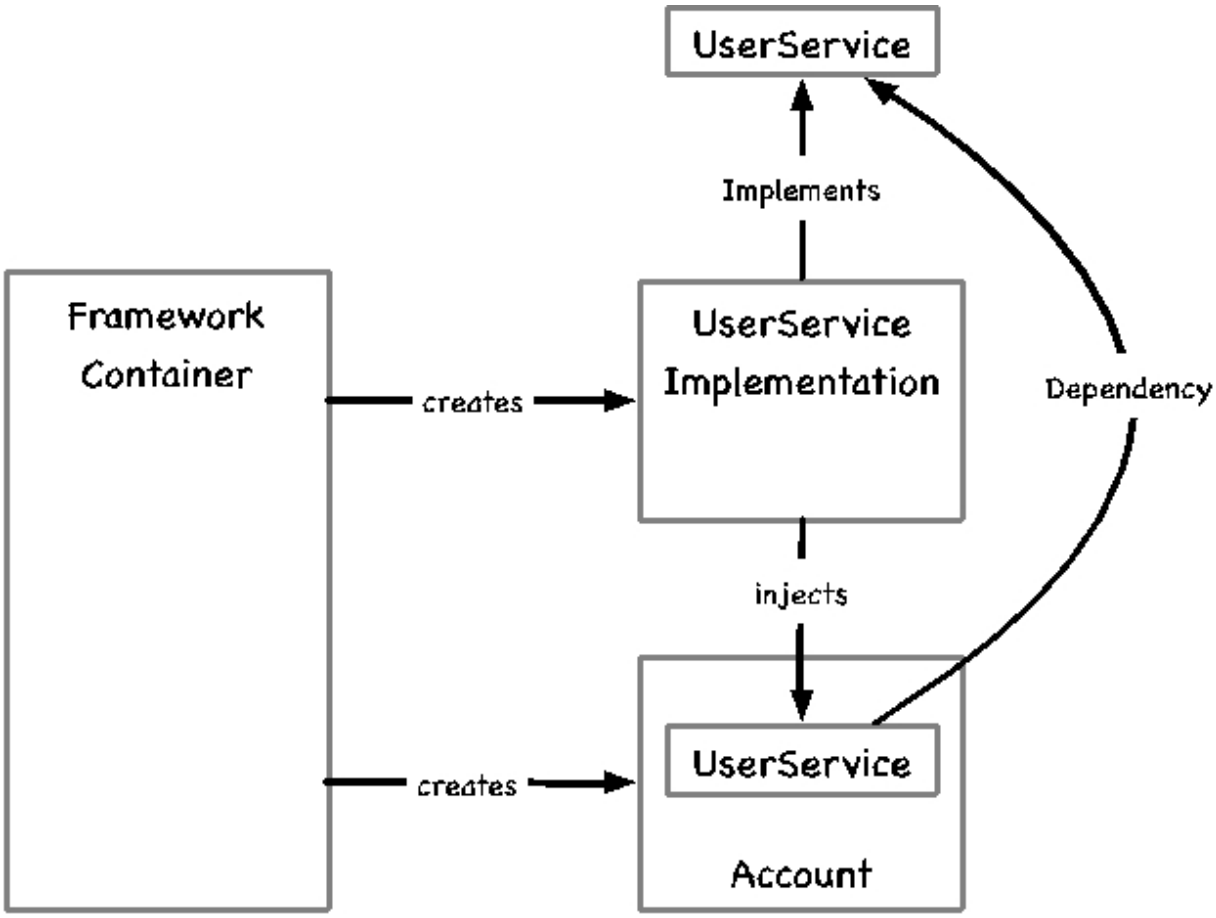
Inversion of Control

Inversion of Control provides a design paradigm where dependencies are not explicitly created by the objects that require these, but such objects are created and provided by the external source.

Dependency Injection

Dependency Injection is a form of IoC that deals with providing object dependencies runtime; through constructors, setters or the service locators. Annotations and Interfaces are used to identify the dependency sources to create and inject dependencies.

- Mode of dependency injection:
    - Through constructor
    - Through setter
    - Through method parameter
- It's the responsibility of the dependency injection framework to inject the dependencies.



The code below demonstrates dependency injection as a constructor parameter.

```
public class Account {

    UserService userService ;
  AccountService accountService ;

    public Account(UserService userService,
            AccountService accountService) {
      this .userService =
            userService;
      this .accountService =
            accountService;
   }
}
```

Not only in production systems, but DI and IoC also provide immense help in unit testing too, by providing an ability to mock dependencies. Spring framework is an example of a DI container.

Note

It is important to ensure that dependency objects are initialized before they are requested.

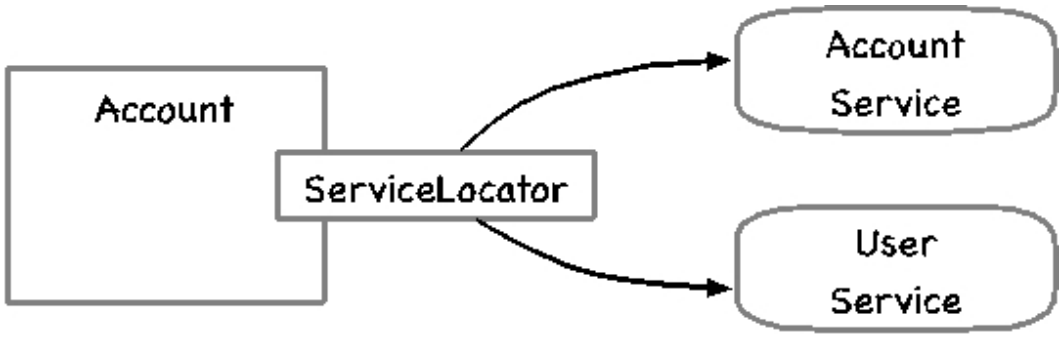
Questions

- What is Inversion of Control?
- What is Dependency Injection?
- What is the difference between the Inversion of Control and Dependency Injection?
- What are the different ways to implement Dependency Injection?
- What the different ways to identify dependency sources?
- Who has the responsibility to inject dependent objects?
- How Dependency injection is useful for unit testing?
- How Dependency Injection can be used for modular software development?

SERVICE LOCATOR

The service locator is an object that encapsulates the logic to resolve the service requested. The service locator also provides an interface to register services with it, which allows you to replace the concrete implementation without modifying the objects that depend on these services.

In the figure below, Account class uses ServiceLocator to resolve the Account Service and User Service it depends on.




```
public class Account {

    UserService userService ;
  AccountService accountService ;

    public Account() {
      this .userService =
          ServiceLocator .getService(UserService .class );
      this .accountService =
          ServiceLocator .getService(AccountService .class );
  }
}
```

Benefits of Service Locator

- The class does not have to manage any service dependency and its life cycle.
- It helps to test a class independently, without the availability of real services class depends on.
- It enables runtime optimization; as services can be registered and unregistered runtime.

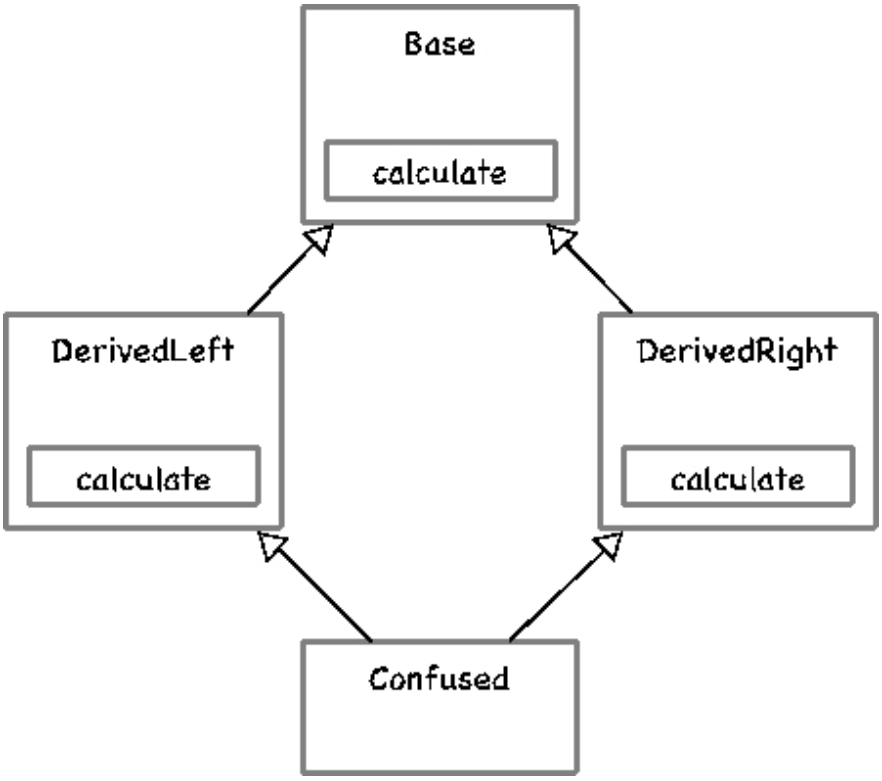
Questions

- Explain the Service Locator design pattern?
- What are the benefits of using a Service Locator design pattern?
- What is the difference between Service Locator and Dependency Injection pattern?
- When would you prefer Service Locator over Dependency Injection and vice versa?
- How does Service Locator help in testing?

DIAMOND PROBLEM

Java doesn't allow extending multiple classes because of the ambiguity that could arise when more than one super-classes have the same signature method, and the compiler can't decide which super-class method to use.

Consider the inheritance hierarchy set out in the figure below. If the method calculate() defined in the Base class is overridden by both, DerivedLeft and DerivedRight , then it creates ambiguity regarding which version of calculate() does the Confused class inherits.



In the code below there is an ambiguity regarding which version of calculate() should be called. This is known as Diamond Problem in Java.

```
public static void main (String [] args){
   Base base = new Confused ();
```

```
    base .calculate();
}
```

Diamond Problem with Default Interface Method

Diamong problem is possible with the introduction of Default Interface methods. If Base , DerivedLeft, and DerivedRight are Interfaces, and there exists calculate() as default interface method is all three, it will cause the Diamond Problem.

In such a scenario the Confused class has to explicitly re-implement the calculate() method; otherwise, the ambiguity will be rejected by the compiler.

Questions

- Explain the Diamond Problem in Java?
- Why Java does not provide multiple inheritances?
- Using default interface methods, a class can still inherit two interfaces with the same signature method; would this not cause Diamond Problem? How can you solve it?

PROGRAMMING TO INTERFACE

Programming to interface forms the basis for modular software development by facilitating decoupling between software components. A high level of decoupling improves maintainability, extensibility, and testability of software components. Modular software design also helps to improve speed to market, as it facilitates parallel software development between multiple teams working with the same code base.

It's the Programming to Interface design paradigm that forms the foundation for Inversion of Control, which manages dependency relationships in any large software application.

Let take a very simple example. Suppose we have a method to sort a collection which accepts Map Interface as the parameter. It means that the sort() method is not tied to any specific type of Map i mplementation and you can pass any concrete implementation of Map interface.

```
public static void main ( String [] args){
   sort( new HashMap<>());
   sort( new TreeMap<>());
   sort( new ConcurrentSkipListMap<>());
   sort( new TreeMap<>());
}

public static void sort(Map map){
   // perform sort
}
```

Benefits of programming to interface

- Based on the context, you can runtime select the most appropriate behavior.
- For testing, you can pass mock objects or stubs implementation.
- The interface/API definitions or the contract does not change frequently.
- Programming to Interface also facilitates parallel development between teams, as developers from different ream can continue writing code against an interface before doing final integration.

Questions

- What is the concept of programming to interface?
- What are the benefits of programming to interface?
- How does programming to interface facilitate decoupling between software components?
- How dependency injection and programming to the interface are inter-related? Can you achieve dependency injection without supporting programming to interface?
- What are the benefits of modular software?
- How does programming to interface help with unit testing?

ABSTRACT CLASS VS INTERFACE

Abstract Class

An Abstract class cannot be instantiated but can be extended. Extend abstract class when you want to enforce a common design and implementation among classes, and expect the derived class to implement required specialized functionality.

Interface

The Interface is a set of related methods, which defines its behavior and its contract with the outside world. Use an interface when you want to define common behavior among unrelated classes. An interface can also be defined without any method and such interface is known as marker interface; such interfaces are generally used to categorize the classes. An example of a marker interface is java.io.Serializable , which does not define any method but must be implemented by the classes that support serialization.

Difference between Abstract Class and Interface

- An Abstract class can be updated to add more capabilities to a class whereas Interface can be added to implement new behavior to the class. Though with the introduction of default interface methods , even Interfaces can be extended to have more capabilities.
- An Interface can be multiple inherited; whereas, an abstract class cannot.
- Interface define behavior, so it can even be applied to unrelated classes; whereas, related classes extend Abstract class.

- Abstract class methods can have any type of access modifier; whereas, Interface has all public members.
- An Abstract class can have state associated, which is not possible with Interface.
- An Abstract class can be extended without breaking the classes that extend it; whereas, any change in Interface, except for non-default and non-static methods, will break the existing implementation.

Questions

- If an abstract class cannot be instantiated, then what could be a need to define a constructor for an Abstract class?
    - *A constructor can be used to perform the required field initialization and also to enforce class constraints.*
- Define Abstract class? What role an Abstract class plays in a class design?
- Define Interface? What role does Interface play in class design?
- When would you prefer using Abstract class over Interface and vice-versa?
- Explain various differences between Abstract Class and Interface?
- What are marker interfaces? How are marker interfaces used?
- Can you declare an interface method static?
- With the introduction of default interface methods; how Abstract class is still different from an Interface?

INTERNATIONALIZATION AND LOCALIZATION

Internationalization

The internationalization of software is the process to ensure that software is not tied to only one language or locale. Its shortened name is i18n .

Localization

The localization of software is the process to ensure that software has all the resources available to support a specific language or locale. Its shortened name is l10n .

Note

Internationalization facilitates localization.

Questions

- What is Internationalization?
- What is localization?
- What is the difference between localization and internationalization?
- Can you achieve localization without building support for Internationalization?

IMMUTABLE OBJECTS

An object is considered immutable when there is no possibility of its state change after its construction.

Advantages

- It is easier to design and implement an immutable object, as you don't have to manage state change.
- Immutable objects are inherently thread-safe because they cannot be modified after creation. So there is no It is easier to design and implement an immutable object, as you don't have to manage state change.
- Immutable objects are inherently thread-safe because they cannot be modified after creation. So there is no need to synchronize access to it.
- Immutable objects reduce Garbage Collection overhead.
- need to synchronize access to it.
- Immutable objects reduce Garbage Collection overhead.

Disadvantages

- A separate object needs to be defined for each distinct value because you cannot reuse an Immutable object.

The rule for defining Immutable Objects

- Declare the class final .
- Allow only the constructor to create the object. Don't provide field setter.
- Mark all the fields private .

Example of an immutable class, *Employee* .

```
final public class Employee {

    final private int id ;
   final private String name ;
   final private String department ;

    public Employee( int id,
                String name,
                 String department) {
      this . id = id;
      this . name = name;
      this . department = department;
  }

    public int getId() {
      return id ;
  }

    public String getName() {
      return name ;
  }

    public String getDepartment() {
      return department ;
  }
}
```

Questions

- What is an immutable object?
- What are the rules for defining an immutable object?
- What are the advantages/disadvantages of an immutable object?
- How do you create an immutable object?

- What are the different situations where you can use immutable objects?
- What is the difference between a final and an immutable object?
- What is the optimization benefit of declaring a variable final?
- Can you list some of the problems with Immutability?
    - *It's harder to define constructors with loads of arguments.*
    - *Since the enforcement of immutability is left to the developer, even a single setter added accidentally can break it.*

CLONING

Cloning is the process of creating a copy of an object.

Simply assigning an existing object reference to an object, results in two references pointing to the same object.

There are two types of cloning, shallow cloning and deep cloning.

Shallow Cloning

Shallow cloning simply copies the values of the properties; for primitive property members, an exact copy is created; and for reference type members, its address is copied. So, for the reference type members, both original and the newly created copy, point to the same object in heap.

Deep Cloning

Deep cloning recursively copies the content of each member to the new object. Deep cloning always creates an independent copy of the original object. To create a deep clone, a dedicated method known, as CopyConstructor should be written.

Questions

- What is cloning?
- What is shallow cloning?
- Explain drawbacks with shallow cloning?
- What is deep cloning?
- What is CopyConstructor?
- When would you prefer deep cloning over shallow cloning and vice versa?

DATA TYPES

NAN

Not a Number also is known as NaN , is the undefined result produced because of arithmetic computations like divide by zero, operating with infinity, etc. No two NaNs are equal.

NaNs are of two types:

Quiet NaN - When a quiet NaN is results, there is no indication unless the result is checked.
Signalling NaN - When a signaling NaN results, it signals invalid operation expression.

Questions

- What is NaN or Not a Number?
- What is Quiet NaN?
- What is Signalling NaN?
- Are two NaNs equal?

ENUMSET

- The EnumSet is a specialized set implementation to be used with an Enum type.
- The EnumSet is represented internally as bit vectors, in a very compact and efficient manner.
- The EnumSet provides optimized implementation to perform bit flag operations and should be used in place of performing int flag operations.

The following code demonstrates the usage of EnumSet.

```
private enum Vehicle {
    CAR ,
    JEEP ,
    MOTORCYCLE ,
    SCOOTER
};



public static void main( String [] args){
   EnumSet< Vehicle > TWOWHEELERS =
                EnumSet.of( Vehicle . MOTORCYCLE ,
                Vehicle . SCOOTER );
    if ( TWOWHEELERS .contains( Vehicle . MOTORCYCLE ){
   }
}
```

Questions

- What is EnumSet?

- Why should you prefer EnumSet to perform int based bit flag operations?

COMPARING THE TYPES

Primitive types can be compared only by using the equality operators (== and !=); whereas, reference types can be compared using both equality operator and equals() method, depending upon what we want to compare.

Equality Operators

For reference types, the equality operators == and != are used to compare the addresses of two objects in memory and not their actual content.

.equals() Method

Use equals() method when you want to compare the content of two objects. Object class in Java defines equals() method, which must be overridden by the subclasses to facilitate content comparison between its objects. If the equal() method is not overridden by a class, then the equals() method of the java.lang.Object class is called, which uses an equality operator to compare references.

Questions

- What are the different ways to compare types in Java?
- What does the equality operator compare for the reference types?
- What does the equals method compare?
- When would you prefer using equals method to equality operator?
- What happens if a class does not override the equals method?

Float Comparison

Two float numbers should not be compared using equality operator ==; as the same number, say 0.33, may not be stored in two floating-point variables exactly as 0.33, but as 0.3300000007 and 0.329999999767.

So to compare two float values, compare the absolute difference of two values against a range.

```
if ( Math .abs( floatValue1 - floatValue2 ) < EPSILON ){
    //
}
```

where EPSILON is a very small number like 0.0000001, and that depend on the desired precision.

Questions

- Why shouldn't you use the equality operator to compare float value?
- What is the preferred way to compare two float values?
- What should be the criteria to decide on the value of the range that should be used to compare two float values?

String Comparison

String class object uses equality operator , ==, to tests for reference equality and equals() method to test content equality.

You should always use equals() method to compare the equality of two String variables from different sources. Interned Strings can be compared using equality operators too.

Questions

- Why should you use the equals method to compare String objects?
- What is the pitfall of using an equality operator to compare two String objects?
- What are interned string? How can you compare two interned strings?

Enum Comparison

Enum can neither be instantiated nor copied. So always only a single instance of the enum is available, the one defined with the enum definition.

As only one instance of the enum is available, you can use both equality operator (==) and equals() method for comparison. But prefer using equality operator, ==, as it does not throw NullPointerException and it also performs compile-time compatibility check.

Questions

- What are the different ways to compare two enums?
- Explain why you can use both equality operator and equals method to compare enum?
- Which is the preferred way to compare two enum values?

ENUM VS PUBLIC STATIC FIELD

The code below demonstrates the usage of enum and public static field.

The following are advantages of using enum over public static int.

- The Enums are compile-time checked, whereas int values are not. With public static int , you can pass any int value to AddVehicle() method.
- An int value needs to be validated against an expected range; whereas, enums are not.
- Bitwise flag operations are built into enumSet.

enum

```
private enum Vehicle {
    CAR ,
    JEEP ,
    MOTORCYCLE ,
    SCOOTER
};


public void AddVehicle( Vehicle vehicle){
}
```

public static field

```
public static int CAR = 0 ;
public static int JEEP = 1 ;
public static int MOTORCYCLE = 2 ;
public static int SCOOTER = 3 ;
```

```
public void AddVehicle( int vehicle){
}
```

Questions

- What are the advantages of using enum over public static int fields?
- Why do you need to perform extra validation with the int parameter as compared to the enum parameter?
- Why should you prefer using an enum over a public static int?

WRAPPER CLASSES

Each Primitive data type has a class defined for it that wraps the primitive datatype into the object of that class. Wrapper classes provide lots of utility methods to operate on primitive data values. As the wrapper classes enable primitive types to convert into reference types, these can be used with the collections too.

Questions

- What are the wrapper classes?
- What are the advantages of using the wrapper type over the primitive type?
- How can you use primitive types with collections?

AUTO BOXING AND AUTO UNBOXING

Auto boxing is an automatic conversion of primitive type to an object, which involves dynamically memory allocation and initialization of corresponding Wrapper class object. Auto unboxing is the automatic conversion of a Wrapper class to a primitive type.

In the code below, value 23.456f is auto boxed to an object Float(23.456f) and the value returned from addTax() is auto unboxed to float.

```
public static void main( String [] args){
    float beforeTax = 23.456f ;
    float afterTax = addTax( beforeTax );
}

public static Float addTax( Float amount){
   return amount * 1.2f ;
}
```

Questions

- What are auto boxing and auto unboxing?
- What are the advantages of auto boxing?

BIGINTEGER AND BIGDECIMAL

BigInteger and BigDecimal are used to handle values that are larger than Long.MAX_VALUE and Double.MAX_VALUE. Such large values are passed as String values to the constructor of BigInteger and BigDecimal . BigDecimal supports utility methods to specify the required rounding and the scale to be applied.

```
BigInteger bInt =
      new BigInteger( "98765432109876543210987 65" );
```

Both BigInteger and BigDecimal objects are immutable, so any operation on it creates a new object. BigInteger is mainly useful in cryptographic and security applications.

Questions

- What are the BigInteger and BigDecimal types?
- How are the values of BigInteger and BigDecimal internally stored?
- What are the usages of BigInteger?

STRINGS

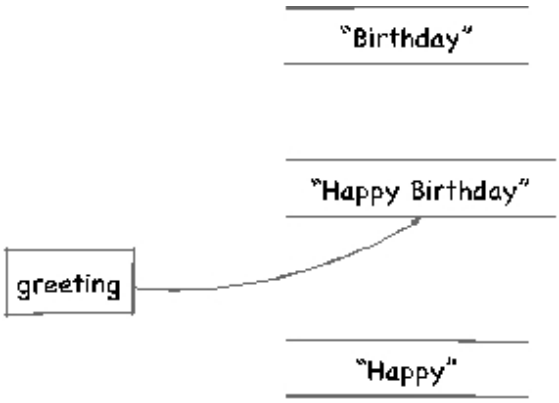
STRING IMMUTABILITY

The String object is immutable, which means once constructed, the object which String reference refers to, can never change. Though you can assign the same reference to another String object.

Consider the following example:

```
String greeting = "Happy" ;
greeting = greeting + " Birthday" ;
```

The code above creates three different String objects, " Happy ", " Birthday " and " Happy Birthday ".



- Though you cannot change the value of the String object you can change the reference variable that is referring to the object. In the above example, the String reference greeting starts referring to the String object "Happy Birthday".
- Note that any operation performed on String results in the creation of a new String.
- String class is marked final , so it's not possible to override the immutable behavior of the String class.

Advantages

- As no synchronization needed for String objects, it's safe to share a String object between threads.
- As String does not change, the Java environment caches String literals into a special area in memory known as a String Pool for optimization. If a String literal already exists in the pool, the same string literal is shared.
- Immutable String values safeguard against any change in value during execution.
- As hash-code of String object does not change, it is possible to cache hash-code and not calculate every time it's required.

Disadvantages

- The String class cannot be extended to provide additional features.
- If loads of String literals are created, either as new objects or because of any string operation, it will put the load on Garbage Collector.

Questions

- Why String objects are called Immutable?
- How is the String object created in memory?
- What are the advantages/disadvantages of String Immutability?
- Why String objects are considered thread-safe?
- What are the advantages of declaring the String class final?
- What memory optimization is performed by the Java environment for strings?
- Why you don't have to re-calculate hash-code of String object over the time it's used?

STRING LITERAL VS OBJECT

String Literal

String literal is a Java language concept where the String class is optimized to cache all the Strings created within double quotes, into a special area known as String Pool.

```
String cityName = "London" ;
```

String Object

The String object is created using the new() operator into the heap, like any other object of reference type.

```
String cityName = new String ( "London" );
```

Questions

- What is String literal?
- What are the differences between a String Literal and a String Object?
- How are the String Literals stored?

STRING INTERNING

- String interning is a concept of storing only a single copy of each distinct immutable String value.
- When you define any new String literal it is interned . Same String constant in the pool is referred for any repeating String literal.
- String pool literals are defined not only at the compile-time but also during runtime. You can explicitly call a method intern() on the String object to add it to the String Pool , if not already present.
- Placing an extremely large amount of text in the memory pool can lead to memory leak or performance issues.

Note: Instead of using String object, prefer using a string literal so that the compiler can optimize it.

Questions

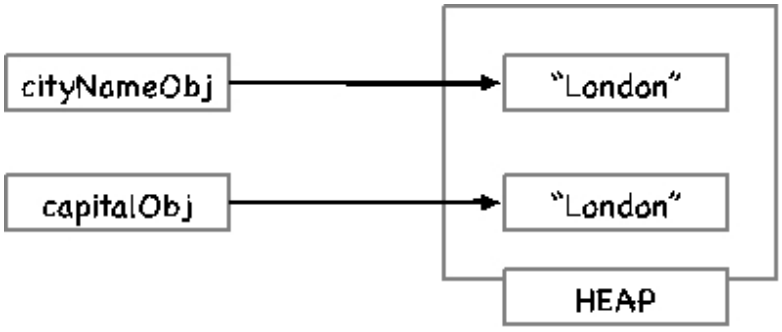
- What is String interning?
- Can String Objects also be interned?
- What happens when you store a new String literal value that is already present in the string pool?
- What are the drawbacks of creating a large number of String literals?
- Which one is preferred: String Object or String Literal? Why?

STRING POOL MEMORY MANAGEMENT

The String pool is a special area in memory managed by the Java compiler for String memory optimization. If there is already a String literal present in the string pool, compiler refers the new String literal reference to an existing String variable in the pool, instead of creating a new literal. Java compiler can perform this optimization because String is immutable.

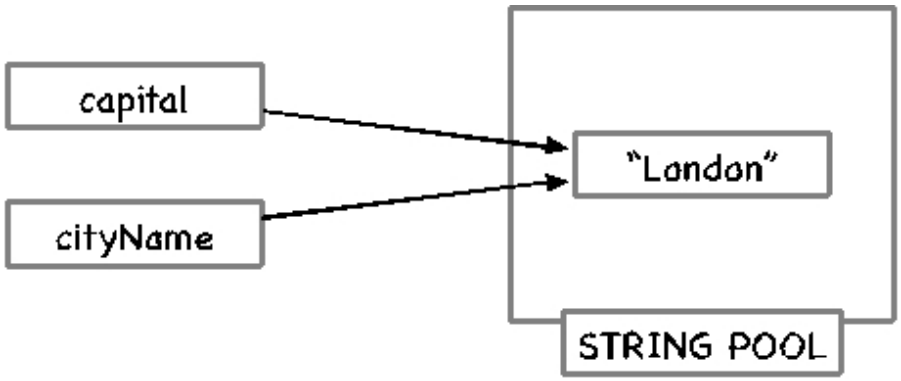
In this example below, both the String objects are different and are stored into Heap .

```
String cityNameObj = new String( "London" );

String capitalObj = new String ( "London" );
```



Whereas in this example below, both String literal reference to the same object in the memory pool.

```
String cityName = "London" ;

String capital = "London" ;
```



Questions:

- Explain String Pool Memory Management?
- How are String Literals stored in memory?
- How String Pool is optimized for memory?
- How are String Objects stored in memory?
- Why can Java not use a String Pool-like mechanism to store objects of other types of data?

IMMUTABILITY - SECURITY ISSUE

It's the responsibility of the Garbage Collector to clear string objects from the memory; though you can also use reflection to do so, that's not recommended.

Since Strings are kept in String Pool for re-usability, chances are that the strings will remain in memory for a long duration. As String is immutable and its value cannot be changed, a memory dump or accidental logging of such String can reveal sensitive content like password or account number, stored into it.

So instead, it's advisable to use char array (char []) to store such sensitive information, which can be explicitly overwritten by an overriding content, thus reducing the window of opportunity for an attack.

Questions:

- How are String literals cleared from the String Pool?
- Can you use reflection to clear a String object?
- What are the security issues associated with the immutable Strings?
- Why should you not use Strings to store sensitive information such as passwords, access keys, etc.?
- What other data type can be used to store the password instead of String?

CIRCUMVENT STRING IMMUTABILITY

Immutability feature in String can be bypassed with reflection, though it is NOT recommended to use reflection to do so because it is a violation of security and is considered an attack. The following code demonstrates how reflection can be used to circumvent string immutability:

```
String accountNo = "ABC123" ;

Field field = String . class .getDeclaredField( "value" );
field .setAccessible( true );

char [] value = ( char []) field .get( accountNo );

// Overwrite the content
value [ 0 ] = 'X' ;
value [ 1 ] = 'Y' ;
value [ 2 ] = 'Z' ;

// Print "XYZ123"
System . out .println( accountNo );
```

Questions

- Can you override the String class to modify its immutability?
- Is it possible to circumvent string immutability?
- Is it recommended to circumvent string immutability using reflection? Why?

STRINGBUILDER VS STRINGBUFFER

Similarities

- Both StringBuilder and StringBuffer objects are mutable, so allows String values to change.
- The objects of both the classes are created and stored in heap.
- Both the classes have similar methods.

Differences

- StringBuffer methods are synchronized , so its thread-safe whereas StringBuilder is not.
- The performance of StringBuilder is significantly better than StringBuffer , as StringBuilder does not have synchronization overheads.

Note: If you need to share String objects between threads then use StringBuffer , otherwise StringBuilder .

Questions

- What are the similarities and differences between StringBuffer and StringBuilder?
- When would you prefer StringBuffer to StringBuilder?
- Between StringBuffer and StringBuilder, which one would you prefer in a single-threaded application?

UNICODE

Unicode is an international standard character encoding system that represents most of the written languages in the world. Before Unicode, there were multiple encoding systems prevalent: ASCII, KOI8, ISO 8859, etc., each encoding system having its code values and character set with different lengths. So to solve this issue, a uniform standard is created, which is known as Unicode . Unicode provides a platform and language independent unique number for each character.

Questions

- What are Unicode characters?
- What are the advantages of using Unicode characters?
- What are the encoding systems problems which predominated before Unicode?

INNER CLASSES

INNER CLASSES

Inner Class – is a class within another class.
Outer Class – is an enclosing class that contains the inner class.

Note

- The compiler generates a separate class file for each inner class.

Advantages of inner classes

- It's easy to implement callbacks using inner classes.
- The inner class has access to the private members of the enclosing class that aren't even available to the inherited classes.
- The inner class helps to enforce closures that make the enclosing instance and the surrounding scope available.
- The outer classes provide additional namespace to the inner classes.

Questions

- What is the inner class?
- What is the outer class?
- What are the advantages of defining an inner class?
- What are closures? How inner class can be used to create closures?

- What are callbacks? How inner class can be used to create callbacks?
- Can an inner class access private members of the enclosing outer class?
- What benefit does the outer class bring?

STATIC MEMBER NESTED CLASS

- A static nested class is declared as static inside a class like any other member.
- A static nested class is independent and has nothing to do with the outer class. It is generally used to keep together with the outer class.
- It can be declared public, private, protected, or at the package level.

Declaration of static nested class

```
// outer class
public class Building {
    // static member inner class
    public static class Block{


     }
}
```

Creating an object of static nested class

```
// instance of static member inner class
Building.Block block =
      new Building.Block();
```

Questions

- What is a static nested class?

- Static classes, nested or outer, are the same; then what's the benefit of defining an inner class as static?

LOCAL INNER CLASS

- A local inner class is declared and can be used inside the method block.
- It cannot be declared public, private, protected, or at the package level.

Creation of local inner class

```
// outer class
public class CityNames {
    private List <String> cityNames =
          new ArrayList<>();


     public Iterator <String> nameIterator(){
       // local inner class
       class NameIterator
            implements Iterator <String> {
          @Override
          public boolean hasNext() {
             return false ;
          }
          @Override
          public String next() {
             return null ;
          }
      }
       // return an instance of a local inner class.
       return new NameIterator();
   }
}
```

Note

- To be able to use the inner class outside, the local inner class must implement a public interface or Inherit a public class and override methods to redefine some aspects.

Questions

- What is the difference between an inner class and a local inner class?
- Why can't you use an access modifier with the local inner class?
- Explain the rules for defining local inner class?
- What problem does a local inner class solves?

NON-STATIC NESTED CLASS

- A non-static nested class is declared inside a class like any other member.
- It can be declared public, private, protected, or at the package level.
- Non-static nested classes are closures as they have access to enclosing instance.
- An object of the outer class is required to create an object of a non-static inner class.

Declaration of non-static nested class

```
// outer class
public class Building {
    // non-static member inner class
    public class Block{
   }
}
```

Creating an object of non-static nested class

```
// instance of the outer class
Building building =
        new Building();
// instance of non-static member inner class
Building.Block block =
        building . new Block();
```

Questions

- What is a non-static nested class?
- Why a non-static nested class can be used as closures?
- Can you create an instance of a non-static inner class without defining an outer class?

ANONYMOUS INNER CLASS

- Anonymous inner class does not have a name.
- The anonymous inner class is defined and its object is created at the same time.
- Anonymous inner class is always created using new operator as part of an expression.
- To create an Anonymous class, a new operator is followed by an existing interface or class name.
- The anonymous class either implements the interface or inherits from an existing class.

Creation of anonymous inner class

```
// outer class
public class CityNames {
    private List <String> cityNames =
          new ArrayList<>();

     public Iterator <String> nameIterator(){
       // Anonymous inner class
       Iterator <String> nameIterator =
            new Iterator <String> () {
         @Override
         public boolean hasNext() {
            return false ;
        }
         @Override
         public String next() {
            return null ;
```

```
        }
      };
      // return an instance of a local inner class.
      return nameIterator();
   }
}
```

Notes

- Do not return inner classes as it has reference to the outer enclosing class, otherwise memory leaks may occur .
- Use anonymous class when you want to prevent anyone from using the class anywhere else.
- Serialization of Anonymous and Inner classes must be avoided, as there could be compatibility issues during de-serialization, due to different JRE implementation.

Questions

- What is an anonymous inner class?
- How an anonymous inner class is different from the local inner class?
- Why you shouldn't return an inner class object from a method?
- If you want to prevent anyone from using your class outside, which type of inner class you should define?
- Why should you avoid serialization of anonymous and inner classes?

FUNCTIONAL PROGRAMMING

FUNCTIONAL INTERFACE

Functional Interface is an interface with only one abstract method but can have any number of default methods.

```
@FunctionalInterface
public interface Greator<T > {
   public T greater(T arg1, T args2);
}
```

Annotation @FunctionalInterface generates a compiler warning when the interface is not a valid functional interface.

```
@FunctionalInterface
public interface Greator<T > {
   public T greater(T arg1, T args2);
}
```

Account class, defined below, used as an argument to the functional interface Greater .

```
public class Account {
   private int balance ;

    public Account(int balance) {
      this .balance = balance;
   }

    public int getBalance() {
      return balance ;
   }

    @Override
    public String toString() {
      return "Account{" +
            "balance=" + balance +
            '}' ;
   }
}
```

The code below demonstrates the usage of Lambda expression to find the account with the greater balance. Similarly, the same Function Interface , Greater , can be used to compare other similar business objects too.

```
public static void main(String [] args){

    Greator<Account > accountComparer =
        (Account acc1, Account acc2) ->
            acc1.getBalance() > acc2.getBalance() ?
                acc1 :
                acc2;

    Account account1 =
        new Account(6 );
   Account account2 =
        new Account(4 );

    System .out .println(
        " Account with greater balance: "
         + accountComparer .greater(account2 , account1 ));
}
```

Java also provides a set of predefined functional interfaces for most common scenarios.

Questions

- What is Function Interface?
- What are the benefits of using Functional Interface?

LAMBDA EXPRESSION

Lambda expressions provide a convenient way to create an anonymous class . Lambda expressions implement Functional Interface more compactly. Lambda Expressions are primarily useful where you want to pass some functionality as an argument to another method and defer the execution of such functionality until an appropriate time.

Lambda expression can be just a block of a statement with method body and optional parameter types but without method name or return type. It can be passed as a method argument and can be stored in a variable.

```
// lambda expressions
() -> 123
(x,y) -> x + y
(Double x, Double y) -> x*y
```

Questions

- What is Lambda Expression?
- How are the Lambda Expression and Anonymous class-related?
- Can you pass Lambda Expression as a method parameter?
- What is the meaning of deferred execution of functionality, using a Lambda Expression?
- What are the benefits of using Lambda Expression?
- How are the Lambda Expression and Functional Interface related?

PURE FUNCTIONS

Pure functions are functions whose results depend only on the arguments passed onto them and are neither influenced by any change of state in the application nor altering the application state. Pure functions always return the same result for the same arguments.

```
public int increaseByFive( int original){
    int toAdd = 5 ;
    return original + toAdd ;
}
```

Questions

- What is a Pure Function?
- What is the use of Pure Function in Functional Programming?
- How is the Pure Function guaranteed to always return the same results for the same arguments?

FLUENT INTERFACE

A fluent interface is used to transmit commands to subsequent calls, without the need to create intermediate objects and is implemented by method chaining. The fluent interface chain is terminated when a chained method returns void . Fluent interface improves readability by reducing the unnecessary objects created.

In the code below, the Fluent Interface is used to add a new Employee.

```
employee.create()
    .atSite( "London" )
    .inDepartment( "IT" )
    .atPosition( "Engineer" );
```

Fluent interfaces are used primarily in scenarios where you construct queries, create a series of objects, or construct nodes in hierarchical order.

Questions

- What is the Fluent Interface?
- What are the benefits of defining and using Fluent Interface?
- Describe some usage of Fluent Interface?

GENERICS

GENERICS

Generics is a mechanism that allows the same code to work on objects of different types in a type, class or interface, or a method; while providing compile-time protection. Generics are introduced to enforce the safety of type, in particular for collection classes.

- Once a parameter type is defined in generics , the object will work with the defined type only.
- In generics , the formal type parameter (E in the case below) is specified with the type (class or interface)

```
// Generic List type
// E is type parameter
public interface List < E > {
   void add( E x);
   Iterator < E > iterator();
}
```

- In generics , the parameterized type (Integer in this case) is specified when a variable of type is declared or the object of the type is created.

```
// variable of List type declared
// With Integer parameter type
List <Integer> integerList =
       new ArrayList<Integer>();
```

Notes

- There is a convention for naming parameters commonly used in Generics . E for element, T for type, K for the key, and V

for values.

- A generic type is compiler support; all the type parameters are removed during the compilation and this process is called erasure .
- Due to strong type checking, generics help avoid many ClassCastException instances.

Generic Class Example

- The following NodeId generic class definition can be used with the object of any type.

```
// NodeId generic class with type parameter
public class NodeId< T > {
    private final T Id ;
    public NodeId( T id) {
       this . Id = id;
   }
    public T getId() {
       return Id ;
   }
}
```

- Usual SuperType – SubType rules do not apply to generics. In this example, even though Integer is derived from Object , still NodeId with type parameter Integer cannot be assigned to NodeId with type parameter Object .

```
// Parameter Type - Object
NodeId<Object> objectNodeId =
       new NodeId<>( new Object());
// Parameter Type - Integer
NodeId<Integer> integerNodeId =
       new NodeId<>( 1 );
// This results in an error
objectNodeId = integerNodeId ;
( error )
```

Questions

TYPE WILDCARDS

- A wildcard in generics is represented in the form of "?" . For example, a method which takes List<?> as a parameter will accept any type of List as an argument.

```
public void addVehicles( List <?> vehicles) {
    //...
}
```

- Optional upper and lower limits are set to enforce limitations, as the exact form of parameter defined by the wild card is not known.

```
// Only vehicles of type Truck can be added
public void addVehicles
     ( List <? extends Truck> vehicles) {
    //...
}
```

Notes

- Do not return wildcard in a return type as its always safer to know what is returned from a method.

Upper Bound

- To impose a restriction, upper bound can be set on the type parameters. Upper bound restricts a method to accept unknown type arguments extended only from the specified data type, like Number on the example below.

```
// Upper bound of type wild card
public void addIds( List <? extends Number> T){
```

Lower Bound

- To impose a restriction, lower bound can be set on the type parameters. Lower Bound restricts a method to accept unknown type argument, which is a super-type of specified data type only, like Float in the example below.

```
// Lower bound of type wild card
public void addIds( List <? super Float> T){
```

Type Inference

- Type inference is a compiler's ability to look at method invocation and declaration to infer the type arguments.
- In Generics , the operator called Diamond operator , <>, facilitates type inference.

```
// Type inference using Diamond operator
List <Integer> integerList =
      new ArrayList<>();
```

Questions

- What is Type Wildcard in Generics?
- Explain the Upper Bound type wildcard?
- Explain Lower Bound type wildcard?
- How are the different types of wildcards used in generics?
- What is automatic type inference in Generics? What is the diamond operator in Generics?

GENERIC METHOD

- Generic methods define their type parameters.

```
// generic method
public < T > void addId ( T id){
```

- If we remove the <T> from the above method, we will get a compilation error as it represents the declaration of the type parameter in a generic method.
- The type (class or interface) that has a generic method does not have to be of genetic type.
- While calling the generic method, we do not need to explicitly indicate the type parameter.

Notes

- Prefer using Generics and parameterized classes/methods to enforce compile-time type safety.
- Use the Bounded Type parameter to increase the flexibility of method arguments, at the same time it also helps to restrict the types that can be used.

Questions

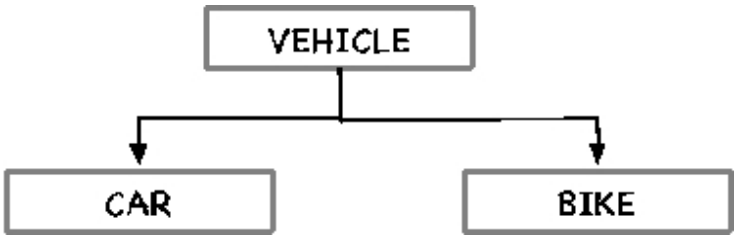
- Explain the Generics method?
- Can you add a Generic method to a non-Generic type?

- What are the benefits of defining the bounded type as method parameters?
- How is compile type safety enforced by Generics?

JAVA GENERICS VS JAVA ARRAY

Java Generics

Consider the following hierarchy,



As both Car and Bike are derived from Vehicle , is it possible to assign List<Car > or List<Bike > to a variable of List<Vehicle > ?

No, List<Car > or List<Bike > cannot be replaced with List<Vehicle >, because you cannot put a Bike in the same list that has cars. So even though Bike is a Vehicle , it's not is a Car .

```
List< Bike > bikes =
        new ArrayList< Bike >();


// suppose this is allowed
List< Vehicle > vehicles = bikes ;
vehicles .add( new Car ());


// Error - Bike and Vehicle are
// considered incompatible types.
```

```
Bike bike = vehicles .get( 0 );
```

Java compiler does this checking compile time for incompatible types. For more on this refer to Type Erasure .

JAVA ARRAY

However, unlike Generics, the array of Car can be assigned to the array of Vehicle :

```
Car [] cars = new Car [ 3 ];
Vehicle [] vehicles = cars ;
```

For Array, even though the compiler allows the above code to compile but when you run this code, you will get ArrayStoreException .

```
// this will result in ArrayStoreException
vehicles [ 0 ] = new Bike ();
```

So even though the compiler did not catch this issue, the runtime type system caught it. An array is reifable types, which means that run time is aware of its type.

Questions

- What are the reifable types?
- Why can't you assign a Generic collection object of sub-type to a Generic collection object of super-type?
- Why it's allowed to assign an array object of sub-type to an array object of super-type? Are Java array more polymorphic?

TYPE ERASURE

For Java generic types, due to a process known as type erasure , Java compiler discards the type information and it is not available at run time. So because the type information is not available runtime, java compiler takes extra care to stop you at compile time itself, preventing any heap pollution.

Generics type is also of non-reifiable types, which means that its type information is removed during the compile time.

Questions

- What are non-reifable types?
- What is type erasure?
- What would happen if type erasure is not there?

CO-VARIANCE

Covariance is a concept that you can read items from a generics defined with the upper bound type wildcard, but you cannot write anything into the collection.

Consider the following declarations with upper bound type wildcard:

```
List<? extends Vehicle > vehicles =
      new ArrayList<Bike >();
```

You are allowed to read from vehicles generic collection because whatever is present in the list is sub-class of Vehicle and can be up-casted to a Vehicle .

```
Vehicle vehicle = vehicles .get(0 );
```

However, you are not allowed to put anything into a covariant structure.

```
// This is a compile-time error
vehicles .add(new Bike ());
```

This would not be allowed, because Java cannot guarantee the actual type of the object in the generic structure. It can be anything that extends Vehicle , but the compiler cannot be sure. So you can read, but not write.

Questions

- Explain co-variance?
- Why can't you add an element of subtype to a generic defined with upper bound type wildcard?

CONTRA-VARIANCE

Contra-variance is the concept that you can write items to a generic defined with lower bound type wildcard , but you cannot read anything into the collection.

Consider the following declarations with lower bound type wildcard:

```
List<? super Car > cars = new ArrayList<Vehicle >();
```

In this case, even though the ArrayList is of type Vehicle but you can add Car into it through contra-variance; because Car is derived from Vehicle .

```
cars .add(new Car ());
```

However, you cannot assume that you will get a Car object from this contra-variant structure.

```
// This is a compile-time error
Car vehicle = cars .get(0 );
```

Questions

- Explain contra-variance?
- Why can't you read an element from a Generic defined with lower bound type wildcard?

CO-VARIANCE VS CONTRA-VARIANCE

- Use covariance when you only intend to read generic values from the collection.
- Use contra-variance when you only intend to add generic values into the collection.
- Use the specific generic type when you intend to do both, read from, and write to the collection.

Questions

- When should you use co-variance?
- When should you use contra-variance?

COLLECTIONS

COLLECTIONS

Collections are the data structures that are basic building blocks to create any production level software application in Java. Interviewers are interested in understanding different design aspects related to the correct usage of collections. Each collection implementation is written and optimized for a specific type of requirement, and interview questions are to gauge the interviewee's understanding of such aspects.

Questions are often asked to check whether the candidate understands the correct usage of collection classes and is aware of alternative solutions available.

Following are few aspects on which questions on collections are asked:

- Collection types in Java.
- Unique features of different collection types.
- Synchronized collection.
- Concurrent collection.
- Ordering of elements in a collection.
- Speed of reading from collection.
- Speed of writing to a collection.
- The uniqueness of the elements in a collection.
- Ease of inter-collection operation.
- Read-only collections.
- Collection navigation.

COLLECTION FUNDAMENTALS

The collection is a container that groups multiple elements. Following is a simple example of a collection.

```
// Create a container list of cities
List <String> cities = new ArrayList<>();
// add names of cities
cities .add( "London" );
cities .add( "Edinburgh" );
cities .add( "Manchester" );
```

Notes

- Collections work with reference types.
- All collection interface implementation is Generic .
- Unlike arrays, all collection types can grow or shrink in size.
- Java provides a lot of methods to manipulate collections based on their use, so always test the existing methods before you implement one.

Collection Framework

The collection framework is defined by the following components.

- Interfaces - are the abstract types defined for each specific type of usage and collection type.
- Implementation - are concrete implementation classes to create an object to represent a different type of collection.

- Algorithms - are applied to the collections to perform various computations and to manipulate the elements in the collection.

Collection Framework helps you to reduce programming efforts, by providing data structures and algorithms to operate on them.

Questions

- Explain collections?
- How collections and Generics related?
- Can you use collection with primitive types?
- How can you use collection with the primitive types?
- Explain the difference between collections and arrays?
- What is the benefit of the collection framework?
- What are the different components of the collection framework?

COLLECTION INTERFACES

An interface defines its behavior in the form of the signature of the methods. To use a collection, you should always write code against collection interfaces and not the class implementations, so that the code is not tied to a specific implementation. This protects from possible changes in the underlying implementation class.

The following are the most important interfaces that define collections and their behavior. Each child node below is inherited from its parent node.

+ Collection
    + Queue
        + BlockingQueue
            + TransferQueue
            + BlockingDeque
        + Deque
    + BlockingDeque
    + List
    + Set
        + SortedSet
        + NavigableSet

+ Map
    + SortedMap

Questions

- Why should you write code against the collection interface and not concrete implementation?

COLLECTION TYPES

- Set - Set contains unique elements.
- List - List is an ordered collection.
- Queue - Queue holds elements before processing in a FIFO manner.
- Deque - Deque holds elements before processing in both, FIFO and LIFO manner.
- Map - The map contains the mapping of keys to corresponding values.

Questions

- What are the different collection types?
- Define Set collection Type?
- Define List collection Type?
- Define Queue collection Type?
- Define Deque collection Type?
- Define Map collection Type?
- What is the difference between Queue and Deque?

Set

Basic Set

```
// Create a set
Set <String> set =
        new HashSet<>();
```

- Set is a collection of unique elements.
- Elements in the Set are stored unordered.
- Only one null element can be added to a Set .
- Duplicate elements are ignored.
- When ordering is not needed, Set is the fastest and has a smaller memory footprint.

Linked Hashset

```
// Create a linkedHashSet
Set <String> linkedHashSet =
        new LinkedHashSet<>();
```

- LinkedHashSet keeps the Set elements in the same order in which they were inserted.
- Insertion order is not affected in LinkedHashSet if an element is re-inserted.
- Iterator in LinkedHashSet returns elements in the same order in which these were added to the collection.

Sorted Set

- SortedSet imposes the ordering of elements to be either sorted in natural order by implementing a Comparable interface or custom sorted using a Comparator object.
- TreeSet is an implementation class for the SortedSet interface.

```
// Create a sorted set of city names
SortedSet <String> cityNames =
    new TreeSet<>();
```

- Use a Comparator object to perform custom sorting.

```
SortedSet <City> sortedCitiesByName =
    new TreeSet<>( Comparator .comparing(
        City::getName));
```

- If an element implements a Comparable interface, then the compareTo() method is used to sort in the natural order.

Navigable Set

- NavigableSet inherits from the SortedSet and defines additional methods.
- NavigableSet can be traversed in both, ascending and descending order.
- TreeSet is one of the implementation classes for the NavigableSet interface.

Questions

- Can you add duplicate elements in a Set?
- Which collections type should you use when ordering is not a requirement? Why?
- Can you add a null element to a Set?
- Which Set class should you use to maintain the order of insertion?
- What happens to the order, if the same element is inserted again in a Set? Will it maintain its original position or inserted in the end?
- Which Set class should you use to ensure the ordering of the elements?
- Which Set class should you use when you need to traverse in both the directions?

List

- The list is an ordered collection of objects.
- The list can have a duplicate.
- The list can have multiple null elements.
- In List, an element can be added at any position.
- Using ListIterator, a list can be iterated in both, forward and backward direction.

```
// Create a list of cities
List <String> cities =
        new ArrayList<>();
```

ArrayList

- ArrayList is based on an array.
- It performs better if you access elements frequently.
- Add and remove is slower in the ArrayList. If an element is added anywhere but the end requires shifting of the element. If elements are added beyond its capacity then the complete array is copied to a newly allocated place.

Linked List

- LinkedList is based on the list.
- LinkedList is slower in accessing elements and only sequential access allowed.
- Adding and removing elements from LinkedList is faster.
- LinkedList consumes more memory than ArrayList as it keeps pointers to its neighboring elements.

List Iterator

- ListIterator iterates list in both the directions

```
// full list iterator
ListIterator <String> fullIterator =
        cityList .listIterator();

// partial list iterator starts at index 3
ListIterator <String> partialIterator =
        cityList .listIterator( 3 );
```

Notes

- Random access is better in ArrayList as it maintains an index-based system for its elements whereas LinkedList has more overhead as it requires traversal through all elements.

Questions

- Can you add duplicate elements to a List?
- Can you add null to a list?
- In which scenarios would you use ArrayList in: frequent add/update or frequent reading?
- In which scenarios would you use LinkedList in: frequent add/update or frequent reading?
- Which one takes up more memory between ArrayList and LinkedList? Why?
- Between ArrayList and LinkedList, which one would you prefer if you need random access? Why?

Queue

- A queue is a collection designed to hold elements before processing.
- A queue has two ends, a tail, and a head.
- A queue works in a FIFO manner, first in and first out.

Types of Queues

- Queue - simple queue which allows insertion at tail and removal from the head, in a LIFO manner.
- Deque - allows insertion and removal of elements from head and tail.
- Blocking Queue - blocks the thread to add element when its full and also blocks the thread to remove element when it's empty.
- Transfer Queue - special blocking queue where data transfer happens between two threads.
- Blocking Deque - a combination of Deque and blocking queue .
- Priority queue - element with the highest priority is removed first.
- Delay queue - element is allowed to be removed only after the delay associated with it has elapsed.

Basic Queue

- A queue has one entry point (tail) and one exit point (head).
- If entry and exit point is the same, it’s a LIFO (last in first out) queue .

```
// simple queue
Queue <String> queue =
      new LinkedList<>();
```

Priority Queue

- In the PriorityQueue, a priority is associated with the elements in the queue.

- Element with the highest priority is removed next.
- PriorityQueue does not allow the null element.
- An element of queue either implement a Comparable interface or use a Comparator object to calculate priority.

```
// City class implements Comparable interface
Queue <City> pq =
      new PriorityQueue<>();
```

Deque

- Deque allows insertion and removal from both ends.
- Deque does not provide indexed access to elements.
- Deque extends the Queue interface.

```
// Create a Deque
Deque <String> deque =
      new LinkedList<>();
```

Blocking Queue

- BlockingQueue interface inherits from Queue .
- BlockingQueue is designed to be thread-safe.
- BlockingQueue is designed to be used as producer-consumer queues.

Transfer Queue

- TransferQueue extends BlockingQueue .
- It may be capacity bounded, where Producer may wait for space availability and/or Consumers may wait for items to become available.

```
// Transfer Queue
TransferQueue <String> ltq =
      new LinkedTransferQueue<String>();
```

Questions

- What is the purpose of the Queue?
- How is Deque different from Queue?
- What is a priority queue?
- What criteria should be used to delete an item from the priority queue?
- What is a Delay Queue?
- What is a Blocking Queue?
- Which Queue is a thread-safe queue?
- Which Queue implements the Producer-Consumer pattern?
- Explain the difference between the Blocking Queue and the Transfer Queue?
- Which Queue is capacity bounded, allowing only the specified number of elements in Queue? What happens if you try to put more than the capacity?

Map

- The map contains key-value mapping.
- Usually, Map allows one null as its key and multiple null as values, but it’s left to the Map's implementation class to define restrictions.

Implementation

HashMap

- HashMap is based on a hash table.
- HashMap does not guarantee the order of the elements in the map.
- HashMap's hash function provides constant-time performance for the get() and the put() operations.
- HashMap allows one null for the key and multiple nulls for the value.

```
// Create a map using HashMap
Map <String, String> hashMap =
      new HashMap<>();
```

LinkedHashMap

- LinkedHashMap is a hash table and linked list implementation of Map interface.
- LinkedHashMap stores entry using a doubly-linked list.
- Use LinkedHashMap instead of HashSet if the insertion order is to be maintained.
- The performance of HashMap is slightly better than LinkedHashMap as LinkedHashMap has to maintain the linked list too.
- It ensures iteration over entries in its insertion order.

```
// LinkedHashMap
LinkedHashMap lhm =
      new LinkedHashMap();
```

WeakHashMap

- WeakHashMap stores only weak references to its keys.
- WeakHashMap supports both null key and null values.
- When there is no reference to keys, they become a candidate for garbage collection.

```
// WeakHasMap
Map map   =
        new WeakHashMap();
```

Sorted Map

- SortedMap provides complete ordering on its keys.
- Sorts the map entries on keys based either on natural sort order (Comparable) or custom sort order (Comparator).
- SortedMap interface inherits from Map.

```
// sorted map
SortedMap <String,String> sm =
        new TreeMap<>();
```

Navigable Map

- NavigableMap extends the SortedMap interface.
- TreeMap class is the implementation of NavigableMap .
- NavigableMap can be accessed in both, ascending and descending order.
- It supports navigation in both directions and getting the closest match for the key.

```
// Create a Navigable Map
NavigableMap <String,String> nm =
        new TreeMap<>();
```

Concurrent Map

- ConcurrentHashMap i s concrete implementation of ConcurrentMap interface.
- ConcurrentMap uses a fine-grained synchronization mechanism by partitioning the map into multiple buckets and locking each bucket separately.
- ConcurrentHashMap does not lock the map while reading from it.

```
// Create a Concurrent Map
ConcurrentMap <String,String> cm =
      new ConcurrentHashMap<>();
```

Notes

- The Map keys and Set items must not change state, so these must be immutable objects.
- To avoid implementation of hashCode() and equals() , prefer using immutable classes provided by JDK as key in Map .
- Never expose collection fields to the caller, instead provide methods to operate on those.
- HashMap offers better performance for inserting, locating, and deleting elements in a map.
- TreeMap is better if you need to traverse the keys in sorted order.

Questions

- How Map is different from other collections?
- Can you add a null value as the key to a Map?
- Can you guarantee ordering of elements in HashMap?
- How HashMap can provide constant-time performance for the get() and the put() operations.
- What is the difference between HashMap and LinkedHashMap?
- If the insertion order is to be maintained, which one should you use: LinkedHashMap or HashSet? Why?

- What is WeakHashMap?
- If do you want to maintain the ordering of key, which Map class should you use?
- Which Map class should you use to access it in both ascending and descending order?
- Which Map implement is optimized for thread safety?
- How does ConcurrentHashMap enforce scalability, with good efficiency still maintained?
- Why the key used for Map and value used for Set should be of immutable type?
- What are the disadvantages of exposing an internal collection object to the caller?
- Which collection is best for traversing in sorted order?

ALGORITHMS

- Sorting - is used to sort the collection in ascending or descending order. You can either use the natural order if the key class has implemented a Comparable interface or need to pass the Comparator object for custom sorting.
- Searching - BinarySearch algorithm is used to search keys.
- Shuffling - it re-orders the elements in the list in random order; it is generally used in games.
- Data Manipulation - reverse, copy, swap, fill, and addAll algorithms are provided to manipulate the usual data on the collection elements.
- Extreme Value - min and max algorithms are provided to find the minimum and maximum values in the specified collection.

Questions

- What are the different algorithms supported by the collection framework?
- Which search algorithm is used to search keys?
- How is the shuffling algorithm used?

COMPARABLE VS COMPARATOR

Comparable interface is implemented to sort the collections in a natural order and the Comparator object is used to perform a custom sort on the collections.

If an element implements a Comparable interface, then the compareTo() method is used to sort in the natural order. The CompareTo() method compares the specified object with the existing object for order. It returns a negative integer, zero, or a positive integer as the existing object is smaller than, equal to, and greater than the specified object.

Comparable interface implementation

```
@Override
public int compareTo( Employee employee) {
    //comparison logic
}
```

A Comparator object contains logic to compare two objects that might not align with the natural ordering. Comparator interface has compare() method, which takes two objects to return a negative integer, zero or a positive integer as the first argument is smaller than, equal to and greater than the second.

Using a Comparator object to perform custom sorting.

```
SortedSet <City> sortedCitiesByName =
        new TreeSet<>( Comparator .comparing(
            City::getName));
```

Comparator is generally used to provide a custom comparison algorithm in a situation when you do not have complete control over the class.

Questions

- What is the difference between the Comparable Interface and Comparator Class?
- When should you implement Comparable Interface to sort a collection?
- When do you use Comparator Class to sort a collection?

HASHCODE() AND EQUALS()

- The HashTable , HashMap, and HashSet uses hashCode() and equals() methods to calculate and compare objects that are used as their key.
- All classes in Java inherit the base hash scheme from the Object base class unless they override the hashCode() method.
- Any class that overrides hashCode() method is supposed to override equals() method too.

Implementation of hashCode() and equals()

```
final public class Employee {

    final private int id ;
   final private String name ;
   final private String department ;

    @Override
   public boolean equals( Object o) {
      if ( this == o) return true ;
      if (o == null || getClass() !=
            o.getClass())
         return false ;

          Employee employee = ( Employee ) o;

          if ( id != employee . id ) return false ;
      if ( name != null ?
            ! name .equals( employee . name ) :
```

```
            employee . name != null )
        return false ;

        return !( department != null ?
         ! department .equals( employee . department ) :
          employee . department != null );

    }

    @Override
  public int hashCode() {
    int result = id ;
    result = 31 * result +
        ( name != null ?
            name .hashCode() :
            0 );
    result = 31 * result +
        ( department != null ?
            department .hashCode() :
            0 );
    return result ;
  }

}
```

Rules of hashCode and equals

- Once an object is created, it must always report the same hashCode() during its lifetime.
- Two equal objects of the same class shall return the same hash-code.

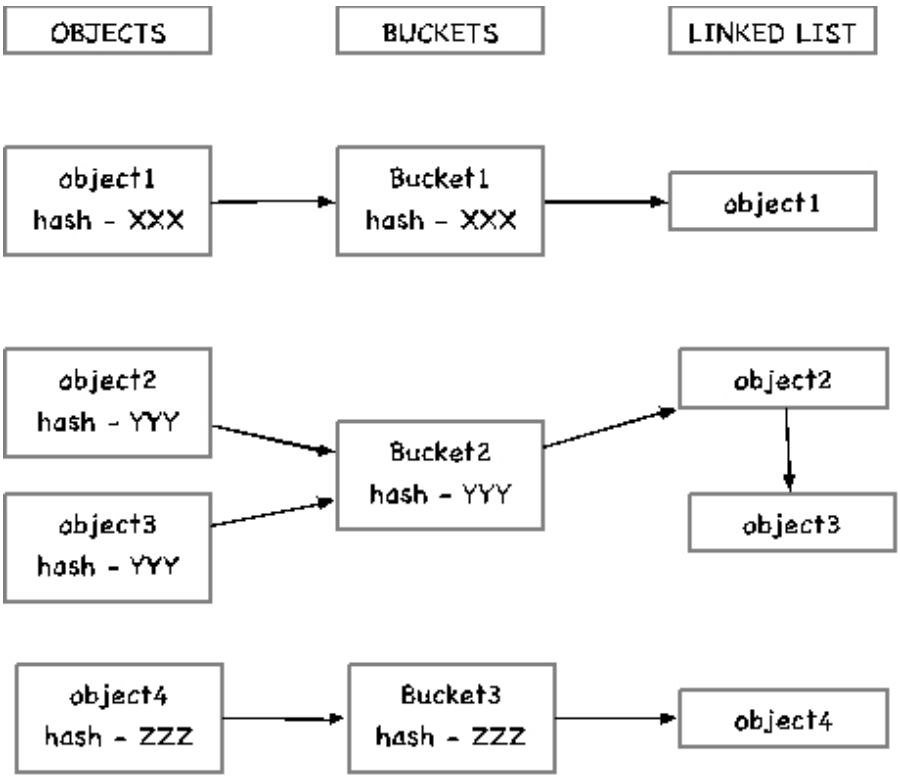
Note

Two objects, which are not equal, can have the same hash-code.

How hashCode() and equals() are used

- The hashCode() and equals() method is used to segregate items into separate buckets for lookup efficiency. If two items have the same hash-code then both of these will be stored in the same bucket, connected by a linked list.
- If the hash-code of two objects is different, then the equals() method is not called; otherwise, the equals() method is called to compare the equality.

The figure below conceptually demonstrates the storage of objects in a hash bucket.



Questions

- What are the different collection types in Java that use hashCode and equals methods? Why do these collections type use hashCode and equals method?
- If you override the hashCode method, why should you always override the equals method too?
- The hashCode method is already present in the java.lang.Object class, why should you override it?
- Why does an object have to return the same hash-code value every time during its lifetime?
- Is it necessary that an object returns the same hash-code every time program runs?
- Is it necessary that two equal objects return the same hash-code?
- Can two objects, that are not equal, still have the same hash-code?
- What is the purpose of calculating hash-code?
- What is the purpose of the equals method?
- If two objects have the same hash code, how are these stored in hash buckets?
- If two objects have different hash codes, do you still need to implement an equals method?
- If there are two objects with the same hash code, then how the equals method is used?

HASHTABLE VS HASHMAP

HashTable and HashMap are data structures used to keep the key-value pair. These maintain an array of buckets and each element is added to a bucket based on the hashcode of the key object.

The major difference between these is that the HashMap is non-synchronized. This makes HashMap better for single-threaded applications, as unsynchronised Objects perform better than synchronized ones due to lack of locking overhead. In a multi-threaded application, HashTable should be used.

A HashMap can also be converted to a synchronized collection using the following method:

```
Collections .synchronizedMap( idToNameMao );
```

Questions

- What is the purpose of HashTable and HashMap?
- What is the difference between HashTable and HashMap?
- How do HashTable and HashMap store objects?
- Why HashMap is better for single-threaded applications?
- Why performance of HashMap better than HashTable?
- How can you use HashMap in a multi-threaded environment?

SYNCHRONIZED VS CONCURRENT COLLECTIONS

Both synchronized and concurrent collection classes provide thread safety, but the primary difference between these is in terms of scalability and performance.

Collection Synchronization

- Collection synchronization is achieved by allowing access to the collection only from one thread, at a time.
- Synchronization is achieved by making all the public methods synchronized .
- Any composite operation that invokes multiple methods, needs to be handled with locking the operation at the client-side.
- Examples of synchronized collections are HashTable , HashSet, and synchronized HashMap .

Collection Concurrency

- Collection concurrency is achieved by allowing simultaneous access to the collection from multiple threads.
- ConcurrentHashMap implements very fine-grained locking on collection by partitioning the collection into multiple buckets based on hash-code and using different locks to guard each hash bucket.
- The performance is significantly better because it allows multiple concurrent writers to the collection at a time, without locking the entire collection.

- Examples of concurrent collection are ConcurrentHashMap , CopyOnWriteArrayList and CopyOnWriteHashSet .

For large collections prefer using a concurrent collection like ConcurrentHashMap instead of HashTable as the performance of Concurrent collection will be better due to less locking overhead.

Note that in multi-threaded scenarios, where there are interoperation dependencies, you still need to provide synchronized access to these composite operations on concurrent collections, as depicted below:

```
Object synchObject = new Object();

ConcurrentHashMap < String , Account > map =
      new ConcurrentHashMap<>();

public void updateAccount( String userId){
    synchronized ( synchObject ) {
       Account userAccount = map .get(userId);
       if ( userAccount != null ) {
          // some operation
       }
   }
}
```

Questions

- What is the purpose of Synchronized and Concurrent collections?
- What is collection synchronization?
- What is collection concurrency?
- How do you achieve collection synchronization?
- How do you ensure coordinated collection access when a composite operation needs several methods to complete the task?
- How do you achieve collection concurrency?

- How does ConcurrentHashMap offer multiple threads concurrent access to the same collection?
- Why performance of ConcurrentHashMap is better even though it allows simultaneous writes from multiple threads?
- What would you prefer between the ConcurrentHashMap and the HashTable? Why?

ITERATING OVER COLLECTIONS

There are several ways to iterate over a collection in Java. The following are the most common methods.

Iterable.forEach

The forEach method is available with the collections implementing the Iterable interface, and the forEach method takes a parameter of type Consumer .

```
List< Account > accounts =
        Arrays .asList(
              new Account( 123 ),
              new Account( 456 ));

accounts .forEach(acc -> print(acc));
```

for-each looping

- A for-each loop can be used to loop the map.entrySet(), to get key and value both.

```
for (Map.Entry< String , Account > accountEntry :
        map .entrySet()) {
    print( "UserId - " + accountEntry .getKey() + ", " +
            "Account - " + accountEntry .getValue());
}
```

- A for-each loop can be used to loop the map.keySey() to get keys.

- A for-each loop can be used to loop map.values() to get values.
- While running a for-each loop, collection cannot be modified.
- A for-each loop can only be used to navigate forward.

Iterator

- The Iterator can move in both directions backwards and forward.
- When using the Iterator, you can remove entries during an iteration, which is not possible when you use a for-each loop.
- The for-each loop also uses Iterator internally.

```
Iterator<Map.Entry< String , Account >> accountIterator =
        map .entrySet().iterator();

while ( accountIterator .hasNext()){
   Map.Entry< String , Account > accountEntry =
           accountIterator .next();

      print( "UserId - " + accountEntry .getKey() + ", " +
           "Account - " + accountEntry .getValue());
}
```

Notes

- The for-each loop should be preferred over the for loop, as the for loops can be the source of errors, specifically related to index calculations.
- The Iterator is considered more thread-safe because it throws an exception if the collection changes during iteration.

Data Independent Access

The code should be written in such a way that client code should not be aware of the internal structure used to store the collection. This enables making internal changes without breaking any client code. So to facilitate

data-independent access to the collection, it must be exposed such that the Iterator can be used to iterate through all the elements of the collection.

Questions

- Explain different ways to iterate over collection?
- Can you modify the collection structure while iterating using a for-each loop?
- Can you modify a collection element value while iterating using a for-each loop?
- What are the limitations with the for-each loop, regarding the navigation direction?
- What is different between the for-each loop and the Iterator, regarding the navigation direction?
- Can you modify the collection structure while iterating using an Iterator?
- Can you modify a collection element value while iterating using an Iterator?
- Between the for-each loop and the for loop, which one should you prefer? Why?
- Why Iterators are considered more thread-safe?
- Explain the concept of Data Independent Access of the collection?
- How can you design your class to provide Data Independent Access to the collections used in the class?

FAIL FAST

The Iterator methods are considered fail-fast because Iterator guards against any structural modification made to the. This ensures that any failure is reported quickly rather than application landing into a corrupt state sometime later. In such a scenario, ConcurrentModificationException is thrown. The Iterators from java.util are fail fast.

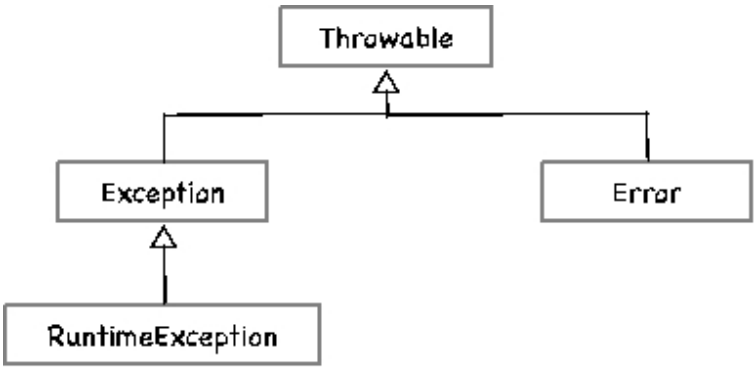
Questions

- What is the term fail-fast used in collection iteration?
- What is the benefit of a fail-fast iterator?
- What is a fail-safe iterator?
    - *The fail-safe iterator doesn't throw any exception when a collection is modified, because fail-safe iterator works on the clone of the original collection. Iterators from java.util.the concurrent package is fail-safe.*

ERROR AND EXCEPTION HANDLING

EXCEPTION

Exception class hierarchy in java



- An exception is an abnormal situation that interrupts the flow of program execution.
- All exceptions inherit from Throwable .
- You subclass Exception class if you want to create a checked exception or RuntimeException if you want to create an unchecked exception.
- Though you can theoretically subclass Throwable class to create checked exception, this not recommended as Throwable is super-class for all Java exceptions and errors.

Questions

- What is an Exception?
- What are the root level exception super classes in Java?
- Which super-class should you sub-class to create checked exception?
- Which super-class should you sub-class to create unchecked exceptions?

CHECKED VS UNCHECKED VS ERROR

Checked Exception

- The checked exceptions are checked at compile time.
- Except for RunTimeException and Error classes, the checked exceptions extend Throwable or its sub-classes.
- The checked exceptions are programmatically recoverable.
- You can handle checked exceptions either by using try/catch block or using the throws clause in the method declaration.
- Static initializers cannot throw checked exceptions.

```
public static void main(String args[]) {
   FileInputStream fis = null ;
   try {
      fis = new FileInputStream("details.txt" );
   } catch (FileNotFoundException fnfe) {
      System .out .println("Missing File :" + fnfe);
   }
}
```

Unchecked Exception

- The unchecked exceptions are checked at runtime.
- Unchecked exceptions extend RuntimeException class.
- You can't fairly hope to recover from those exceptions.
- The unchecked exceptions can be avoided using good programming techniques.
- Throwing unchecked exceptions helps to reveal other shortcomings.

```
public int getAccountBalance(
       String customerName) {
  int balance = 0 ;
   if (customerName == null )
      throw new IllegalArgumentException("Null argument" );
   // logic to return calculate balance
    return balance ;
}
```

Error

- The error classes are used to define irrecoverable and fatal exceptions, which the applications are not supposed to catch.
- Generally programmers cannot do anything to recover from these exceptions.
- Even if you catch OutOfMemoryError , you may get it again because there is a high probability that the Garbage Collector may not be able to free memory.

Use checked exception only for the scenario where failure is expected and there is a very reasonable way to recover from it; for anything else use unchecked exception.

Questions

- What are the checked exceptions?
- What are unchecked exceptions?
- What types of exceptions does the Error class in Java defines?
- How can you handle checked exceptions?
- What happens if an exception is unhandled?
- What are the different ways to handle checked exceptions?
- Which exception classes will you use in the catch block to handle both: checked and unchecked exceptions?
- How do you choose between checked and unchecked exceptions?
- Can you recover from unchecked exceptions?
- How can you avoid unchecked exceptions?
- Can you throw checked exceptions from the static initializer block? Why?

    - *You cannot throw because there is no specific place to catch it and it's called only once. You have to use try/catch to handle checked exceptions .*
- What should you do to handle an Error?

EXCEPTION HANDLING BEST PRACTICES

Even though exception handling is primarily driven by context, but there must be a consistency in the exception handling strategy. Following are few exception handling best practices:

1. Never suppress an exception - as it can lead your program to an unsafe and unstable state.
2. Don't perform excessive exception handling - specifically when you do not know how to completely recover from it.
3. Never swallow an exception - as it may lead the application into an inconsistent state, and even worst, without recording the reason for it.
4. Don't catch and continue program execution - with some default behavior. Default behavior defined today may not be valid in the future.
5. Don't show a generic error message to the user - instead, clean the exception handling code to report user-friendly message with a suggestion about the next step.
6. Don't put more than one exception scenarios in single try-catch - as it will be impossible to ascertain the reason for the exception.
7. Don't catch multiple checked exceptions in a single catch block - as it will be impossible to ascertain the reason for the exception.
8. Don't unnecessarily wrap the exception - which may mask the true source.
9. Don't reveal sensitive information - instead sanitize exceptions generated specifically from the sources that may reveal sensitive information.

10. Always log exception - unless there is a compelling reason not to do so.
11. Don't catch Throwable - as it will be impossible to ascertain the reason for the exception.
12. Don't use exception to control the flow of execution - instead, use a boolean to validate a condition where possible.
13. Handle different scenarios programmatically - instead of putting all coding logic in a try block.
14. Explicitly name the threads - in a multithreaded application, it significantly eases the debugging.
15. Never throw a generic exception - as it will be impossible to ascertain the reason for the exception.

Questions

- Describe some exception handling best practices?
- What are the pitfalls of suppressing an exception?
- What is the problem with showing a generic error message?
- What is the downside of swallowing as an exception?
- What are the pitfalls of handling multiple exceptions in a single catch block?
- What would you do if an exception is thrown from a source that contains sensitive information? What would you log in such a case and what message would you show to the user?
- What is the pitfall of wrapping all the exceptions into a Generic exception class?
- Why shouldn't you use exceptions to control the flow of program execution?
- Should you log all the exceptions? Why?

- Why shouldn't you use Throwable or some other root level class to catch exceptions?
- What criteria should be used to decide which block of code should be included in the try block?
- If a nested call is made that goes through multiple methods, would you be using a try-catch for each method? Why?

TRY-WITH-RESOURCE

try-with-resource is a Java language construct, which makes it easier to automatically close the resources enclosed within the try statement.

```
try (FileInputStream fis =
        new FileInputStream("details.txt" )) {
   // Code to read data
}
```

- The resource used with try-with-resources must inherit the AutoCloseable Interface.
- You can specify multiple resources within a try block.

Questions

- Which java construct can you use to close the system resources automatically?
- Which Interface resources must inherit to use a class object within a try-with-resource construct?

THREADING

THREADING TERMS

- A thread is the smallest piece of executable code within a process.
- A program is a set of ordered operations.
- A process is an instance of a program.
- Context Switch is an expensive process of storing and restoring the state of a thread.
- Parallel processing is the simultaneous execution of the same task on multiple cores.
- Multithreading is the ability of a CPU to execute multiple processes or threads concurrently.
- Deadlock occurs when two threads are waiting for each other to release the lock.

Basic Concepts

- All Java programs begin with the main() method on a user thread.
- The program terminates when there is no user thread left to execute.
- Thread maintains a private stack and series of instructions to execute.
- Thread has a private memory called thread-local storage, which can be used to store thread's current operation-related data, in a multi-threaded environment.
- JVM allows the process to have multiple threads.
- Each thread has a priority.

Questions

- What is a thread?

- What is a program?
- What is a process?
- What is the difference between a thread and a process?
- What is the difference between program and process?
- Explain context switching of thread?
- What is parallel processing?
- What is multi-threading?
- How parallel processing and multithreading related?
- What is deadlock?
- What is a user thread?
- What is thread-local storage? What type of objects should you would store with thread local storage?
- Do threads share stack memory?

THREAD LIFECYCLE

The following are various stages of thread states.

New - Thread is created but not started.

Runnable - Thread is running.

Blocked - Thread waiting to enter the critical section.

Waiting - Thread is waiting by calling wait() or join().

Time-waiting - Thread waiting by calling wait() or join() with a specified timeout.

Terminated - Thread has completed its task and exited.

Notes

You can get the state of thread using the getState() method on the thread.

Questions

- Describe the different stages of the thread lifecycle?
- What is the difference between the blocked state and the waiting state?
- How can you find a thread state?
- How thread sleep is different from thread wait?

THREAD TERMINATION

The thread should be stopped calling interrupt() method. Calling interrupt on a thread even breaks it out of Thread.sleep() state.

The long operation executing on the thread should recurrently call isInterrupted() method to check whether the thread is requested to be stopped, where you can safely terminate the current operation and perform any required cleanup.

```
if (Thread .currentThread().isInterrupted()) {
// cleanup and stop execution
//, for example, a break in a loop
}
```

Notes

The stop() , suspend(), and resume() methods are deprecated, as using these may lead the program to an inconsistent state.

Questions

- Define a good strategy to terminate a thread?
- What is thread interrupt? How thread's interrupt method is different from the thread stop method?
- What happens if a thread is sleeping and you call interrupt on the thread?
- Does calling interrupt stop the thread immediately?

IMPLEMENTING RUNNABLE VS EXTENDING THREAD

Thread instantiated implementing Runnable Interface

```
public class RunnableDerived
        implements Runnable {
    public void run() {
    }
}
```

Thread created extending Thread class

```
public class ThreadDerived
        extends Thread {
    public void run() {
    }
}
```

- Runnable's run() method does not construct a new thread but runs as a normal process in the same thread on which it was created, whereas the Thread's start() methods construct a new thread.
- Runnable is the preferred way to create a thread unless you are specializing Thread class, which is unlikely in most of the case.
- By Implementing Runnable you are providing a specialized class an additional ability to run too.
- Also by separating the task as Runnable you can execute the task using different means.

Questions

- What are the different ways to create a thread?
- How implementing the Runnable interface is different from extending Thread class?
- When should you extend the Thread class?
- When should you inherit Runnable Interface?
- Which one is the preferred way between Runnable and Thread?
- Does implement Runnable creates a thread?
- Both Thread class and Runnable Interface have run methods, what is the difference?

RUNNABLE VS CALLABLE INTERFACE

Runnable Interface

```
@FunctionalInterface
public interface Runnable {
   public abstract void run();
}
```

Callable Interface

```
@FunctionalInterface
public interface Callable<V > {
   V call() throws Exception ;
}
```

- A Runnable cannot return a result and cannot throw a checked exception.
- A Callable needs to implement a call() method while a Runnable needs to implement the run() method.
- A Callable can be used with ExecutorService methods but a Runnable cannot be.

Questions

- What is the difference between the Callable and Runnable interface?
- What is the benefit of using Runnable over Callable?
- Can you throw checked exceptions from the Runnable interface?

- Why Runnable and Callable interfaces called Functional Interface?
- Which Interface returns the result: Runnable or Callable?

DAEMON THREAD

A daemon thread i s a thread, which allows JVM to exit as soon as the program. As soon as JVM halts, all the daemon threads exist without unwinding stack or letting any finally blocks to execute. Daemon thread executes on very low priority.

Threads inherit the daemon status or parent thread, which means that any thread that is created by the main thread will be a non-daemon thread.

Generally, the daemon threads are used to support background tasks or services for the application. Garbage Collection happens on Daemon thread.

You can create a daemon thread like the following:

```
Thread thread = new Thread();
thread .setDaemon(true );
```

All non-daemon threads are called user threads. Those threads stop the JVM from closing.

The process terminates when there are no more user threads. The Java main thread is always a user thread.

Questions

- What is a daemon thread?
- What is user thread? What is the main thread?
- Can a program exit if daemon thread is still running?
- Can a program exit if the user thread is still running?

- Does a finally block on a Daemon thread still get a chance to execute When THE JVM halts exiting all the Daemon thread?
- When a new thread is created; is it created as a user thread or a Daemon thread?
- What happens if no user thread is running but a Daemon thread is still running?
- What is Java's main thread: a user thread or a Daemon thread?
- What type of thread is created when you create a new thread on a main thread: daemon or user?
- What kind of work you can use Daemon thread for?

RACE CONDITION AND IMMUTABLE OBJECT

A race condition arises when multiple threads access shared data simultaneously to modify it. As the order of data read and write by these threads cannot be predicted, this can lead to an unpredictable data value.

An object is considered immutable when there is no possibility of its state change after the construction. If an object is immutable, it can be shared across multiple threads without worrying about the race conditions .

To make an object Immutable

- Declare the class final .
- Allow only constructor to create an object. Don't provide field setter .
- Mark all the field private .

Questions

- How immutable objects help to prevent race conditions?
- Why is race condition capable of generating unpredictable results?
- Why immutable objects are considered safe in a multi-threaded environment?
- Why it's suggested to declare an immutable class as final?
- Why constructor should be the only way to construct an immutable object? What happens if setters are provided?

THREAD POOL

Thread Pool is a set of a number of worker threads that exist separately from the Runnable and Callable tasks.

A fixed thread Pool reduces the overhead of thread creation. It helps the application to degrade gracefully when there is a surge in request that go beyond its capacity to process, by preventing an application from entering into a hanging state or crashing.

Thread Pool also enables a loosely coupled design by decoupling the creation and execution of tasks.

Creating a fixed thread pool is easy with the help of Executors class, where you can use the newFixedThreadPool() factory method to create ExecutorService to execute tasks.

Questions

- What is a thread pool?
- How a thread pool reduces the overhead of thread creation?
- How does a thread pool help to prevent an application from hanging or crashing?
- What is a fixed thread pool and how is it created?
- How does the thread pool enables loosely coupled design?

SYNCHRONIZATION

CONCURRENT VS PARALLEL VS ASYNCHRONOUS

Parallel processing is the simultaneous execution of the same task on multiple cores.

Concurrent processing is the simultaneous execution of multiple tasks; either on multiple cores or by a pre-emptively time-shared thread on the processor.

Asynchronous processing is independent execution of a process, without waiting for a return value from intermediate operations.

Questions

- Explain concurrent processing?
- Explain parallel processing?
- Explain asynchronous processing?
- Explain the difference between concurrent and parallel processing?
- Does parallel processing require multiple threads?
- When an application is concurrent but not parallel?
    - *When application processes multiple operations simultaneously without dividing these operations further into smaller tasks.*

- When an application is parallel but not concurrent?
    - *When application processes one operation dividing the operation into smaller tasks that are processed in parallel.*
- When an application is neither concurrent, not parallel?
    - *When application processes only one operation without dividing the operation into smaller tasks.*
- When an application is both concurrent and parallel?
    - *When application processes multiple operations simultaneously and also dividing these operations further into smaller tasks that are processed in parallel.*

THREAD SYNCHRONIZATION

A race condition occurs when multiple threads concurrently access shared data to modify it. As it is not possible to predict the order of data read and write by these threads, it may lead to an unpredictable data value.

Critical Section is the block of code that if accessed concurrently, by more than one thread, may have undesirable effects on the outcome.

Thread Synchronization is controlling access to the critical sections to prevent undesirable effects in the program.

Synchronization establishes a memory buffer, known as happen-before, which ensures that any other thread that subsequently acquires the same local objects may access all changes made by the thread to the local objects in the critical sections.

Questions

- Explain race condition in multi-threading?
- What is a critical section?
- What is thread synchronization?
- What is a memory barrier?
- What is the concept of happen-before in thread synchronization?

SYNCHRONIZED METHOD VS SYNCHRONIZED BLOCK

The synchronized keyword is used to mark a critical section in the code. Mutual exclusion synchronization is achieved using locking the critical section using the synchronized keyword.

This can be done in the following two ways.

Marking a method as a critical section

```
public class DatabaseWrapper {
   Object reference = new Object();
    // Method marked as critical section
   public synchronized void writeX() {
    // code goes here
    }
    // Method marked as critical section
   public static synchronized void writeY() {
   //  code goes here
    }
}
```

Marking block of code as a critical section

```
public void writeToDatabase() {
   // multiple threads can reach here
    // Code marked a critical section
    synchronized (this ) {
    // only one thread can
      // execute here at a time
    }
   // multiple threads can execute here
}
```

Notes

- Minimize the scope of locking to just a critical section . This will improve overall performance and minimize chances for encountering a race condition.
- Prefer synchronized block over synchronized method, as block locks only on a local object as opposed to an entire class object.
- From within synchronized , never call a method provided by the client code or the one that is designed for inheritance.

Questions

- How is the synchronized keyword used?
- What is the difference between synchronized methods and synchronized block?
- Why synchronized block is preferred to synchronized method?
- Does the synchronized method lock the entire object?
- What happens if you call a method supported by the client from within a synchronized block or process?

CONDITIONAL SYNCHRONIZATION

Conditional synchronization is achieved using conditional variable along with wait() and notify() or notifyAll() methods.

```
// conditional synchronization
public void operation()
      throws InterruptedException {
   synchronized (reference ) {
      if (condition1 ) {
      // wait for notification
          reference .wait();
      }
      if (condition2 ) {
      // Notify all waiting threads
          reference .notifyAll();
      }
   }
}
```

1. There are two methods to signal the waiting thread(s).
    a. notify() - signals only one random thread.
    b. notifyAll() - signals all threads in a wait state.
2. The wait() method has an overload to pass timeout duration too, wait(long timeout) .
3. Between notify() and notifyAll() method, prefer using notifyAll() as it notifies all the waiting threads.
4. The notify() method wakes a single thread, and if multiple threads are waiting to be notified, then the choice of thread is arbitrary.

Questions

- What is conditional synchronization?
- What is the propose of the wait call?
- What is the difference between notify and notifyAll method?
- With multiple threads waiting for the notification which one will be notified when you call to notify?
- Which one should you prefer between notify and notifyAll ? Why?

VOLATILE

In a multi-threaded application, every thread maintains a copy of the variable from the main memory to its CPU cache. So any change made by a thread to the variable in its CPU cache will not be visible to other threads.

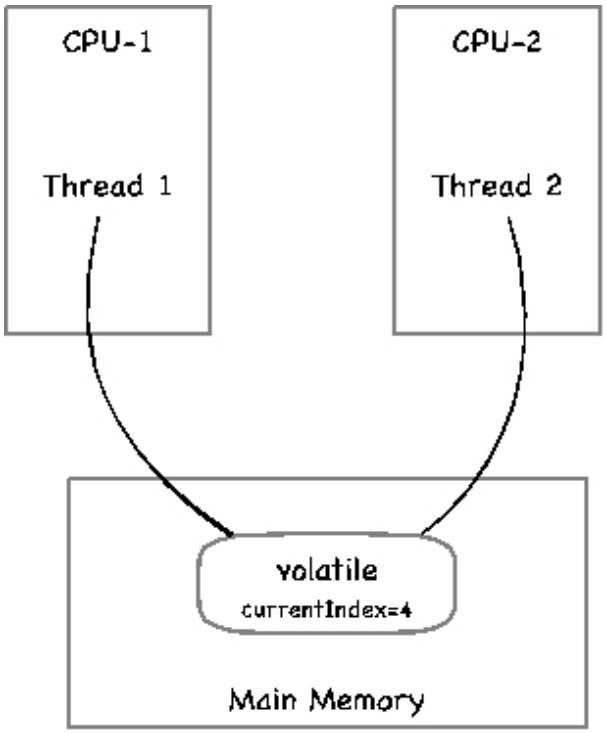
A field marked volatile is stored and read directly from the main memory. As volatile fields are stored in main memory, all the threads have visibility to the most updated copy of the volatile field's value, irrespective of the thread modified it.

Consider a class Ledger, which has a member currentIndex to keep track of the number of entries made. In a multi-threaded environment, each thread will increment currentIndex value independently.

```
public class Ledger {
   public int currentIndex = 0 ;
}
```

If we mark currentIndex as Volatile , then each thread will use its value from the main memory and will not create a copy of it.

```
public class Ledger {
   public volatile int currentIndex = 0 ;
}
```



Questions

- What is a volatile variable?
- Explain the problem that volatile field solves?
- Where are the volatile variables stored?

STATIC VS VOLATILE VS SYNCHRONIZED

static Variable

The static variables are used in the context of class objects where there in only one copy of a static variable exists, regardless of how many objects of the class are created.

But if multiple threads are accessing the same variable, each thread will make a copy of that variable in its CPU cache and change made by a thread will not be visible to other threads.

volatile Variable

volatile variables are used in the context of threads, where only one variable exists regardless of how many threads or objects accessing it and everyone always gets the most recently updated value. Volatile forces all the reads and writes to happen directly in the main memory and not in CPU caches.

synchronized

Both static and volatile are field modifiers dealing with memory visibility related to variables; whereas, synchronized deals with controlling access to a critical section in code using a monitor, thus preventing concurrent access to a section of code.

Questions

- What are the static variables?

- All the objects of a class share static variables. But in a multi-threaded environment; why a change made by one object to the static variable is not visible to an object on another thread?
- If both static and volatile variables are shared between objects, then what's the problem a volatile variable solves?
- How are volatile and static variables different from the synchronized, since even the synchronized monitor guards memory objects too?

THREADLOCAL STORAGE

Each thread has a private memory called thread-local storage , which can be used to store thread's current operation-related data. Usually, the ThreadLocal variables are implemented as private static fields and are used to store information like Transaction Identifier, User Identifiers, etc.

ThreadLocal declaration

```
ThreadLocal <String > threadLocal =
      new ThreadLocal<String >();
```

Setting thread local value

```
threadLocal .set("Account id value" );
```

Getting thread-local value

```
String accountId = threadLocal .get();
```

- As ThreadLocal objects are contained within a thread, you don't have to worry about synchronizing the access to that object.
- ThreadLocal object existence is linked to the thread for which it was created, unless there are other variables which reference the same object too.
- To prevent leak, it's a good practice to remove ThreadLocal object using remove() method.

```
threadLocal .remove();
```

Questions

- What is thread-local storage?

- What are the things you should store in Thread Local Storage?
- Why shouldn’t you synchronize the access to the objects that are stored in ThreadLocal Storage?
- Why should you remove object from Thread Local Storage?

WAIT() VS SLEEP()

wait()

Conditional Synchronization with wait().

```
public void manageWaitFor(int timeInMs)
       throws InterruptedException {
   synchronized (reference ) {
      if (condition1 ) {
         // wait for notification
          reference .wait(timeInMs);
      }
   }
}
```

sleep()

Thread sleeping for a specified interval.

```
public void manageSleepFor(int timeInMs)
       throws InterruptedException {
   //Pause for timeInMs milliseconds
    Thread .sleep(timeInMs);
   //Print a message
    print("Slept for :" + timeInMs + "ms." );
}
```

- The wait method is called on the object's monitor; whereas, sleep is called on a thread .
- Waiting objects can be notified; whereas, a sleeping thread cannot.
- A sleeping thread cannot release a lock; whereas, awaiting object can.
- To wake a sleeping thread you need a reference of it, which is not needed for a waiting object.

Questions

- What is the difference between wait and sleep?
- Why it's possible to notify the waiting object but not the sleeping?
- As you need handle to a sleeping thread to wake it; do you need direct access to a waiting object too? Why?
- What is the mechanism to signal an object to come out of wait?

JOINING THREADS

Threads are usually joined when there is a dependency between the threads. The join() method of the target thread is used to suspend the current thread. In such situations a current thread cannot proceed, until the target thread on which it depends on, has finished execution.

```
// main thread joined with the thread
public void main(String[] args) {
   Thread thread1 = new Thread(
         ThreadMethodRef::
               threadMethod);
    thread1 .start();
    // current thread waits
    // until thread1 completes
      // execution
    thread1 .join();
}
```

Questions

- What is the purpose of the thread's join method?
- Why do you need to join two threads?

ATOMIC CLASSES

Atomic classes provide the ability to perform atomic operations on the primitive types, such that only one thread is allowed to change the value until the method call completes. Atomic classes like AtomicInteger and AtomicLong wraps the corresponding primitive types. There is one present for reference type too, AtomicReference .

There is no need to provide synchronized access to Atomic Class objects. Method incrementAndGet() is AtomicInteger is often used in place of pre and post-increment operators.

Questions

- What are the atomic classes?
- Why don't you need to synchronize access to an object of the atomic class?
- Why pre and post-increment operator are not thread-safe?
    - *Pre and post-operation are multiple operations under the hood- read, increment, and write. All three are not synchronized together, so any thread context switch that happens in between will result in undesired results.*
- What is the difference between Atomic and Volatile variables?
    - *Atomic variables provide atomic access even for the compound operation like pre and post-increment operation, which is not possible if the variable is declared as Volatile. Volatile just guarantees happen-before reads.*

LOCK

- Locking is a mechanism for controlling shared resource access in a multithreaded system.
- ReentrantLock class implements a lock interface .
- A lock can be acquired and released in different blocks of code.
- Lock interface has method tryLock() to verify resource availability.
- As a good practice, the acquired lock must be released in the finally block.

```
// Thread-safe class
public class SafeAccount {
    // Create lock object
    private Lock lockObject =
        new ReentrantLock();
    public void addMoney() {
        // Acquire the lock
        lockObject .lock();
        try {
            // add some money logic here
        } finally {
            // Release the lock
            lockObject .unlock();
        }
    }
}
```

Questions

- Explain the locking mechanism in a multi-threaded environment?

- Do you need to acquire and release lock in the same block of code?
- Why should you prefer using tryLock() instead of a lock()?

ReadWriteLock

- ReadWriteLock maintains a pair of associated locks, one for writing and the other for read-only operations.
- Only one thread can acquire a write lock, but multiple threads can have read lock.
- ReadWriterLock interface is implemented by ReentrantReadWriteLock .

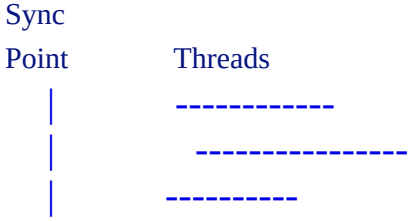
```
// ReentrantReadWriteLock lock
ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
// read lock
Lock rl = rwl .readLock();
// write lock
Lock wl = rwl .writeLock();
```

Questions

- What are the benefits of using ReadWriteLock?
- In which scenarios should you prefer ReadWriteLock over other locking mechanism?

SYNCHRONISERS

Synchronizers synchronize multiple threads to protect a Critical Section .



Synchronizer Types

- Barriers
- Semaphore
- Phasers
- Exchangers
- Latch

Questions

- What is the purpose of synchronizers?
- What are the different types of synchronizers available in Java?

Barriers

- In Barriers , a set of threads waits for each other to arrive at the barrier point before moving ahead.
- CyclicBarrier is a concrete implementation of the Barrier synchronizer.

```
// barrier with five threads
CyclicBarrier barrier =
        new CyclicBarrier( 5 );
    }
}
```

- A Barrier is also called a cycle because it can be reused after calling reset() on it.
- Action can be passed to the CyclicBarrier to execute when all the threads reach barrier point.

```
// barrier with an action to run
// at the barrier point.
CyclicBarrier barrier =
        new CyclicBarrier( 5 , () -> {
// barrier point action code.
});
```

- If any of the thread is terminated prematurely then all the other threads waiting at the barrier point will also exit.
- Barriers are generally used when you divide an operation into multiple tasks running on separate threads, and wait for all the tasks to complete before moving ahead.

Questions

- Explain Barrier synchronizer?
- Can you reuse the same Barrier object again? How?

- What if one of the threads, which other threads await a Barrier, dies?
- What job do you use Barrier for?

Semaphore

- Semaphore maintains a specified number of permits to access a Critical Section .

```
// Semaphore created with four permit
// for four threads
Semaphore semaphore =
        new Semaphore( 4 );
```

- To gain a permit, use the acquire() method. Each call to the acquire() method is blocked until the permit becomes available.
- To release the permit, use the release() method.
- A permit can be released by a different thread, other than the one that acquired it.
- If release() is called more times than acquire() , then for each such additional release, an additional permit will be added.
- If you wish to acquire a mutually exclusive lock, initialize the Semaphore with only one permit.
- Semaphore is generally used to allow limited access to an expensive resource.

Questions

- Explain Semaphore synchronizer?
- What happens if you call release() more time than acquire()?
- What happens when you call acquire, but permits are not available?
- How can you acquire a mutually exclusive lock using Semaphore?
- Where do you use Semaphore?

Phaser

- Unlike other barriers, the number of parties registered with the Phaser can dynamically change over time .

```
// Phaser with four registered parties
Phaser phaser =
    new Phaser( 4 );
```

- A phaser can also be reused again.
- Use a register() method to register a party.
- When the final party for a given phase arrives, an optional action can be performed, and then the Phaser advances to the next phase.
- Use the arriveAndAwaitAdvance() method to wait for all parties to arrive before proceeding to the next phase.
- Phasers monitor the count of registered, arrived, and un-arrived parties. Even a caller who is not a registered party can monitor these counts on a Phaser.
- A party can be de-registered using the arriveAndDeregister() method, from moving to the next phase.

Questions

- Explain Phasor synchronizer?
- What is special about Phasor regarding the number of parties that can register with it?
- Can you reuse the same Phaser object again?
- Can you monitor the count of Phasor monitor using some external object who is not registered as a party with the Phaser?
- What is the difference between Semaphore and Phaser?

Exchanger

- The exchanger lets two threads wait for each other at a Synchronization point to swap elements.

```
/ / Exchanging array of strings
Exchanger<ArrayList<String>>
        exchanger =
        new Exchanger <ArrayList<String>>();
```

- Exchangers use exchange() method to exchange information.

```
// exchanger exchanging data
objectToExchange =
            exchanger .
                exchange( objectToExchange );
```

- On exchange() , the consumer empties the object to be exchanged and waits for the producer to exchange it with full object again.

Questions

- Explain Exchanger synchronizer?
- How many threads are required with the Exchanger object?
- What is the primary purpose of Exchanger synchronizer?
- Does Exchanger synchronizer use the same object to exchange every time or a different object can be exchanged?

Latch

- Latch makes the group of threads wait till a set of operations is finished.
- The latch cannot be reused.
- CountDownLatch class provides an implementation for Latch .
- All threads wait calling await() method till countDown() is called as many times the latch counter is set.

```
// Create a countdown latch with
// five counter
CountDownLatch cdl =
        new CountDownLatch( 5 );

// Count down on the latch after
// completion of thread job
cdl .countDown();

// waiting for count down signals
cdl .await();
```

Questions

- Explain Latch synchronizer?
- Can you reuse the same Latch object again; like Barrier and Phaser?
- What is the signaling process when a thread finishes its work?

EXECUTOR FRAMEWORK

The executor framework provides an infrastructure to execute a set of related tasks on a thread.

Executor takes care to manage the following.

- Creating and destroying threads.
- Maintaining an optimal number of threads for a task.
- Parallel and sequential execution of tasks.
- Segregating task submission and task execution.
- Policies are related to controlling task execution.

```
// Executor interface definition
public interface
        Executor {
    void execute ( Runnable command);
}
```

Questions

- Explain the Executor framework?
- What are the different capabilities of the Executor framework?

Executor Service

- ExecutorService inherits from the Executor interface providing the following additional methods.
    - shutdown() - shuts down the executor after submitting the tasks.
    - shutdownNow() - interrupts the current task and discards the pending tasks.
    - submit() - adds tasks to the Executor .
    - awaitTermination() - waits for existing tasks to terminate.

- ExecutorService provides Future objects to track the progress and the status of the executing task.
- All the tasks submitted to the Executor are queued, which are executed by the thread pool threads.

```
// Executor created with five threads in its thread pool
ExecutorService exec =
    Executors.
        newFixedThreadPool( 5 );
```

- To create a thread pool with a single thread, use newSingleThreadExecutor() method.

Handling Results

- The run() method of the Runnable interface cannot return a result or throw an exception.
- Tasks, which can return a result, are instance of a Callable interface.

```
//tasks can return results derived from Callable using the call method
public interface Callable < V > {
   V call() throws Exception;
}
```

- The submit() method returns Future object which helps to track task.

```
// ExecutorService example
public class ExecService {
    public static void main(String[] args)
            throws ExecutionException,
           InterruptedException {
        // Create executor with five threads
           // in its thread pool.
        ExecutorService exec =
              Executors.newFixedThreadPool( 5 );
        // Submit the callable task to the executor
        Future <String> task =
              exec .submit(
                    new Callable <String>() {
                       @Override
                       public String call()
                             throws Exception {
                          //some logic
                          return null ;
                       }
                    });
        // waits for the result
        String result =
              task .get();
        // Shutdown executor
        exec .shutdown();
    }
}
```

- If there is an exception during the task execution, calling get() method on the ExecutorService will throw an instance of ExecutionException.

Scheduling Task

- ScheduledExecutorService can be used to schedule a future task.
- M ethods used to schedule task.
    - schedule(
        task ,
        delayTime ,
        TimeUnit.SECONDS)
    - scheduleAtFixedRate(
        task ,
        delayTime ,
        repeatPeriod ,
        TimeUnit.SECONDS)
    - scheduleWithFixedDelay(
        task ,
        delayTime ,
        fixedDelay ,
        TimeUnit.SECONDS);

ExecutorCompletionService

- ExecutorCompletionService uses Executor to execute the task.
- CompletionService of Executor can be used to get results from multiple tasks.
- ExecutorCompletionService provides concrete implementation for CompletionService.

```
// Create executor with five threads
ExecutorService es =
      Executors.newFixedThreadPool( 5 );

// ExecutorCompletionService returns an object
ExecutorCompletionService<Result> cs =
       new ExecutorCompletionService<>( es );

// submit task to ExecutorCompletionService
cs .submit( longTask );

// get the result of task
Future <Result> completedTask =
```

```
cs .take();
```

Notes

- Always associate context-based names to the threads, it immensely helps in debugging.
- Always exit gracefully, by calling either shutdown() or shutdownNow() based on your use case.
- Configure thread pool for the ExecutorService such that the number of threads configured in the pool are not significantly greater than the number of processors available in the system.
- You should query the host to find the number of processors to configure the thread pool.

```
Runtime.getRuntime().
    availableProcessors();
```

Questions

- Explain ExecutorService?
- How do you track the progress and status of the executing tasks?
- Does Executor service use dedicate threads to executes queued tasks?
- Can you use Runnable object with ExecutorService? Why?
- How do you find if an exception is thrown in the ExecutorService?
- Can you schedule a task to run in the future with the ExecutorService?
- With ExecutorService, how can you get results from the multiple tasks?
- What is the difference between submit() and execute() methods of ExecutorService?

    - *If you use submit(), you can get any exception thrown by calling get() method on Future; whereas, if you use execute(), an exception will go to UncaughtExceptionHandler .*
- How can you exit gracefully from ExecutorService?
- What should be the criteria for configuring thread pool size? How can you set that?

FORK-JOIN

- The fork-Join framework takes advantage of multi-processors and multi-core systems.
- It divides the tasks into sub-tasks to execute in parallel.
- The fork() method spawns a new sub-task from the task.

```
// spawn subtask
subTask .fork();
```

- The join() method lets the task wait for other task to complete.

```
// wait for subtask to complete
subTask .join();
```

- Important classes in Fork-Join

  - ForkJoinPool - thread pool class is used to execute subtasks.
  - ForkJoinTask - manages subtask using fork() and join() methods.
  - RecursiveTask - a task that yields result.
  - RecursiveAction - a task that does not yield results.

- Both RecursiveTask and RecursiveAction provide an abstract compute () method to be implemented by the class, whose object represents the ForkJoin task.

Questions

- Explain the Fork-Join framework in Java?
- How does the Fork-Join framework help to optimize task execution?

- What is the difference between RecursiveTask and RecursiveAction?

REFLECTION

REFLECTION

Reflection is used to examine the code runtime and possibly modify the runtime behavior of an application.

Purpose of reflection

Reflection must be used only for special-purpose problem solving and only when information is not publicly available. The class members are marked private for reasons. Few of the popular usage of reflection in day to day development includes the following:

- Reflection enables modular architecture by investigating code and libraries runtime to plugin classes and components in the application.
- Annotations are read using reflection. JUnit uses reflection to discover methods to set up and test.
- Debuggers use reflection to read private members of the class.
- IDEs use reflection to enumerate class members and probe code.
- Object-relational mappers use reflection to create objects from data.
- Dependency Injection framework like Spring uses reflection to resolve dependencies.

Usage

The code below demonstrates the use of reflection to access the private field of class Account .

```
public class Account {
   private float rate = 10.5f ;
}
```

```
public static void main( String [] args)
        throws IllegalAccessException {

      Account account = new Account();
    Class <? extends Account > refClass =
            account .getClass();

      // get all the field
     Field [] fields =
          refClass .getDeclaredFields();

      for ( Field field : fields ){
       // no more private
      field .setAccessible( true );
      print( field .get( account ));
  }
```

Questions

- What is reflection?
- What are the different scenarios where reflection can be used?
- How do debuggers use reflection?
- How can reflection help in creating modular software architecture?
- Can you access private members using reflection?
- Give some examples, where reflection is used in software development libraries, tools, and frameworks?

DRAWBACKS OF REFLECTION

Reflection must be used only when something is not publicly available and there is a very compelling reason, otherwise, you must review your design.

Drawbacks

- Reflection does not have compile-time checking, any change in member name will break the code.
- Reflection violates encapsulation as it reveals the internal data structures.
- Reflection violates abstraction as it reveals the internal implementation and provides the ability to bypass validations applied to the members.
- Reflection is slow, as it has additional overhead at runtime to resolve the members.

Questions

- When should you use reflection?
- How does reflection violate encapsulation?
- How does reflection violate abstraction?
- Why reflection is considered slow?
- Why reflection code is considered fragile and can break?

DATA INTERCHANGE

JSON

JavaScript Object Notation or JSON has become extremely popular for the data interchange in the last few years. Now it's not just an alternative to XML but is a successor to it. With the evolution of Big Data and the Internet of Things, along with the JSON's ability to be easily parsed to JavaScript objects, it has become a preferred data format for integration with the web. The following are the few primary factors behind this.

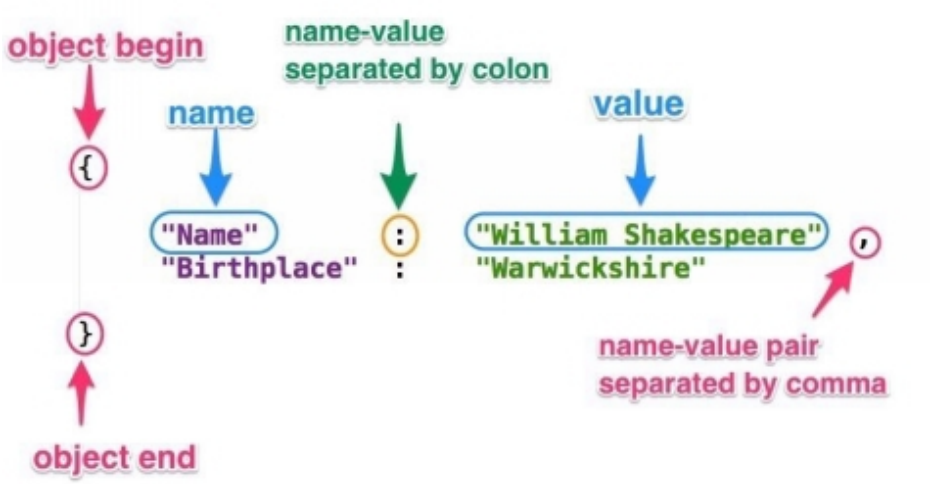
- It's interoperable, as it's restricted to primitive data types only.
- It's lightweight and less verbose than XML.
- It's very easy to serialize and transmit structured data over the network.
- Almost all modern languages support it.
- JavaScript parser in all the popular web browsers supports it.

JSON Structures

JSON structures can be categorized as a JSON Object and JSON array.

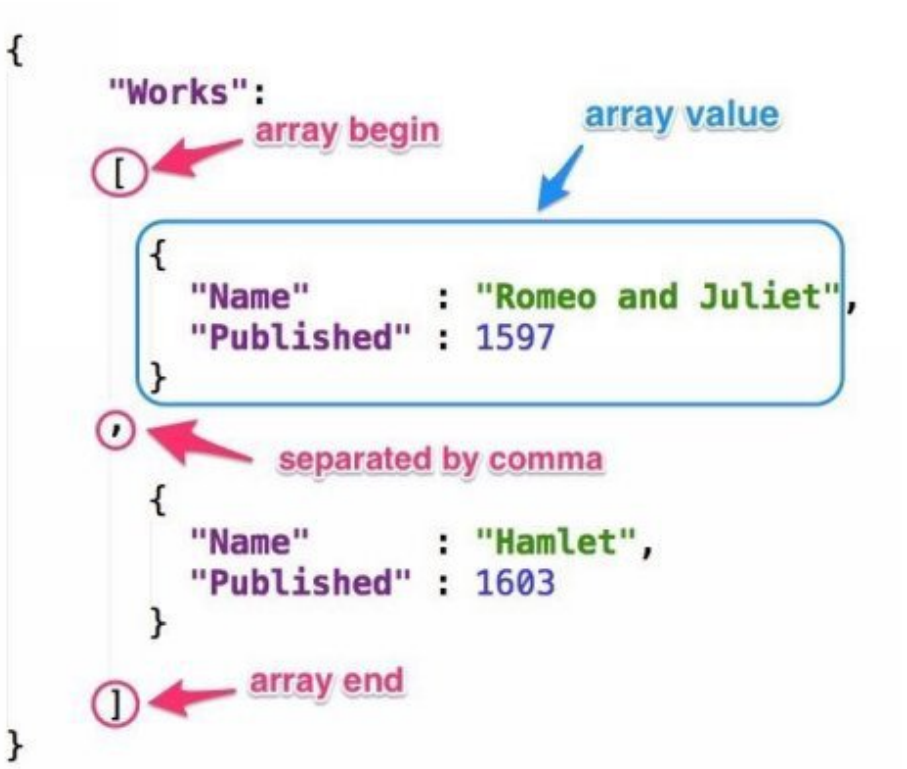
JSON Object

A JSON object is simply a name-value pair separated by a comma. Name is always a string.



JSON Array

JSON Array is an ordered collection of values. Value can be a string, a number, an object, or an array itself.



Questions

- What is JSON?
- What is the main reason for JSON's popularity?
- What are the significant differences between JSON and XML data formats?
- Why JSON has become the preferred format for data interchange?
- Explain the difference between JSON Object and JSON Array?

MEMORY MANAGEMENT

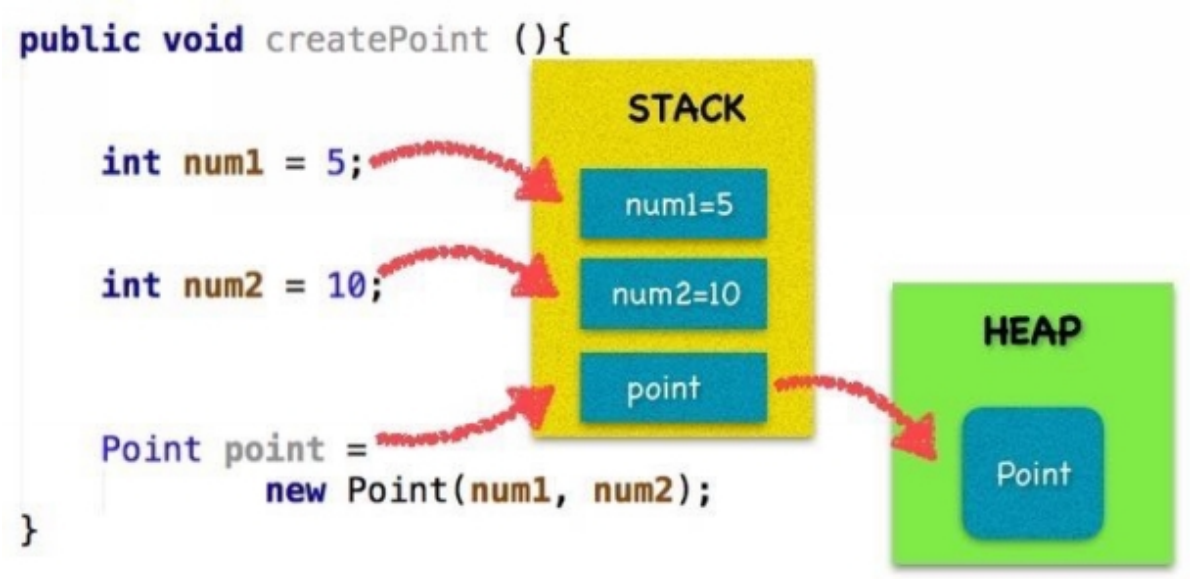
STACK VS HEAP

- A stack is a memory associated with each system thread when it's created; whereas, a heap is shared by all threads in an application.
- For each function, the thread visits, a block of memory is allotted on the top of the stack, for local variables and bookkeeping data, which gets freed when that function returns, in a LIFO order. In contrast, the allocation of memory in Heap is relatively random with no enforced pattern, and variables on the heap are destroyed manually.
- When the thread exits, the stack associated with the thread is reclaimed. When the application process exists, heap memory is reclaimed.
- Allocating and freeing stack memory is simpler and quicker as it is as simple as adjusting pointer. Allocating and freeing memory is comparatively complex in Heap, as there is no fixed pattern of memory allocation.
- The stack memory is visible only to the owner thread, so memory access is straight forward; whereas, heap memory is shared across multiple threads in the application, so synchronization with other thread has performance implications.
- When stack memory is exhausted, JVM throw StackOverFlowError ; whereas, when heap space is exhausted, JVM throws OutOfMemoryError .

Memory allocation in stack and heap

In this example, stack and heap memory allocation is depicted, when a method is invoked.

- num1 and num2 variables are stored in a stack.
- The reference point variable is stored in a stack.
- An object of class Point is stored in heap.



Questions

- What is a stack?
- What is a heap?
- During its lifecycle how is the memory allocated and de-allocated in a stack?
- When does the stack memory gets released?
- When does the heap memory gets released?
- Why memory allocation in a stack is faster compared to the heap?
- Why do you need to apply synchronization to the heap memory and not to the stack memory?
- What exception do you get when stack memory is exhausted?
- What exception do you get when heap memory is exhausted?
- Do you need to explicitly release the objects on the stack?

HEAP FRAGMENTATION

Heap fragmentation happens when a Java application allocates and de-allocated small and large blocks of memory over a period, which leads to lots of small free blocks of memory spread between used blocks of memory. This may lead to a situation when there is no space left to allocate a large block of memory, even though the cumulative size of entire small free blocks is more than the required memory for the large block.

Heap fragmentation causes a long Garbage Collection cycle as JVM is forced to compact the heap. Avoiding allocating a large block of memory, by increasing heap size, etc, can control heap fragmentation .

Questions

- What is heap fragmentation?
- Why does heap fragmentation happen?
- Who has the responsibility to reduce heap fragmentation?
- How can you control heap fragmentation?
- What happens when there is a very high level of heap fragmentation?
- Why does heap fragmentation slow down the application?

OBJECT SERIALIZATION

Converting the content of an in-memory object into bytes to either persist it or to transfer, is called object serialization . These bytes can be converted back to object by de-serializing it.

In Java, an object is serializable if its class implements java.io.Serializable or its subinterface java.io.Externalizable . Members marked as transient are not serialized. ObjectInputStream and ObjectOutputStream are stream classes specifically used to read and write objects.

Questions

- What are serialization and de-serialization?
- Which interface needs to be implemented by the type that want's to support serialization?
- How can you prevent a member from serialization?
- Which classes in Java are used to serialize and de-serialize objects?

GARBAGE COLLECTION

Garbage Collection in Java is the process to identify and remove the unreferenced objects from the memory and also move the remaining objects together to release a contiguous block of memory. When Garbage collection happens, all the running threads in the application are suspended during the collection cycle. Garbage Collector runs on a Daemon thread.

Moving all the surviving objects together reduces the memory fragmentation, thus improving the performance of memory allocation.

During the Garbage Collection cycle, objects are moved to different areas in memory, known as generations , based on their survival age.

System class exposes method gc(), which can be used to request Garbage Collection. When you call System.gc() , JVM does not guarantee to execute garbage collection immediately but may perform when it can. You can also use Runtime.getRuntime().gc() to request Garbage Collection.

Questions

- What is Garbage Collection?
- Explain the Garbage Collection cycle?
- Why Garbage Collection is considered an expensive process?
- Why Garbage Collection cycle improves the performance of an application?
- How does Garbage Collection prevent OutOfMemoryException?
- Which type of thread is used for Garbage Collection: Daemon or user thread?

- What is memory fragmentation?
- Why does the application seem to slow down when Garbage Collection happens?
- Can you explicitly request a Garbage Collection?
- Which methods can you call to request Garbage Collection?
- Does Garbage Collection always happen when requested?

MEMORY MANAGEMENT

JVM memory is divided into two major categories, stack memory, and heap memory.

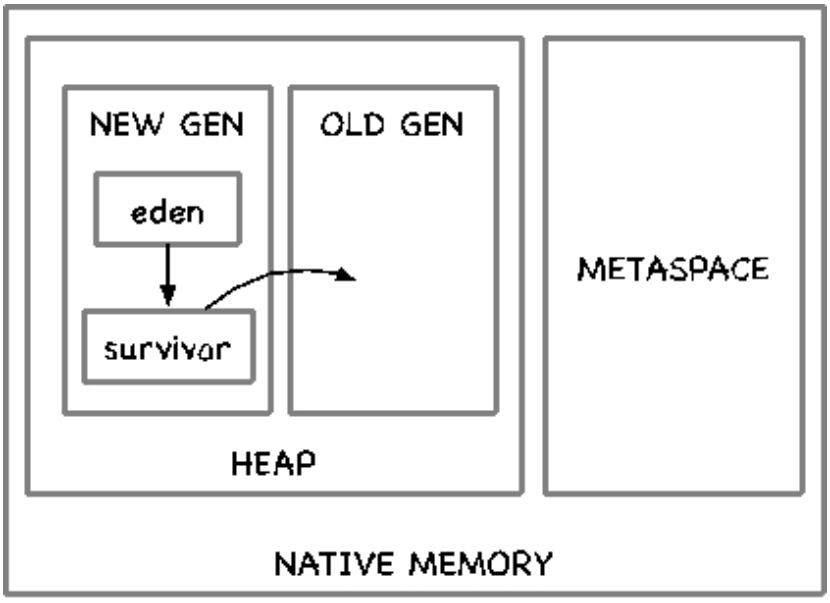
Stack Memory

Stack memory is associated with each system thread and used during the execution of the thread. A stack contains local objects and the reference variables defined in the method; although the referenced objects are stored in heap. Once the execution leaves the method, all the local variables declared within the method are removed from the stack.

Heap Memory

Heap Memory is divided into various regions called generations:

- New Generation - It's divided into Eden and Survivor space. Most of the new objects are created in Eden's memory space.
- Old Generation
- Metaspace



When the new generation is filled, it triggers garbage collection. Objects that survive this GC cycle in Eden are moved to Survivor space. Similarly, after a few cycles of GC, the surviving objects keep moving to the old generation . Metaspace is used by JVM to keep permanent objects, mostly the metadata information of the classes. New generations are more frequently garbage collected than the old generations.

- New Gen and Old Gen are part of heap whereas Metaspace is part of Native memory.
- Metaspace can expand at runtime, as it's part of native memory.

When Garbage collection happens, all the application threads are frozen until GC completes its operation. Garbage collection is typically slow in the old generation; so if lots of Garbage Collection happens in the old generation, it may lead to timeout errors in the application.

Questions

- How is Heap memory divided?
- Explain different generations of heap memory? How objects are moved across generations?
- How is the garbage collection cycle triggered?

- What are Eden and Survivor spaces in New Generation memory?
- When do the objects in Eden move to Survivor space?
- What is Metaspace?
- What is the type of things stored in Metaspace?
- Why Metaspace has virtually unlimited space?
- In which generation Garbage Collection cycle runs slowest? Why?
- Which generation is more frequently garbage collected?

WEAK VS SOFT VS PHANTOM REFERENCE

Generally, if an object is having a reference then the garbage collector will not collect it. This principle is not true for Weak and Soft references.

Weak Reference

A weak reference is a reference that eagerly gets collected by the garbage collector. Weak references are good for caching, which can be reloaded when required.

Soft Reference

Soft reference is slightly stronger than the Weak Reference, as these are collected by garbage collector only when there is a memory constraint. Soft references are generally used for caching re-creatable resources like file handles, etc.

Phantom Reference

Phantom reference is the weakest reference in Java. It is referenced after an object has been finalized, but the memory is yet to be claimed by the Garbage collector. It is primarily used for the technical purpose to track memory usage.

Questions

- What is a weak reference?
- What is a soft reference?
- What is Phantom Reference?

- What are the various reference types that get collected by the Garbage Collector, even when the objects of their types are still in use?
- What is the difference between soft reference and weak reference?
- What is the typical usage of weak reference?
- What is the typical usage of soft reference?
- What is the typical usage of Phantom reference?
- Which is collected first: soft reference or weak reference?

UNIT TESTING

WHY UNIT TESTING?

Two most important reasons

- You can understand the real benefits only by doing it yourself.
- The unit test helps you to sleep well at night.

Other important reasons

- Unit tests provide immediate and continuous feedback on the success/failure of the changes made to the code.
- Units test increases the confidence to make big changes without worrying about breaking any existing feature.
- The unit test helps you to understand the internals of the code and design.
- Unit tests also serve as documentation for various coding scenarios.
- Unit test saves time in the longer run by reducing the multiple cycles of manual verification of different scenarios.

Questions

- What are the benefits of unit testing?
- How does unit testing save time in the long run?
- How can you use unit testing to document the coding scenarios?

UNIT VS INTEGRATION VS REGRESSION VS VALIDATION TESTING

Unit

Unit testing is continuously done while writing the code; to get immediate feedback to the smallest testable change made. Smallest testable units can span across methods or classes but must exclude external dependencies like File I/O, Databases, Network Access, etc.

Integration

Integration testing is done to test end to end integration when all the changes made for a scenario is completely implemented.

Regression

Regression testing is a series of tests performed on entire software to uncover bugs in both functional and non-functional areas. Regression testing is usually done after enhancements, software updates, etc.

Validation

Validation testing is done to verify that after updating or deploying the software, the changes are made as per the requirements.

Questions

- What is unit testing?

- What is integration testing?
- What is regression testing?
- How are the differences between integration testing and regression testing?
- What is validation testing?

TESTING PRIVATE MEMBERS

Private members can be tested using reflection, but it's advisable to do so only when you need to test some legacy code, where changing the visibility of the private method is not allowed.

Usage

The code below demonstrates the use of reflection to access the private field of class Account .

```
public class Account {
   private float rate = 10.5f ;
}


@Test
public void testConcatenate() throws
       IllegalAccessException ,
       NoSuchFieldException {

     Account account = new Account();
    Class <? extends Account > refClass =
          account .getClass();
    Field field =
          refClass .getDeclaredField( "rate" );
    field .setAccessible( true );


     assertEquals( 10.5f , field .get( account ));
}
```

As mentioned, it's not advisable to access private fields using reflection; we have following other options to test code in a private method:

- Test the private method through the public method.
- Change the access modifier of the field, if possible.
- Change the class design.

Questions

- What are the different ways to test private members?
- Can you use Reflection to test private members?
- Why shouldn't you use reflection to test private members?
- What are the various methods to test private members?

MOCKING

The primary responsibility of a unit test is to test the conditional logic in the class code, which should run super-fast for immediate feedback. To enable immediate feedback, a class must have no external dependencies, which is not practical in object-oriented software development, where you need to communicate with external objects to perform File IO, read and write to a database, get data from web service, etc. So the basic idea of mocking to replace these external dependencies with mock objects, to isolate the object under test.

Benefits of mocking

- Mock objects help you to isolate and test only the conditional logic in the class without testing dependencies.
- In the mock object, you can implement partial functionality required for the test without implementing the complete dependency object.
- You don't have to worry about understanding the internal of dependencies, which helps in faster development time.

Questions

- What are mock objects?
- What are the benefits of using mock objects?
- Why do mock objects help to speed up the tests?
- Why should you use mock objects for unit testing?
- What are the types of dependencies that you should replace with mock objects?

JAVA DEVELOPMENT TOOLS

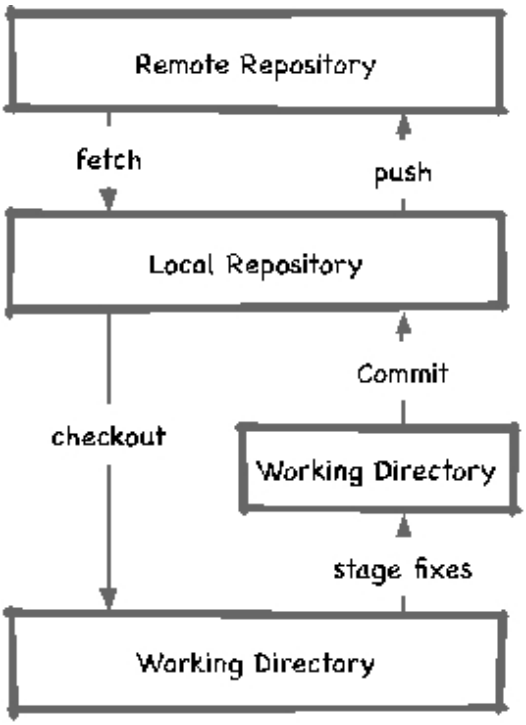
GIT

Git is a source code management system. Its distributed version control system (DVCS) facilitates multiple engineers and teams to work in parallel. Git's primary emphasis is on providing speed with maintaining data integrity.

Git also provides the ability to perform almost all the operations offline, when network is not available. All the changes can be pushed to the server on the network's availability.

Let's discuss a few terms that are frequently used:
Repository - is a directory that contains all the project files.
Clone - it creates a working copy of the local repository.
Branch - is created to encapsulate the development of a new feature or to fix a bug.
HEAD - points to the last commit.
Commit - commits changes to the HEAD and not to the remote repository.
Pull - Gets the changes from the remote repository to the local repository.
Push - Commits the local repository changes to the remote repository.

Git Workflow Structure



Questions

- What is distributed version control systems?
- Why do you use a version control system?
- What are the typical activities you perform with a version control system?
- Explain the branching strategy of code?

MAVEN

Maven is a software project management framework that manages project build, dependency resolution, testing, deployment, reporting, etc. Maven is based on conventions rather than elaborate configurations.

Maven is managed by pom.xml (also known as Project Object Model) file, which facilitates defining various tasks (or goals) and also has a dedicated section to specify various project dependencies, which are resolved by Maven.

Questions

- Why should you need a framework for dependency resolution - you can manually download the files and add a reference?
- What all things does Maven support?
- Why Maven is said to be convention-based?
- What is the meaning of goals in Maven?

JENKINS

Continuous Integration

Continuous Integration is a practice where the engineers, who are working in parallel on the same code repository, merge their changes frequently. Every check-in is followed by a series of automated activities, which aims to validate the changes in the check-in. Typical automated activities following the check-in are:

- Fetching changes from the repository,
- Performing automated build.
- Execute different tests like the unit, integration, validations, etc.
- Deploy the changes.
- Publish the results.

Jenkins

Jenkins is an open-source web-based tool to perform continuous integration. Jenkins provides configuration options to configure and execute all the above-mentioned activities. Available configuration options are: configuring JDK, security, Build Script; integration with Git, Ants, Maven, Gradle, etc.; deployment, etc.

The execution of jobs can be associated with some event or can be based on time-based scheduling.

Questions

- What is Continuous Integration?
- What are the different activities that the Continuous Integration framework should support?

- Explain the capabilities of Jenkins to support Continuous Integration?
- How do you integrate third-party tools with Jenkins?

JAVA INTERVIEW COURSE