JAVA interview notes

Covers 136 topics and 700 technical interview questions

JOLLY

Java Interview Notes

by Jolly

CONTENT

INTRODUCTION

Introduction

Java Interview Notes cover topics that are frequently discussed during Java technical interview round, along with interview questions for each topic. This book also contains lots of code snippets and figures to explain topics.

To be confident in an interview, thorough understanding of software design and language concepts is very important, because during an interview same question can be asked in many different ways.

Interviewers are interested in hearing not just the correct answer, but also your opinion and thoughts about the topic. Your approach towards understanding the questions and sharing your thoughts plays an extremely important role in an interview success.

Confidence to face such interview can only be achieved when you spend really good amount of time coding, reading technical topics and discussing this with peers. *There is no other shortcut available.*

I hope this book helps to prepare you not only for your next interview, but also for your day-to-day software development task.

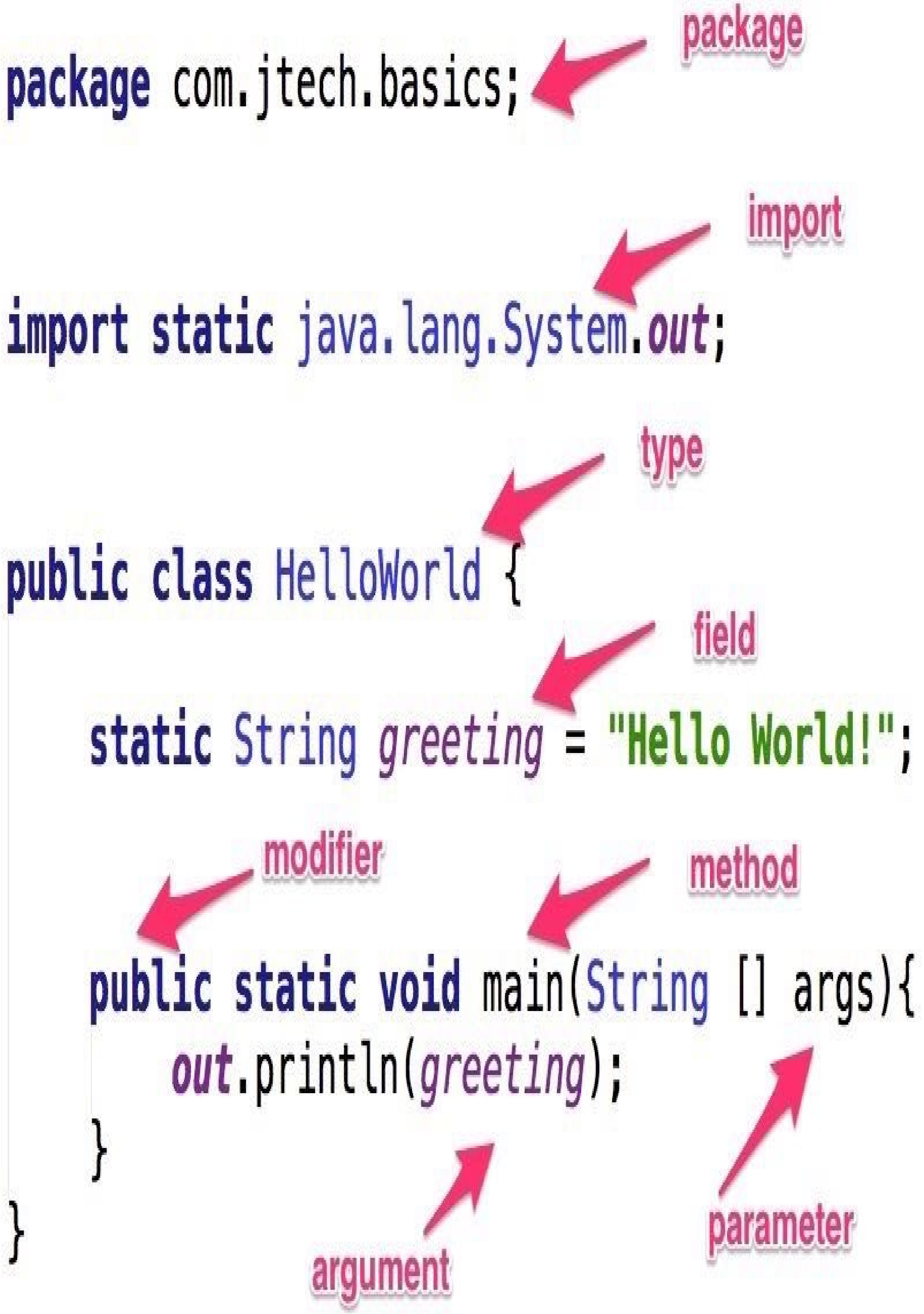
All the best!!!

JAVA

FUNDAMENTALS

Java Program Anatomy

The following code snippet depicts the anatomy of simple Java program.



```
package com.jtech.basics;

import static java.lang.System.out;

public class HelloWorld {

    static String greeting = "Hello World!";

    public static void main(String [] args){
        out.println(greeting);
    }
}
```

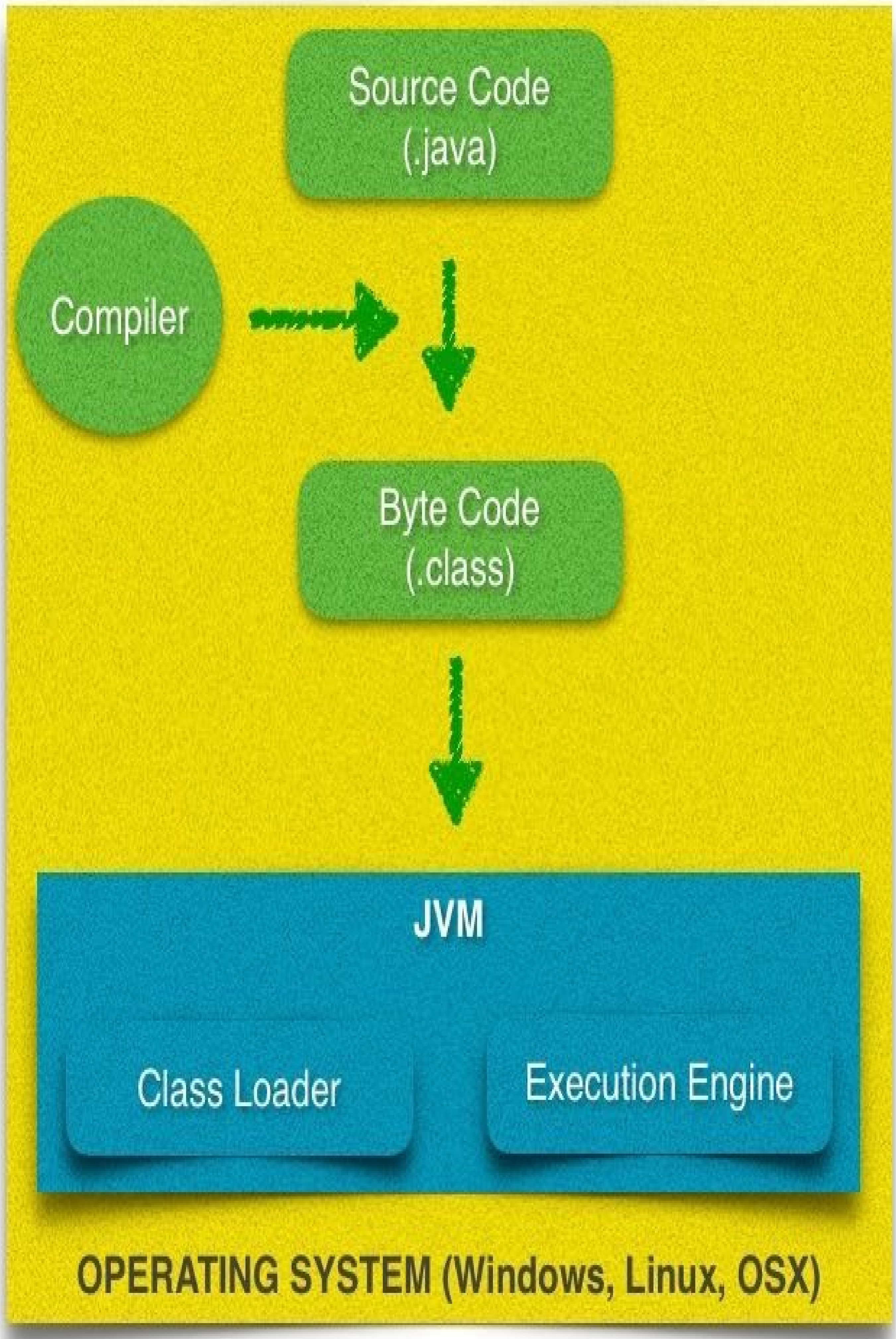
- *Package* - Represents logical grouping of similar types into namespaces. It also prevents naming collision and provides access protection.
- *Import* - Imports the package, so that classes can be used in the code by their unqualified names.
- *Class* - Represents a type template having properties and methods.
- *Field* - Represents a member used for holding values.
- *Method* - Represents an operation/behaviour of the class.
- *Modifier* - Specifies access control level of a class and its members.
- *Parameter* - Specifies the variable declared in the method definition.
- *Argument* - Specifies the data passed to the method parameters.

Questions

- What is package?
- Why do you need to import packages?
- Why do you need to specify access modifiers?
- What is *static import*?
  - *static import enables access to static members of a class without 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


Source Code
(.java)
Compiler
Byte Code
(.class)
JVM
Class Loader
Execution Engine
OPERATING SYSTEM (Windows, Linux, OSX)

1. Java Compiler compiles the Java source code (.java file) into a binary format known as *bytecode* (.class file). Java bytecode is 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. *Class loader* in JVM loads the binary representation of the classes into memory.
4. *Execution engine* in JVM executes the byte code and generates *Operating System* specific machine instructions. These machine 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 machine code?
- What is the difference between bytecode and machine code?
- What is JVM? Is it same or different for different Operating Systems?
- What are the major components of JVM?
- What is the role of class loader in JVM?
- What is the role of Execution Engine in JVM?
- What are machine instructions?

Data Types

Primitive Types

- Primitive types are *byte*, *boolean*, *char*, *short*, *int*, *float*, *long* and *double*.
- Primitive types always have a value; if not assigned, 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 an *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*. A reference type also refers to an object in memory.
- Objects of reference type will have *null* as default value, when it’s unassigned.
- Objects have *variables* and *methods*, which define its state and the behaviour.
- 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 double?
- What happens when you assign a variable of primitive data type to another variable of same type?
- What are reference data types?
- What happens when you assign a variable of reference data type to another variable of same reference type?
- What are the differences between primitive data types and reference data types?
- What are the purposes of variables and methods in a reference type?
- If a variable of reference data type is not assigned, what does it contain?

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 object of any class.

Due to inheritance, all the following *java.lang.Object* class methods, which are not *final* or *private*, 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. *Overriding* class should inherit *Cloneable* interface and implement *clone()* method to define the meaning of copy.
- *toString()* - returns string representation of the object.
- *finalize()* - called by the Garbage Collector to clean up the resources. *java.lang.Object's* implementation of *finalize()* does nothing.

Questions

- What is the base class for all Java classes?
- What are the different methods of *java.lang.Object* class, which are available for overriding in the derived class.
- What happens if your class does not override equals method from the *java.lang.Object* class?
  - *The equals() method in java.lang.Object class compares whether object references are same, and not the content. To compare content, you need to override equals() method.*
- What is the purpose of clone() method?
- Why should the overriding class define the meaning of clone() method?
- What happens if overriding class does not override clone() method?
  - *In case if an object contains references to an external objects, any change made to the referenced object will be visible in the cloned object too.*

Access Modifiers

Access modifiers determine the visibility rules whether other classes can access a variable or invoke a method.

At class level, you can either use *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 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 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 pubic access modifiers?
- What happens when no access modifier is specified with the class?
- If sub-class exists in a different package, does it still has 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 inheritance concept, you should be able to use sub class object with super class reference. This will not be possible if sub class member is declared with more restrictive access modifier.*
- What should be the criteria to decide an access modifier for a 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 *static* keyword is used with the *variables* and the *methods*, it signifies that these members belongs to the *class* and these members are shared by all the objects of the class. Static members does not have a copy and are stored only at a single location is memory. These members should be accessed using class name.

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

final

final Class

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

final Method

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

final Variable

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 - variable which is not initialized at the point of declaration.

Notes

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

Questions

- Explain final class?
- What are the benefits of declaring a class final?
- Explain final method?
- What are the benefits of declaring a method final?
- Explain 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 blank final variable?
- How does declaring a variable as final helps with optimization?

static Initialization Block

- *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.
- *static block* does not have access to the instance members.
- *static block* is called only once for a class.
- A class can define any number of *static blocks*, which gets called in order of their definition in the class.
- You can *only* throw *unchecked* exception from a *static block*.

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

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

Questions

- What is static initialization block?
- What is the primary purpose of the static initialization block? What kind of things should you do in the static block?
- Can you access instance members from static initialization block? Why?
- Does static initialization block gets called every time when 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 for their order of execution?
- Can you throw exception from static initialization block? What type?

finally

The 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 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

Compiler does all in its power to execute the *finally* block, except in the following conditions:

- If *System.exit()* is called.
- If 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 kind of things should you do in a finally block?
- What kind of things should you do in a catch block?
- Does finally block always execute? What are the conditions when the finally block does not execute?

- Should you ever return from the finally block? Why?

finalize()

When the *Garbage Collector* determines that there in no reference to an object exist, it calls *finalize()* on that object; just before removing that object from memory.

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

finalize() could be overridden to release the *resources* like: file handles, database connections, etc.; but you must not rely on *finalize()* method 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() method* only as a backup mechanism for releasing resources. Where possible, use *try-with-resource* construct to automatically release the resources.

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

Questions

- What is finalize method in Java?
- When does the finalize method gets called?
- Who calls the finalize method?
- What kind of things can be done in the finalize method?
- Should you explicitly call finalize method to release resources? Why?
- What are some alternate mechanisms that can be used to release system resources?
- What happens if an unhanded 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 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.
- Hides the internal data structure used to store the value.
- Internal fields can be changed, without requiring changing any user of the code.
- Encapsulates the internal complexity in retrieving or calculating the value.
- Provides ability to specify different access modifiers for getter and setter.
- Provides ability to add debugging information.
- Can be passed around as Lambda expressions.
- Many libraries like mocking, serialization, etc. expects getters/setters for operating on the objects.

Questions

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

varargs vs object array

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

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;
}
```

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

Questions

- What is varargs?
- What are the differences between varargs and object array?
- Can you call a method with zero arguments, which is defined with a varargs as its only 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.
- *Default interface method* can be added to enhance an *Interface* that is not even under your control.
- It does not break any existing implementation of the interface it is added to.
- Implementing class can override the default methods defined in the interface.
- *Default method* is also known as *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 Default method

- If the class inherits multiple interfaces having default methods with same signature, then the implementing class has to provide implementation for that default method to resolve ambiguity.
- 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 couple of minor differences:

- Abstract methods allows 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 enhance 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 behaviour in the classes inheriting the Interface.
- Implementing class cannot override the static methods defined in the interface it is inheriting.
- *Static Interface method* can even be added to enhance an interface which is not under your control.
- Similar to default Interface method, even the static interface method does not break any existing implementation of the interface.

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 are static interface methods?
- 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 static interface method to enhance 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.

Annotation 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, variable, method, parameter, class, interface, enum, package, 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
}
```

Few built-in annotations

- *@Deprecated* - signifies that 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?
- What are the different Java entities where you can apply annotations?

Preferences

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

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

Questions

- What is the use of Preferences class?
- What are the types of information that can be stored with the Preferences?
- While using Preferences class, do you have to handle the internal format required by the Operating System to store the preferences?

Pass by value or Pass by Reference

In Java - method arguments, primitive or object reference, are *always passed by value* and access to an object is allowed only through a reference and *not direct*. 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 reference.

Questions

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

Naming Convention

Camel Case vs Pascal Case

Camel Case is 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 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*.

Naming Convention

Naming convention is a set of rules that govern the naming for the identifiers representing interface, class, method, variables, and other entities. Choice and implementation of naming conventions often becomes matter of debate.

Standard naming convention improves the code readability, which helps in review and overall understanding of the code.

Interface

- Name should be *Pascal Case*.
- Name should be an *adjective* if it defines behaviour, otherwise *noun*.

```
public interface Runnable
```

Class

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

```
public class ArrayList
```

Method

- Name should be *Camel Case*.

```
public boolean isEmpty()
```

Variable

- Name should be *Camel Case*.

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

Constants

- Name should be all *uppercase* letters. Compound words should be separated by underscores.

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

Enum

- Enum set name should be all *uppercase* letters.

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

Acronyms

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

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

Questions

- What is naming convention?
- Why do you need 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

ORIENTED

PROGRAMMING

Polymorphism

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

Polymorphism is primarily achieved by *subclassing* or by *implementing an interface*. The derived classes can have its own unique implementation for certain feature and yet share some of the functionality through inheritance.

Behaviour of object depends specifically on its position in the class hierarchy.

Consider you have a Furniture class, which has addLegs() method; and a Chair and a Table class, both extend Furniture class and have their own implementation of addLegs() method. In this situation, the implementation of addLegs() method that gets called is determined by the runtime, depending whether you have a Chair or a Table 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, it also paves way to have an alternate implementation for *testing*.

Questions

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

Parametric polymorphism

In Java, *Generics* facilitates implementation for *Parametric polymorphism*, which enables using 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 takes *parameter* of type *T* and returns count of wheels, without worrying about what type *T* actually is.

Questions

- What is Parametric Polymorphism?
- How Generics is used to achieve Parametric Polymorphism?
- How are 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.

In the code below, the method *printWheelsCount()* takes *Vehicle* as parameter and prints count of wheels in the *Vehicle*. The *main* method shows *subtype polymorphic* calls, passing objects of *Car* and *Bike* as arguments to the *printWheelsCount()* method. Every place where it expects a type as parameter, it also accepts *subclass* of that type as parameter.

```
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());
}
```

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*, runtime takes the decision to call an implementation, as compiler does not know what to bind at compile time.

Rules for method overriding

- Method arguments and its order must be same in the overriding method.
- Overriding method can have same return type or subtype of base class method's return type.
- Access modifier of overridden method cannot be more restrictive than its definition in base class.
- Constructor, *static* and *final* method cannot be overridden.
- Overridden method cannot throw *checked exception* if its definition in base class doesn't, though overridden method can still throw *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 overriding method throw checked exception; when the overridden method in the super class does not? Why?
- What are the benefits of method overriding?

@Override

@Override annotation is way to explicitly declare the intention of *overriding* the method implementation 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 wrong method signature, when you want to *override* from 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 @Override annotation?
- What happens if you define a method with the same signature as defined in the super class and not use @Override annotation?
- What are the benefits of @Override annotation?

Overloading

- Method *overloading* is 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, decision is made to call a specific implementation.

Rules for method overloading

- Method can be overloaded by defining method with the same name as an existing one, having
  - Different number of argument list.
  - Different datatype of arguments.
  - Different order of arguments.
- Return type of the overloaded method can be different.
- Method with the same name and exactly the same parameters cannot be defined, when they differ only by return type.
- *Overloading* method is not required to throw same exception as the method its overloading.

Operator Overloading

Operator overloading is an ability to enhance the definition of 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 same with the difference only in the 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 type and its behaviour. *Abstraction* is all about hiding details about the data, its internal representation and implementation.

The other related object oriented concept is *encapsulation*, which could be used to abstract the complexities and the internal implementation of a class.

Abstraction also helps making the software maintainable, secure and provides an ability to change 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 (know as *sub class*) by inheriting that class. It focuses on establishing *IS-A* relationship between *sub class* and its *super class*. Inheritance is also used as technique to implement polymorphism; when a derived type implements method defined in the base type.

Rules for Inheritance

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

Questions

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

Composition

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 *HAS-A* relationship between classes. So unlike *Inheritance*, which deals with extending features of a class, *composition* reuses a class by composing it. *Composition* is achieved by storing reference of another class as a member.

Inheritance vs Composition

Problem with *inheritance* is that it breaks *encapsulation* as the derived class becomes tightly coupled to the 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 would you prefer composition over inheritance and vice versa?

FUNDAMENTAL DESIGN CONCEPTS

Dependency Injection vs Inversion of Control

Dependency Injection and *Inversion of Control* promotes modular software development by loosely coupling the dependencies. Modular components are also more maintainable and testable.

Inversion of Control

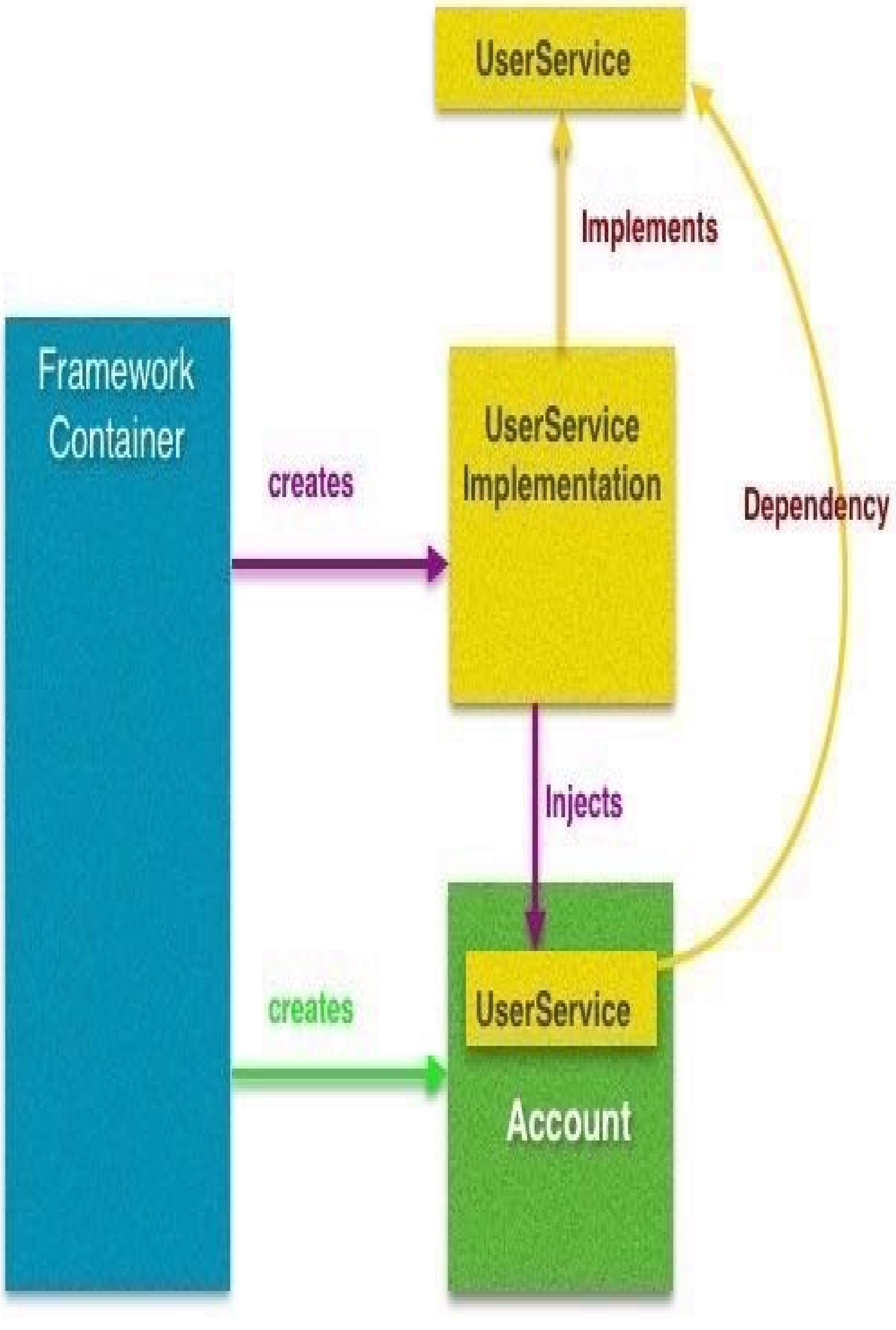
Inversion of Control provides a design paradigm where dependencies are not explicitly created by the objects that requires these; but such objects are created and provided by the external source.

Dependency Injection

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

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

The figure below depicts Dependency Injection concept.


UserService
Implements
Framework Container
creates
UserService Implementation
Dependency
Injects
creates
UserService
Account

The code below demonstrates dependency injection as 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, *DI* and *IoC* provides immense help in unit testing too, by providing an ability to mock dependencies. *Spring framework* is an example of DI container.

Note

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

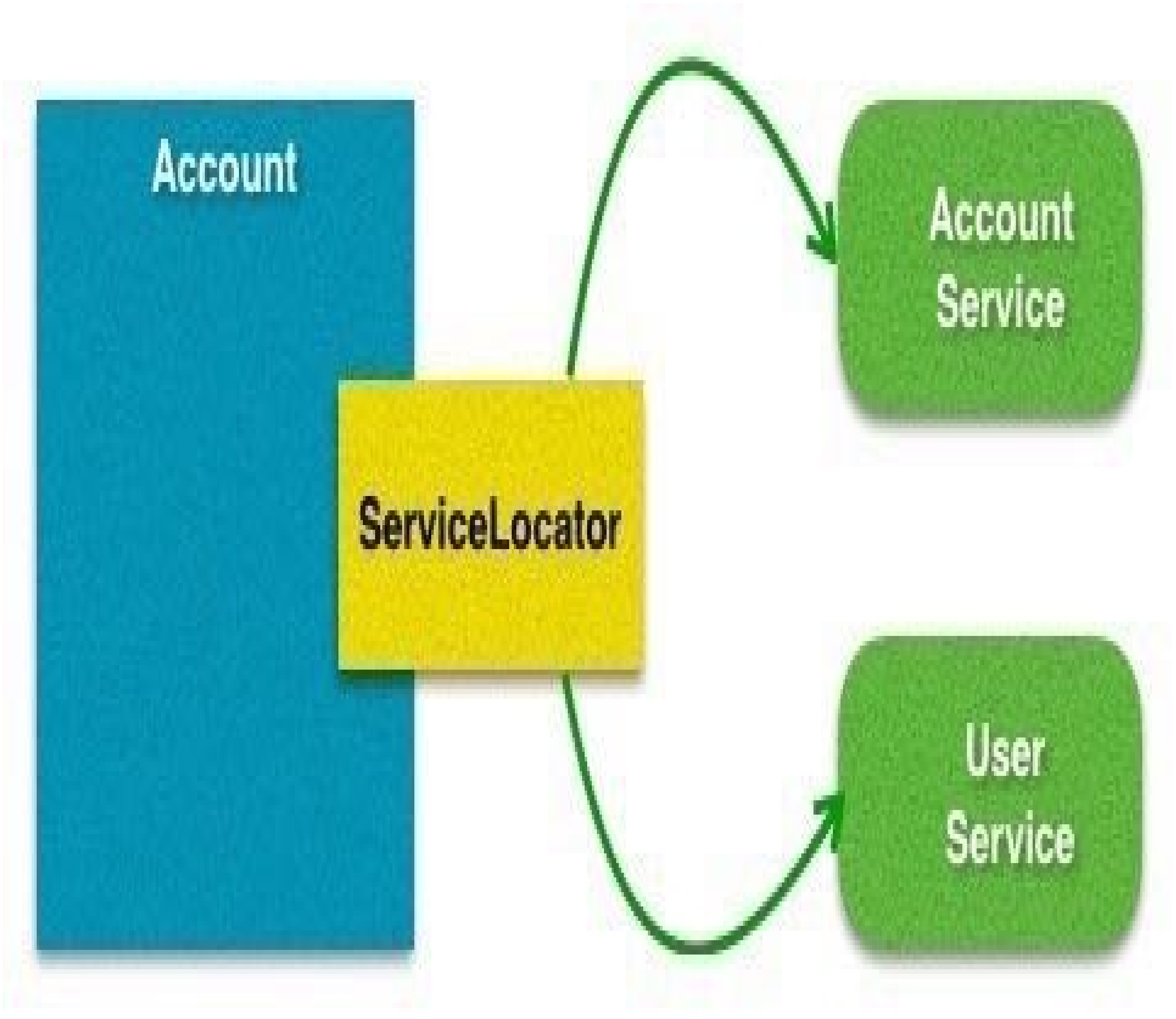
Questions

- What is Inversion of Control?
- What is Dependency Injection?
- What is the difference between 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 can be used in unit testing?
- How Dependency Injection can be used for modular software development?

Service Locator

Service locator is an object that encapsulates the logic to resolve the service requested for. *Service locator* also provides interface to register services with it, which allows you to replace the concrete implementation without modifying the objects that depends 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

- Class does not have to manage any service dependency and its life cycle.
- Testing class in isolation is possible, without the availability of *real* services it depends on.
- Enables runtime resource *optimization*; as services can be registered and unregistered runtime.

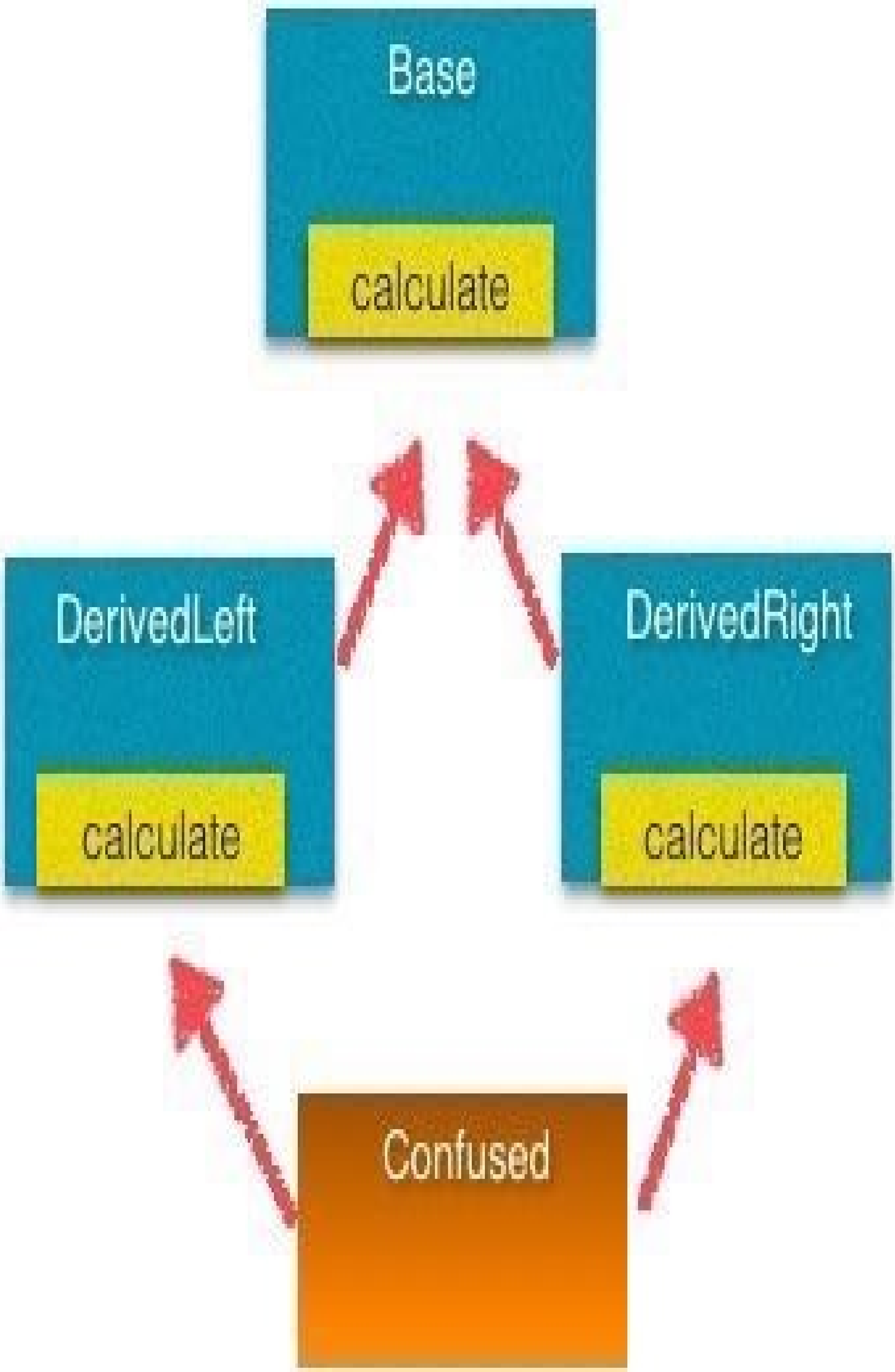
Questions

- Explain Service Locator design pattern?
- What are the benefits of using Service Locator?
- 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 helps in testing?

Diamond Problem

Java doesn’t allows extending multiple classes because of the ambiguity that could arise when more than one super class has method with the same signature, and compiler can’t decide which super class method to use.

Consider the inheritance hierarchy depicted 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.


Base
calculate
DerivedLeft
calculate
DerivedRight
calculate
Confused

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

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

Questions

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

Programming to interface

Programming to interface forms basis for modular software development by facilitating decoupling between software components. 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’s take a very simple example. Suppose we have a method to *sort* a collection, which is defined with Interface *Map* as its parameter. This means, that the *sort()* method is not tied to any specific type of *Map* implementation and you can pass any concrete implementation of the *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 select the most appropriate behaviour, runtime.
- 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 interfaces before doing 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 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 helps in unit testing?

Abstract Class vs Interface

Abstract Class

Abstract class cannot be instantiated but can be extended. You should extend abstract class when you want to enforce a common design and implementation among derived classes.

Interface

Interface is set of related methods, which defines its behaviour and its contract with the outside world. Use interface when you want to define common behaviour among unrelated classes. Interfaces can also be used without methods and are known as *marker interface;* such interfaces are used to categorize the classes. Example of 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

- Abstract class can be updated to add more capabilities to the class whereas Interface can be added to implement new behaviour to the class. Though with introduction of *default interface methods*, even Interfaces can be extended to have more capabilities.
- Interface can be multiple inherited; whereas, abstract class cannot.
- Interfaces can 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.
- Abstract class can have state associated, which is not possible with Interface.
- Abstract class can be extended without breaking the class that extends it; whereas, any change in interface, except made for default and static methods, will break the existing implementation.

Questions

- If an abstract class cannot be instantiated, why would you define a constructor for an Abstract class?
  - *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 class design?
- Define Interface? What role an Interface plays 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

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

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

- Easier to design and implement, 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 object has reduced *Garbage Collection* overhead.

Disadvantages

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

Rule for defining Immutable Objects

- Declare the class *final*.
- Allow only constructor to create 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 final and immutable object?
- How does declaring a variable final helps with optimization?
- Can you list some of the problems with Immutability?
    - It’s harder to define constructors with lots of arguments.
    - Since it’s left to the developer to enforce immutability, even a single setter added accidentally, can break it.

Cloning

Cloning is process of creating 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, exact copy is created and for reference type members, its address is copied. So the reference type members, both original and the newly created, points 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 generally 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 known *NaN*, is 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 resulted, there is no indication unless result is checked.

Signalling NaN - When a signalling NaN is resulted, 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

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

The following code demonstrates 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 for performing bit flag operations?

Comparing the Types

Primitive types can be compared only 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, equality operators == and *!=* are used to compare the addresses of two objects in memory and *not* their actual content.

.equals() Method

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

Questions

- What are the different ways to compare types in Java?
- For reference types, what does the equality operator compares?
- What does the equals method compare?
- When would you prefer using equals method to equality operator?
- What happens if a class does not override 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){
   //
}
```

In the example above, EPSILON is the range used to compare two floats. The value of EPSILON is a very small number like 0.0000001, depending on the desired precision.

Questions

- Why you shouldn’t use equality operator to compare float values?
- What is the preferred way to compare two float values?
- What should be the criteria to define range value 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 equality of two String variables from different sources, though Interned Strings can be compared using equality operators too.

Questions

- Why should you use equals method to compare String objects?
- What is the pitfall of using 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 be copied. So only single instance of enum is available, the one defined with the enum definition.

As only one instance of 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 preferred way to compare two enum values?

Enum vs Public Static Field

Following are advantages of using *enum* over *public static int.*

- Enums are compile time checked, whereas int values are not.
- An *int* value needs to be validated against an expected range; whereas, enums are not.
- Bitwise flag operations are built into enumSet.

The code below demonstrates usage of *enum* and *public static* field. With *public static int*, you can pass any *int* value to *AddVehicle()* method.

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 int parameter as compared to enum parameter?
- Why should you prefer using enum to public static int?

Wrapper Classes

Each *Primitive data type* has a class defined for it, which wraps the primitive datatype into 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 collections too.

Questions

- What are wrapper classes?
- What are the advantages of using wrapper type over 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 dynamic memory allocation and initialization of corresponding Wrapper class object. *Auto unboxing* is automatic conversion of a Wrapper class to 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 is auto boxing and auto unboxing?
- What are the advantages of auto boxing?

BigInteger and BigDecimal

BigInteger and *BigDecimal* are used to handle values which 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("9876543210987654321098765");
```

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

STRINGS

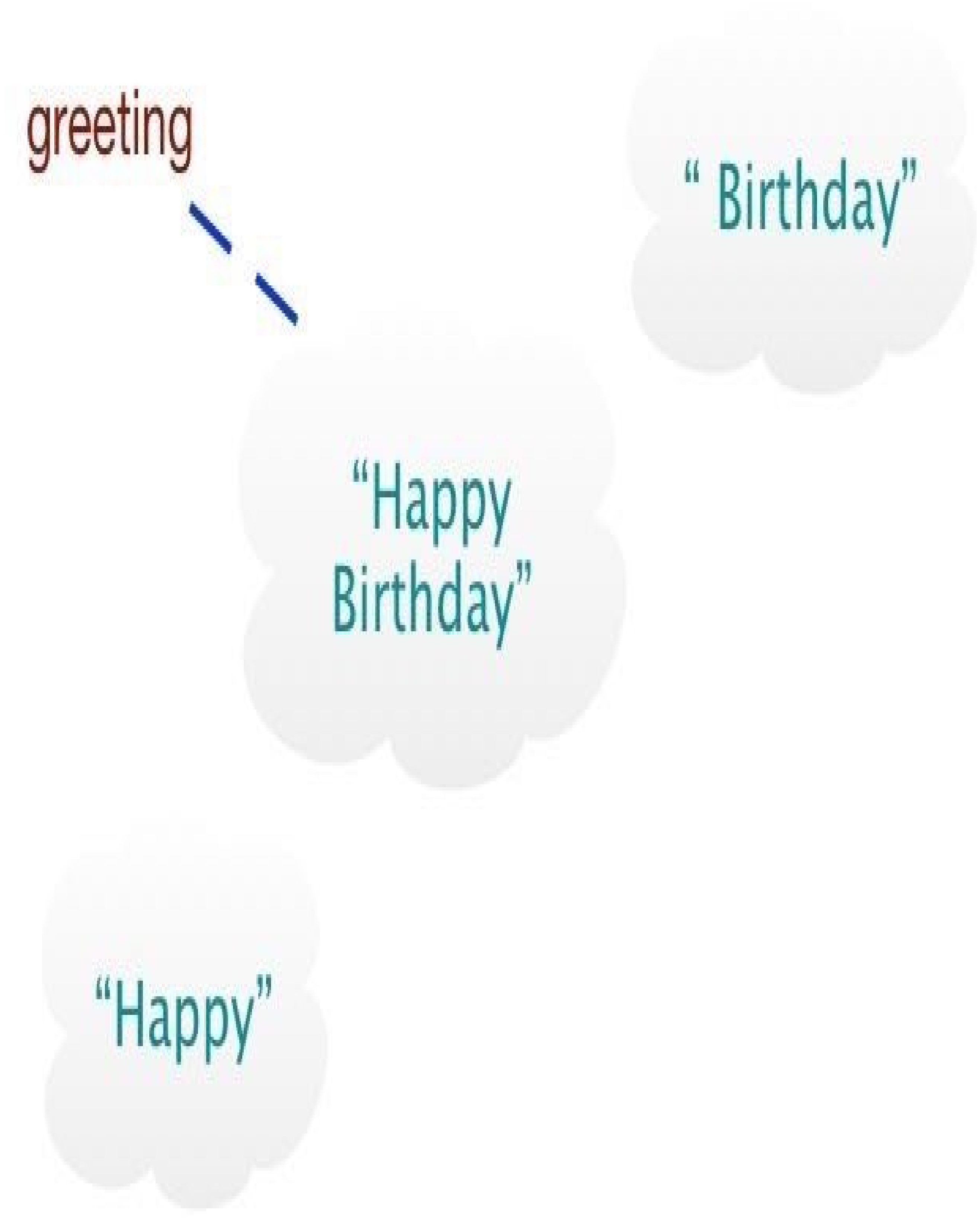
String Immutability

The *String* object is *immutable*, which means once constructed, the object which *String* reference refers to, can never change. Though you can assign 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 but you can change the reference variable that is referring to the object. In the above example, the *String* reference greeting starts referring the String object “Happy Birthday”.
- Note that any operation performed on String results into creation of new String.
- *String* class is marked *final*, so it’s not possible to override immutable behaviour of the *String* class.

Advantages

- As no synchronization is needed for String objects, it's safe to share a *String* object between threads.
- *String* once created does not change. To take advantage of this fact for memory optimization, Java environment caches *String* literals into a special area in memory known as a *String* Pool. If a *String* literal already exists in the pool, then 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

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

Questions

- Why String objects are called immutable?
- How is String object created in memory?
- What are the advantages and 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 calculate hash-code of the String object every 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

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

```
String cityName = new String(“London”);
```

Questions

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

String Interning

- *String interning* is a concept of storing only 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 extremely large amount of text in the memory pool can lead to *memory leak* and/or *performance issue*.

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

Questions

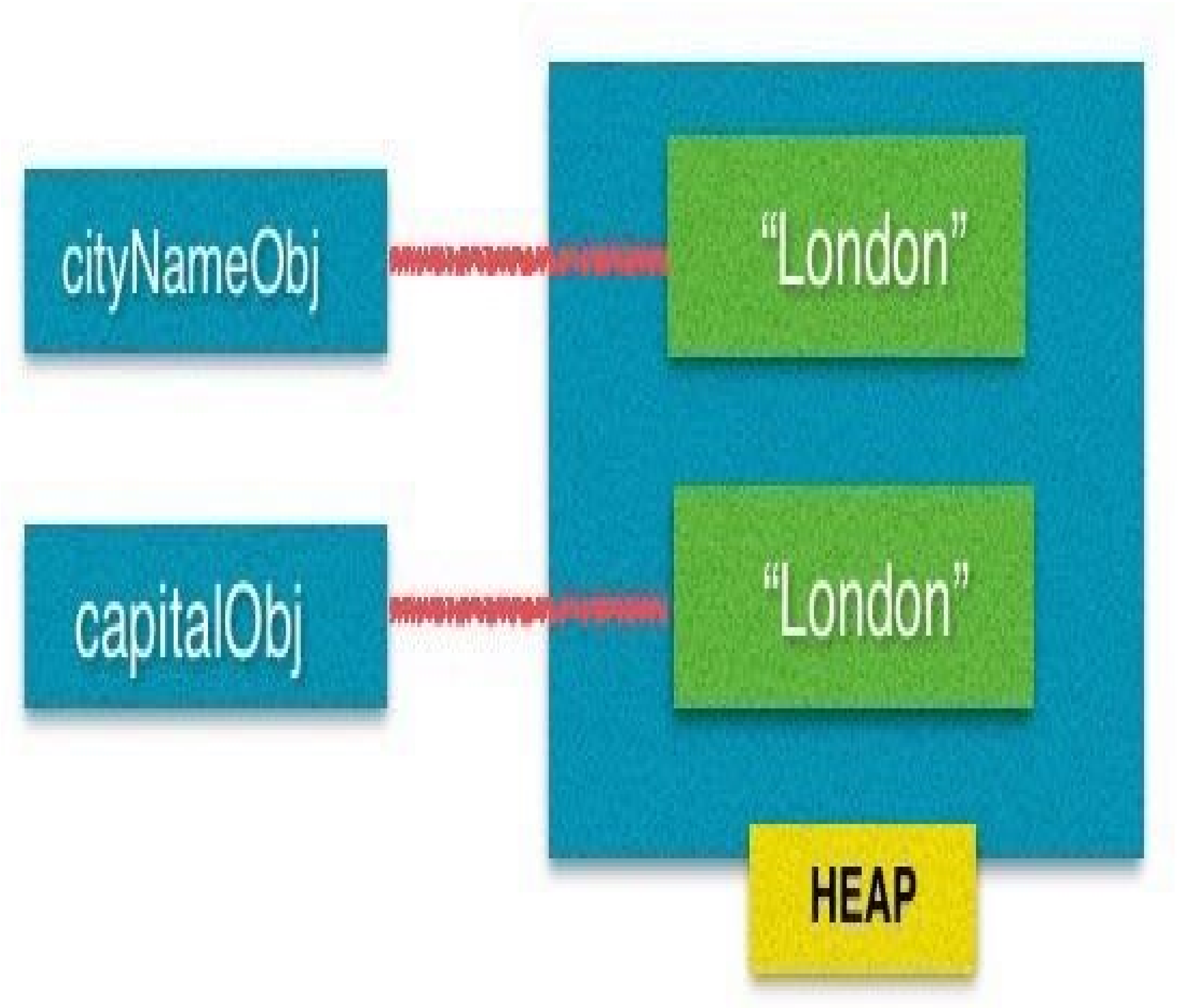
- What is String interning?
- How can you intern a String Object?
- What happens when you store a new String literal value that is already present in the string pool?
- What are the drawbacks of creating large number of String literals?
- Which one is preferred: String Object or String Literal? Why?

String Pool Memory Management

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 the existing String variable in the pool, instead of creating a new literal. Java compiler is able to perform this optimization because String is *immutable*.

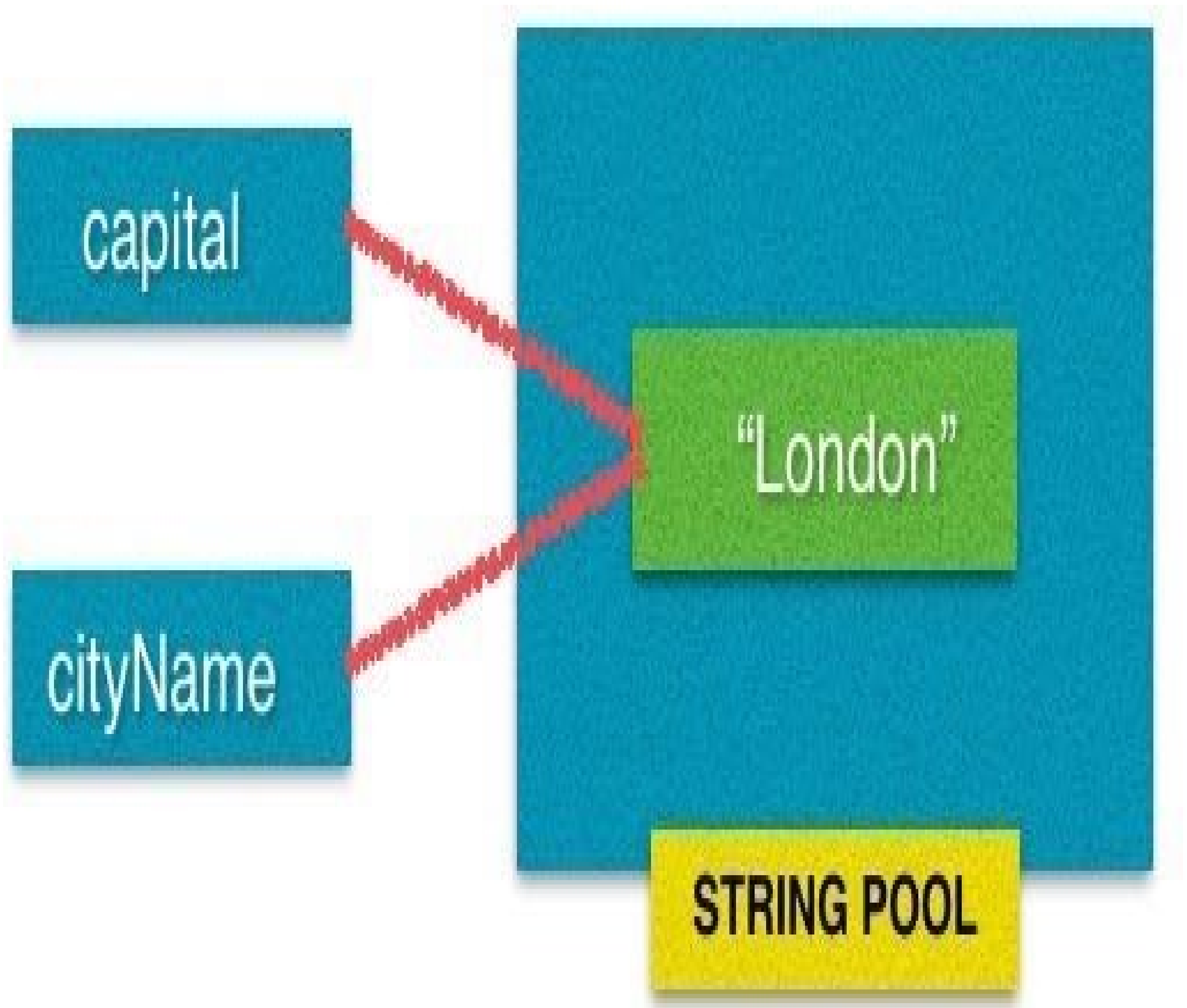
In this example below, both the String objects are different object and are stored into Heap.

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



Whereas in this example below, both *String literal* refer to the same object in 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’t Java use mechanism similar to String Pool, to store objects of other data types?

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, but that’s not recommended.

Since Strings are kept in *String Pool* for re-usability, chances are that the strings will remain in memory for 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 you shouldn’t use String to store sensitive information like password, access key, etc.?
- Why using char array is advisable to store password, instead of String?

Circumvent String Immutability

Immutability feature in String can be bypassed using *reflection*, though using reflection to do so is *NOT* recommended, because it’s a *security violation* and is considered as 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’;

// Prints “XYZ123”
System.out.println(accountNo);
```

Questions

- Can you override String class to modify its immutability?
- Is it technically 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 both allows String values to change.
- Object of both the classes are created and stored in heap.
- Similar methods are available on both the classes.

Differences

- *StringBuffer* methods are *synchronized*, so its thread safe whereas *StringBuilder* is not.
- Performance of *StringBuilder* is significantly better than *StringBuffer*, as *StringBuilder* does not has any 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 international standard character encoding system, which 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 has its own 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 platform and language independent unique number for each character.

Questions

- What are Unicode characters?
- What are the advantages of using Unicode characters?
- What were the problems with old encoding systems?

INNER CLASSES

Inner Classes

Inner Class – is a class within another class.

Outer Class – is an enclosing class, which contains inner class.

Note

- Compiler generates separate class file for each inner class.

Advantages of inner class

- Its easy to implement callbacks using inner class.
- Inner class has access to the private members of its enclosing class, which even the inherited class does not have.
- Inner class helps implementing *closures*; closures makes the surrounding scope and the enclosing instance accessible.
- Outer class provide additional namespace to the inner class.

Questions

- What is inner class?
- What is 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 inner class access private members of the enclosing outer class?
- What benefit does the outer class brings?

Static Member Nested Class

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

Declaration of static nested class

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

    }
}
```

Creating object of static nested class

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

Questions

- What is static nested class?
- If both nested or independent static classes are same, then what’s the benefit of defining an inner class as static?

Local Inner Class

- *Local inner class* is declared and used inside the method block.
- It cannot be declared public, private, protected or at 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 instance of local inner class.
        return new NameIterator();
    }
}
```

Note

- 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 aspect.

Questions

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

Non-Static Nested Class

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

Declaration of non-static nested class

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

Creating object of non-static nested class

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

Questions

- What is non-static nested class?
- Why a non-static nested class can be used as closures?
- Can you create instance of 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 Anonymous class, *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 instance of local inner class.
        return nameIterator();
    }
}
```

Notes

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

Questions

- What is anonymous inner class?

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

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 compiler warning when the interface is not a valid functional interface.

Function Interface Greater

```
@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 +
                ‘}’;
    }
}
```

Code below demonstrates the usage of *Lambda expression* to find the account with the greater balance. Similarly same functional 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 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 anonymous class. Lambda expressions implements *Functional Interface* more compactly. Lambda Expressions are primarily useful when you want to pass some functionality as argument to another method and defer the execution of such functionality until an appropriate time.

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

Pure Functions

Pure functions are function whose results depend only on the arguments passed to it and is neither affected by any state change in the application nor it changes the state of the application. 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 it guaranteed that the Pure Function will always return the same results for the same arguments?

Fluent Interface

Fluent interface is used to transmit commands to subsequent calls, without a 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 number of unnecessary objects created otherwise.

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

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

Fluent interfaces are primarily used in scenarios where you build queries, create series of objects or build nodes in hierarchal order.

Questions

- What is 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 same code in a type (class or interface) or a method to operate on objects of different types, while providing compile-time type safety. *Generics* are introduced to enforce type safety especially for the collection classes.

- Once a type of the parameter 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 variable of type is declared or object of type is created.

```
// variable of List type declared
```

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

Notes

- There is parameter naming convention that is generally used in *Generics*. E for element, T for type, K for key and V for values.
- Generic type is a compiler support. All the type parameters are removed during the compilation and this process is called *erasure*.
- Due to strong type checking, generics 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 error
objectNodeId = integerNodeId;
```

```
(error)
```

Questions

- What is Generics?
- What are the various benefits that Generics provide to the Java collection framework?
- What is the meaning of the statement that “*Generics is compiler support*“? Are Generics not available runtime?
- What are the parameter naming convention that is generally used in Generics?

Type Wildcards

- Wildcard in generics is represented in form of *"?"*. For example, method which takes *List<?>* as parameter will accept any type of List as argument.

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

- Optional *upper* and *lower* bounds are placed to impose restrictions, as exact parameter type represented by wild card is unknown.

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

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

Lower Bound

- To impose restriction, lower bound can be set on the type parameters. Lower Bound restricts a method to accept unknown type argument, which is 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 compiler's ability to look at method invocation and declaration to infer the type arguments.
- In *Generics*, operator called *Diamond operator*, <>, facilitates type inference.

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

Questions

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

Generic Method

- Generic methods define their own type parameters.

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

- If we remove the *<T>* from above method, we will get compilation error as it represents the declaration of the type parameter in a generic method.
- The type (class or interface) that has 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 Bounded Type parameter to increase flexibility of method arguments, at the same time it also helps to restrict the types that can be used.*

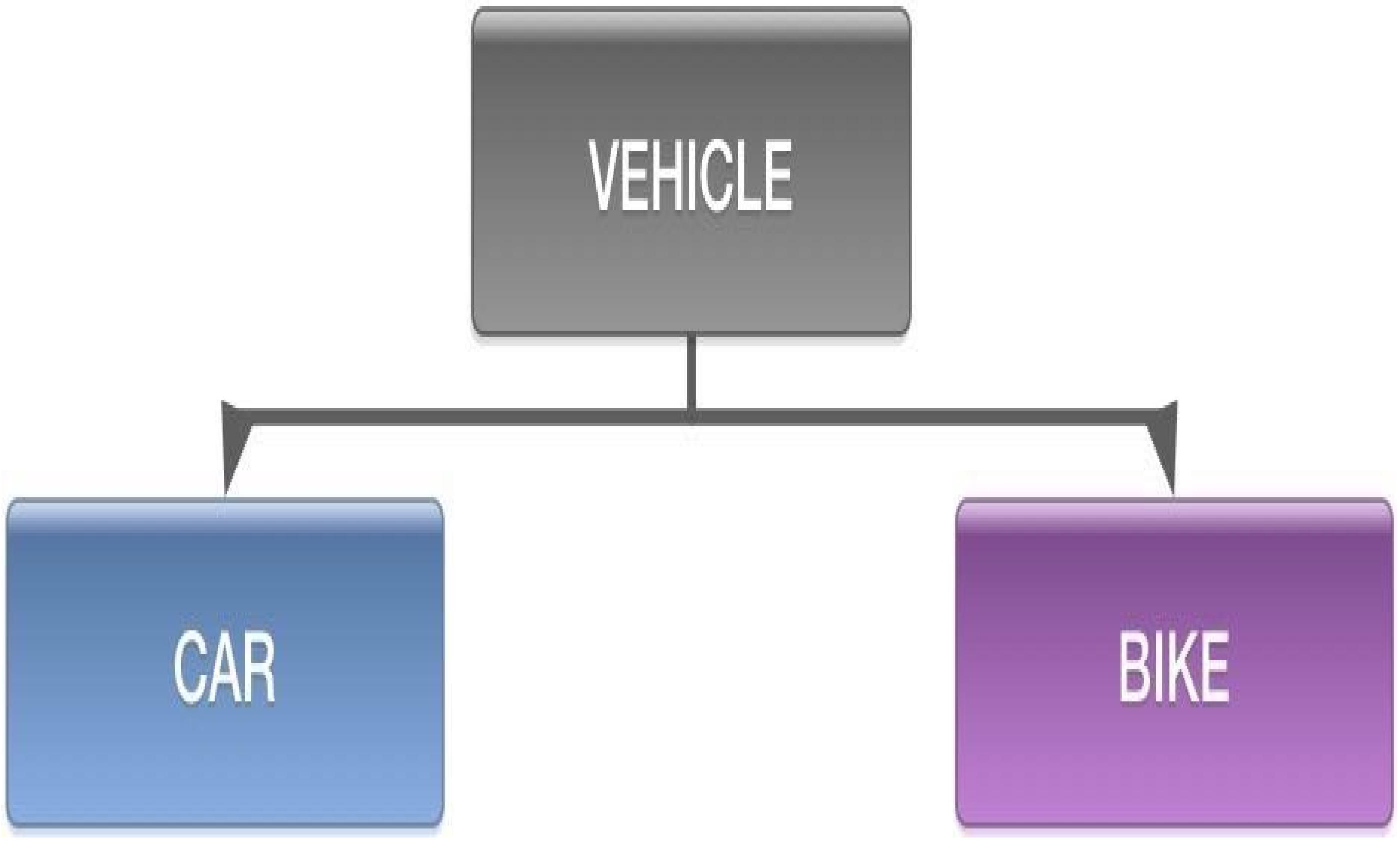
Questions

- Explain Generics method?
- Can you add a Generic method to a non-Generic type?
- What are the benefits of defining bounded type as method parameters?
- How's 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 variable of List<*Vehicle*> ?

Actually not, List<*Car*> or List<*Bike*> can not 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 the incompatible types. For more on this refer Type Erasure.

Java Array

However, unlike *Generics*, array of *Car* can be assigned to 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, runtime type system caught it. Array are *reifable* types, which means that run time is aware of its type.

Questions

- What are 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 than Generics?

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 runtime. As the type information is not available runtime, java compiler takes an extra care to stop you at compile time itself, preventing any heap pollution.

Generics type are 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 where you can read items from a *generics* defined with *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 compile error
vehicles.add(new Bike());
```

This would not be allowed, because Java cannot guarantee what is 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 a concept where you can write items to a *generic* defined with *lower bound type wildcard*, but you cannot read anything from 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,* 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 compile error
Car vehicle = cars.get(0);
```

Questions

- Explain contra-variance?
- Why can't you read 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, and
- 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 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 correct usage of collections. Each collection implementation is written and optimized for specific type of requirement, and interview questions are to gauge interviewee's understanding of such aspects.

Questions are often asked to check whether the interviewee understands 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 collection.
- Uniqueness of elements in a collection.
- Ease of inter-collection operation.
- Read-only collections.
- Collection navigation.

Collection Fundamentals

Collection is a container that groups multiple elements together. 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 are *Generic*.
- All collection types can grow or shrink in size, unlike arrays.
- Java provides lots of methods to manipulate collections based on its usage, so before you add one, always check the existing methods.

Collection Framework

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 object to represent different type of collections.
- *Algorithms* - are applied to these collections to perform various computation 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 collections with the primitive types?
- How can you use collection with the primitive types?
- Explain difference between collections and arrays?
- What is the benefit of collection framework?
- What are the different components of collection framework?

Collection Interfaces

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

Following are the most important interfaces that define collections and their behaviour. 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 FIFO manner.
- *Deque* - Deque holds elements before processing in both, FIFO and LIFO manner.
- *Map* - Map contains 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 collection of unique elements.
- Elements in the *Set* are stored un-ordered.
- Only one null element can be added to a *Set*.
- Duplicate elements are ignored.
- When ordering is not needed, *Set* is fastest and has 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 ordering of elements to be either sorted in a natural order by implementing *Comparable* interface or custom sorted using *Comparator* object.
- *TreeSet* is an implementation class for the *SortedSet* interface.

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

- Use *Comparator* object to perform custom sorting.

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

- If an element implements *Comparable* interface, then *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 *NavigableSet* interface.

Questions

- Can you add duplicate elements in a Set?
- Which collection 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 order of insertion?
- What happens to ordering, if same element is inserted again in a Set? Will it maintain its original position or inserted at the end?
- Which Set class should you use to ensure ordering of the elements?
- Which Set class should you use when you need to traverse in both the directions?

LIST

- *List* is an ordered collection of objects.
- *List* can have duplicate.
- *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 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 element. If elements are added beyond its capacity then the complete array is copied to newly allocated place.

Linked List

- *LinkedList* is based on 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 neighbouring 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 index based system for its elements whereas LinkedList has more overhead as it requires traversal through all its elements.*

Questions

- Can you add duplicate elements to a List?
- Can you add null element to a list?
- In which scenarios would you use ArrayList: frequent add/update or frequent reading?
- In which scenarios would you use LinkedList: frequent add/update or frequent reading?
- Between ArrayList and LinkedList, which one consumes more memory? Why?
- Between ArrayList and LinkedList, which one would you prefer for frequent random access? Why?

Queue

- *Queue* is a collection designed to hold elements prior to processing.
- *Queue* has two ends, a tail and a head.
- *Queue* works in FIFO manner, first in and first out.

Types of Queues

- *Queue* - simple queue which allows insertion at tail and removal from 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 its empty.
- *Transfer Queue* - special *blocking queue* where data transfer happens between two threads.
- *Blocking Deque* - combination of *Deque* and *blocking queue*.

- *Priority queue* - element with highest priority is removed first.
- *Delay queue* - element is allowed to be removed only after delay associated with it has elapsed.

Basic Queue

- *Queue* has one entry point (tail) and one exit point (head).
- If entry and exit point is same, its a LIFO (last in first out) *queue*.

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

Priority Queue

- In *PriorityQueue*, a priority is associated with the elements in the queue.
- Element with highest priority is removed next.
- *PriorityQueue* does not allow *null* element.

- Element of queue either implement *Comparable* interface or use *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 *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 *Consumer* may wait for items to become available.

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

Questions

- What is the purpose of Queue?
- How’s Deque different from Queue?

- What is priority queue?
- What is the criterian to remove element from priority queue?
- What is a Delay Queue?
- What is a Blocking Queue?
- Which Queue is a thread safe queue?
- Which Queue implements Producer-Consumer pattern?
- Explain difference between Blocking Queue and Transfer Queue?
- Which Queue is capacity bounded to allows only the specified number of elements in Queue? What happens if you try to put more than its capacity?

Map

- *Map* contain key-value mapping.
- Usually *Map* allow one null as its key and multiple null as values, but its left to the Map’s implementation class to define restrictions.

Implementation

HashMap

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

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

LinkedHashMap

- *LinkedHashMap* is hash table and linked list implementation of *Map* interface.
- *LinkedHashMap* stores entry using doubly linked list.
- Use *LinkedHashMap* instead of *HashSet* if insertion order is to be maintained.
- Performance of HashMap is slightly better than *LinkedHashMap* as *LinkedHashMap* has to maintain 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* value.
- When there is no reference to keys, they become candidate for garbage collection.

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

Sorted Map

- *SortedMap* provides complete ordering on its keys.
- Sorts 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 implementation of *NavigableMap*.
- *NavigableMap* can be accessed in both, ascending and descending order.
- It supports navigation in both direction and getting 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 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 a sorted order.*

Questions

- How Map is different from other collections?
- Can you add a null value as key to a Map?
- Can you guarantee ordering of elements in HashMap?
- How HashMap is able to provide constant-time performance for *get* and *put* operations?
- What is the difference between HashMap and LinkedHashMap?
- If insertion order is to be maintained, which one would you use: *LinkedHashMap* or *HashSet* ? Why?
- What are WeakHashMap?
- When you want to maintain ordering of key, which Map class would you use?
- Which Map class would you use to access it in both ascending and descending order?
- Which Map implement is optimized for thread safety?
- How does ConcurrentHashMap implements scalability, still maintaining a good performance?
- Why the key used for Map and the value used for Set should be of immutable type?
- What are the disadvantages of exposing internal collection object to the caller?
- Which collection is best if you want to traverse the collection in sorted order?

Algorithms

- *Sorting* - is used to sort collection in ascending or descending order. You can either use the natural order, if the *key* class has implemented *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, generally used in games.
- *Data Manipulation* - reverse, copy, swap, fill and addAll algorithms are provided to perform usual data manipulation on 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 collection elements in natural order and *Comparator* object is used to perform custom sort on the collection elements.

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

Using *Comparator* object to perform custom sorting.

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

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

Questions

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

hashCode() and equals()

- *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 *java.lang.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 hash-code during its lifetime.
- Two equal objects of same class must return the same hash-code.

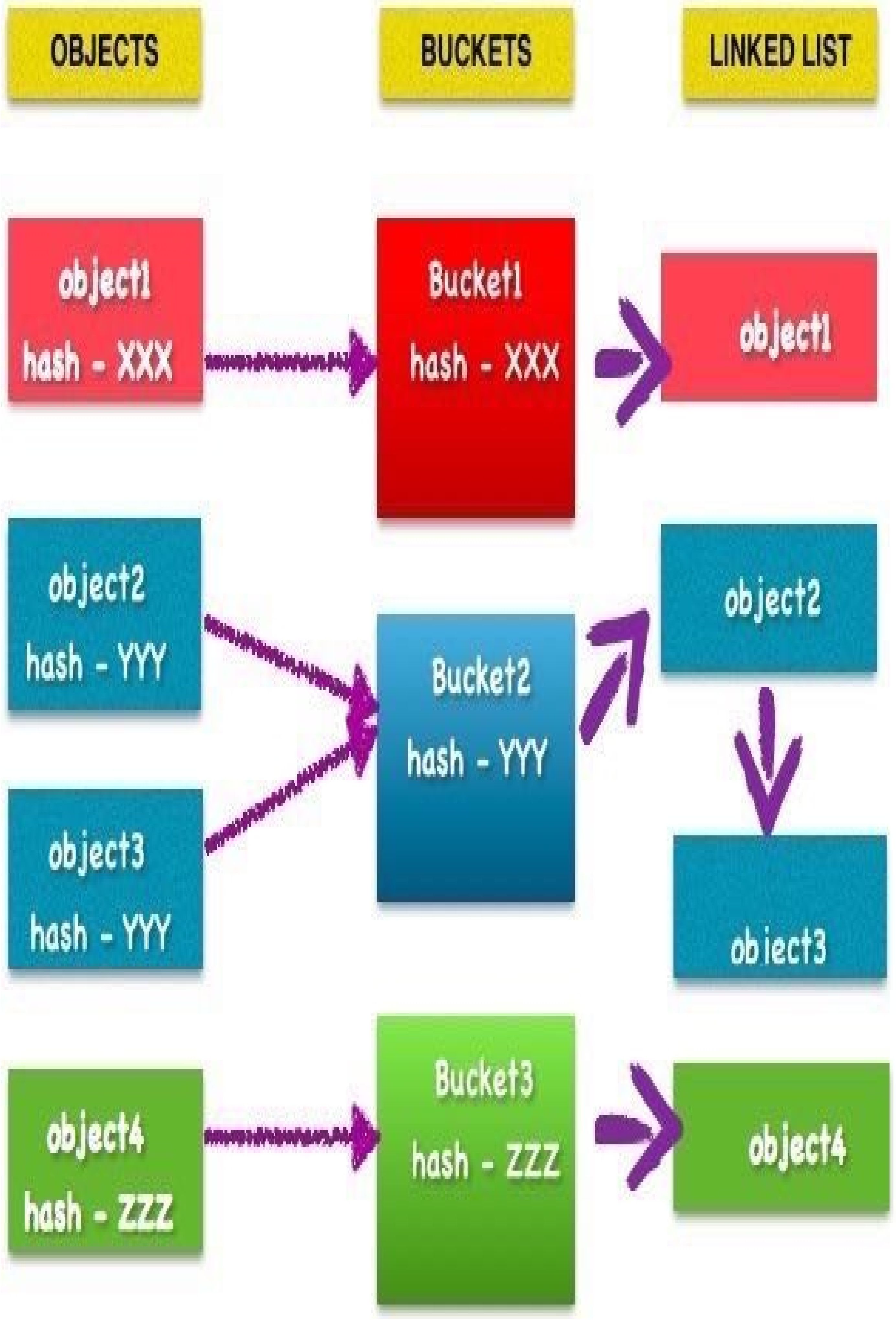
Note

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

How hashCode() and equals() are used

- *hashCode()* and *equals()* methods are 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 hash-code of two objects is different then *equals()* method is not called; otherwise, *equals()* method is called to compare the equality.

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


OBJECTS
BUCKETS
LINKED LIST
object1
hash - XXX
Bucket1
hash - XXX
object1
object2
hash - YYY
Bucket2
hash - YYY
object2
object3
hash - YYY
object3
object4
hash - ZZZ
Bucket3
hash - ZZZ
object4

Questions

- What are the different collections types in Java that use hashCode and equals methods? Why do these collections type use hashCode and equals method?
- When you override hashCode() method, why should you also override equals() method?
- Why do you need to override hashCode() method, when hashCode() method is already present in the *java.lang.Object* class?
- Why should an object 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 object return the same hash-code?
- Can two objects, even when they are not equal, still have the same hash-code?
- What is the purpose of calculating hash-code?
- What is the purpose of equals method?
- If two objects have the same hash code, how are these stored in hash buckets?
- If two objects have different hash code, do you still need equals method?
- If two objects have the same hash code, then how equals method is used?

HashTable vs HashMap

HashTable and *HashMap* are data structures used to keep *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 unsynchronized Objects perform better than synchronized ones due to lack of locking overheads. In a multi-threaded application HashTable should be used.

A HashMap can also be converted to synchronised collection using following method:

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

Questions

- What is the purpose of HashTable and HashMap?
- What is the difference between HashTable and HashMap?
- How does HashTable and HashMap store objects?
- Why HashMap is better for single threaded application?
- Why performance of HashMap is 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 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 a single thread, at a time.
- It is achieved by making all the public method *synchronized*.
- Any composite operation that invokes multiple methods, needs to be handled with locking the operation at client side.
- Examples of synchronized collections are *HashTable*, *HashSet* and *synchronised HashMap*.

Collection Concurrency

- *Collection concurrency* is achieved by allowing simultaneous access to 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 concurrent collection like *ConcurrentHashMap* instead of *HashTable,* as performance of *Concurrent* collection will be better due to less locking overhead.

Note that in *multi-threaded scenarios*, where there are inter-operation 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?
- In case of composite operation, which needs to invoke multiple methods to complete the operation, how can you ensure synchronized access to a collection?
- How do you achieve collection concurrency?
- How does ConcurrentHashMap provides concurrent access to multiple threads?
- Why performance of ConcurrentHashMap is better despite the fact that it allows simultaneous writes from multiple threads?
- Which one would you prefer between ConcurrentHashMap and HashTable? Why?

Iterating Over Collections

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

Iterable.forEach

forEach method is available with the collections that implements *Iterable* interface. *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

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

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

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

Iterator

- *Iterator* has an ability to move in both *backward* and *forward* directions.
- You can remove entries during an iteration when using an *Iterator*, which is not possible when you use *for-each* loop.
- *for-each* 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

- *For-each loop should be preferred over for loop, as for loops are source of errors, specifically related to index calculations.*
- *Iterator is considered to be more thread safe because it throws exception if the collection changes while iterating.*

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 an *Iterator* can be used to iterate through all the elements of the collection.

Questions

- Explain different ways to iterate over collection?
- Can you modify collection *structure* while iterating using for-each loop?
- Can you modify a collection *element* value while iterating using for-each loop?
- What are the limitations with for-each loop, with respect to navigation direction?
- What is difference between for-each loop and Iterator, with respect to navigation direction?
- Can you modify collection *structure* while iterating using an Iterator?

- Can you modify a collection *element* value while iterating using an Iterator?
- Between for-each loop and for loop, which one would you prefer? Why?
- Why Iterators are considered more thread safe?
- Explain the concept of providing *Data Independent Access* to collection?
- How can you design your class to provide Data Independent Access to the collections that are internally used in your class?

Fail Fast

Iterator methods are considered *fail-fast* because Iterator guards against any structural modification made to the collection, after an iterator is retrieved for the collection. This ensures that any failure is reported quickly, rather than the application landing into a corrupt state some time later. In such scenario, *ConcurrentModificationException* is thrown.

Iterators from *java.util* are fail fast.

Questions

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

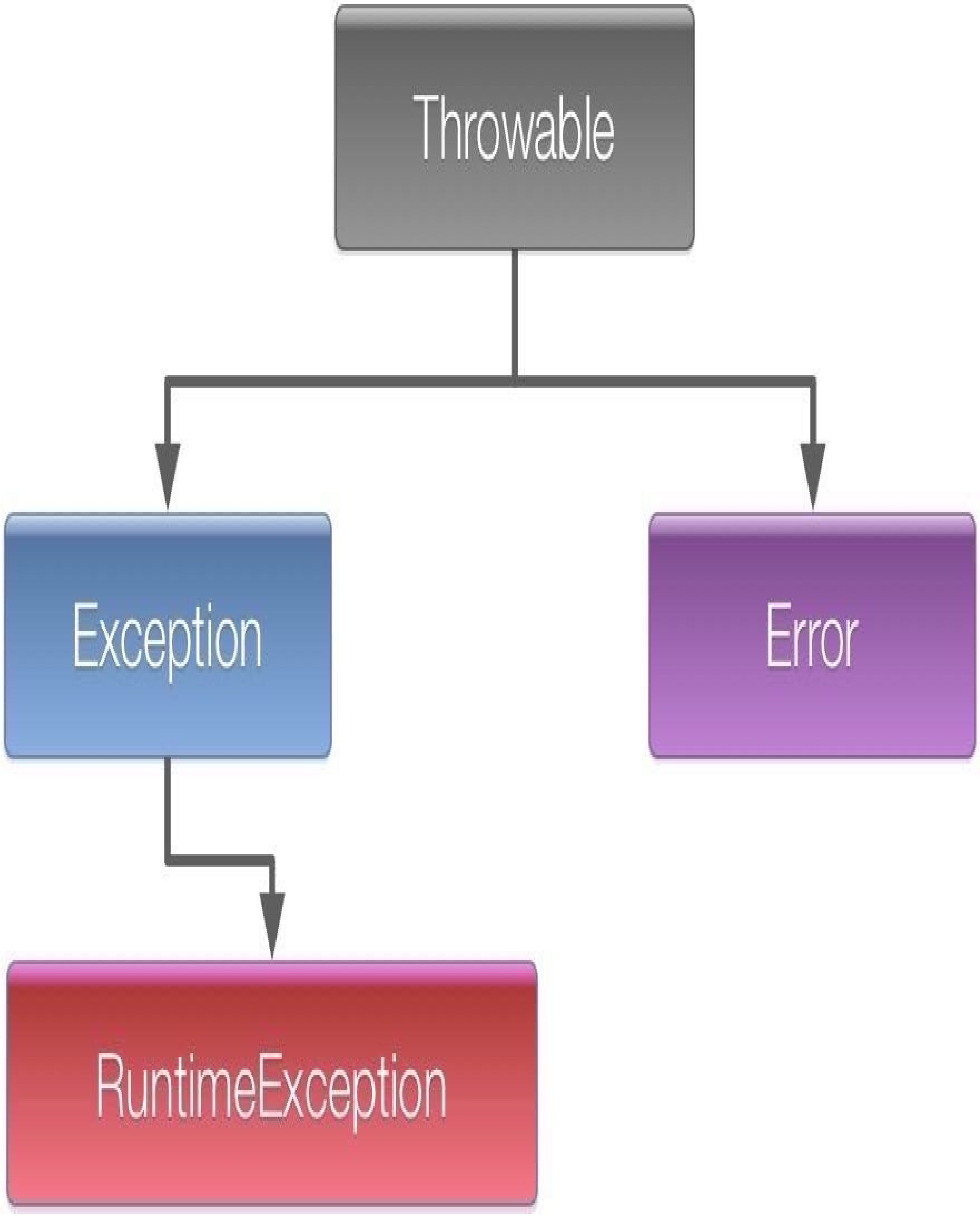
ERROR

AND

EXCEPTION

EXCEPTION

Exception class hierarchy in java



- *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 *unchecked* exception.
- Though you can theoretically subclass *Throwable* class too to create *checked* exception, but that’s not recommended, as *Throwable* is superclass for all exceptions and errors in Java.

Questions

- What is an Exception?
- Explain 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 exception?

Checked vs Unchecked vs Error

Checked Exception

- *Checked exceptions* are checked at compile time.
- Checked exceptions extend *Throwable* or its sub-classes, except *RunTimeException* and *Error* classes.
- Checked exceptions are programmatically recoverable.
- You can handle checked exception either by using *try/catch* block or by using *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

- *Unchecked exceptions* are checked at runtime.
- Unchecked exceptions extend *RuntimeException* class.
- You cannot be reasonably expected to recover from these exceptions.
- Unchecked exceptions can be avoided using good programming techniques.
- Throwing unchecked exception helps to uncover lots of defects.

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

Error

- *Error* classes are used to define *irrecoverable and fatal exceptions,* which applications are not supposed to catch.

- Programmers cannot do anything with these exceptions.
- Even if you catch *OutOfMemoryError*, you will get it again because there is 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 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 un-handled?
- What are the different ways to handle checked exceptions?
- Which exception classes can you use in the catch block to handle both checked and unchecked exceptions?
- How do you make choice between checked and unchecked exceptions?
- Can you recover from unchecked exception?
- How can you avoid unchecked exception?
- Can you throw checked exceptions from static 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 exception.*
- What should you do to handle an Error?

Exception Handling Best Practices

Even though exception handling is primarily driven by context, but it’s very important that 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 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 reason for it.
4. Don’t catch and continue program execution - with some default behaviour. Default behaviour defined today may not be valid in future.
5. Don’t show generic error message to user - instead clean the exception handling code to report user-friendly message with 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 reason for the exception.
7. Don’t catch multiple checked exceptions in single catch block - as it will be impossible to ascertain 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 compelling reason not to do so.
11. Don’t catch Throwable -as it will be impossible to ascertain reason for the exception.
12. Don’t use exception to control the flow of execution - instead use boolean to validate a condition where possible.
13. Handle different scenarios programmatically - instead of putting all coding logic in 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 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 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 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 you shouldn’t use exceptions to control the flow of program execution?
- Should you log all the exceptions? Why?
- Why you shouldn’t use Throwable or some other root level class to catch exceptions?
- What should be the criteria to select the code block that should be enclosed into a try block?
- If a nested call is made, which passes through multiple methods, would you implement try-catch in each method? Why?

try-with-resource

try-with-resource is 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-resource* must inherit *AutoCloseable* Interface.
- You can specify multiple resources within a try block.

Questions

- Which java construct can you use to close the system resources automatically?
- To use a class object within try-with-resource construct, which Interface should the class inherit from?

THREADING

Threading Terms

- *Thread* is a smallest piece of executable code within a process.
- *Program* is set of ordered operations.
- *Process* is an instance of a program.
- *Context Switch* is expensive process of storing and restoring the state of thread.
- *Parallel processing* is simultaneous execution of 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 lock.

Basic Concepts

- All Java programs begin with *main()* method on a user thread.
- 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 process of 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 a program and a process?
- Explain context switching of thread?
- What is parallel processing?
- What is multi-threading?
- How parallel processing and multi-threading related?
- What is deadlock?
- What is a user thread?
- What is thread local storage? What are the things would you store with thread local storage?
- Do threads share stack memory?

Thread Lifecycle

Following are various stages of thread states .

New - Thread is created but not started.

Runnable - Thread is running.

Blocked - Thread waiting to enter critical section.

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

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

Terminated - Thread has completed its task and exited.

Notes

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

Questions

- Describe different stages of thread lifecycle?
- What is difference between blocked state and waiting state?
- How can you find thread’s state?
- How thread sleep() method is different from thread wait() method?

Thread Termination

The thread should be stopped calling *interrupt()* method. Calling thread interrupt even breaks out of *Thread.sleep()*.

The operation executing on thread should recurrently call *isInterrupted()* method to check if 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
}
```

Notes

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 thread's stop method?
- What happens if a thread is sleeping and you call interrupt on the thread?
- Does calling interrupt stops the thread immediately?
- Why you shouldn't call thread's stop and suspend methods?

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 create a new thread but executes as a normal method in the same thread it's created on, whereas *Thread's start()* methods creates a new thread.

- *Runnable* is preferred way to execute a task on a thread, unless you are specializing *Thread* class, which is unlikely in the 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 Runnable interface is different from extending Thread class?
- When should you extend Thread class?
- When should you inherit Runnable Interface?
- Between Runnable and Thread, which one is the preferred way?
- Does implementing 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;
}
```

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

Questions

- What is the difference between Callable and Runnable interface?
- What is the benefit of using Runnable over Callable?
- Can you throw checked exception from Runnable interface?
- Why Runnable and Callable interfaces are called Functional Interface?
- Which of the Interface returns result: Runnable or Callable?

Daemon Thread

A *daemon thread* is a thread, which allows *JVM* to exit as soon as program finishes, even though it is still running. As soon as JVM halts, all the daemon threads exists without unwinding stack or giving chance to *finally* block to execute. Daemon thread executes on very low priority.

Threads inherit the daemon status of 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 following:

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

All non-daemon threads are called user threads. User 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 Daemon thread?
- What is user thread? What is main thread?
- Can a program exit if Daemon thread is still running?
- Can a program exit if user thread is still running?
- When JVM halts exiting all running Daemon thread, does the finally block on a Daemon thread still gets a chance to execute?
- When a new thread is created; is it created as a user thread or a Daemon thread?
- What happens when no user thread is running but a Daemon thread is still running?
- What is Java main thread: a user thread or a Daemon thread?
- When you create a new thread on the main thread, what’s the type of thread created: daemon or user?
- For what type of jobs should you use Daemon thread?

Race Condition and Immutable Object

Race condition occurs when multiple threads concurrently access a 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 unpredictable data value.

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

To make an object Immutable

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

Questions

- How immutable objects help preventing race condition?
- Why race condition may produce unpredictable results?
- Why immutable objects are considered safe in multi threaded environment?
- Why should you declare immutable class as final?
- Why constructor should be the only way to create immutable object? What happens if setters are provided?

Thread Pool

Thread Pool is a collection of specified number of worker threads, which exists 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 of requests beyond its capacity to process, by preventing application from going into a hang state or from 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 *Executors* class where you can use *newFixedThreadPool()* factory method to create *ExecutorService* to execute tasks.

Questions

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

SYNCHRONIZATION

Concurrent vs Parallel vs Asynchronous

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

Concurrent processing is simultaneous execution of multiple tasks; either on multiple cores or by 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

Race condition occurs when multiple threads concurrently access a 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 unpredictable data value.

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

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

Synchronization creates memory barrier, known as *happen-before*, which ensures that all the changes made by a thread to the local objects in the critical section, are available to any other thread that acquires the same local objects subsequently.

Questions

- Explain race condition in multi-threading?
- What is 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

synchronized keyword is used to mark critical section in the code.

Mutual exclusion synchronization is achieved by locking the critical section using *synchronized* keyword.

This can be done in following two ways.

Marking method as 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 critical section

```
public void writeToDatabase() {
    // multiple threads can reach here

    // Code marked as 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 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 entire class object.*

- *Inside synchronized, never call a method provided by the client code or the one that is designed for inheritance.*

Questions

- How is synchronized keyword used?
- What is the difference between synchronized method and synchronized block?
- Why synchronized block is preferred to synchronized method?
- Does synchronized method locks the entire object?
- What problem you may encounter if you call a method provided by the client, from inside the synchronized block or method?

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 waiting thread(s).
    - *notify()* - signals only one random thread.
    - *notifyAll()* - signals all threads in wait state.
2. *wait()* 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. *notify()* 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?
- When you call notify, with multiple threads waiting for the notification, which one will be notified?

- Between notify and notifyAll, which one would you prefer? Why?

Volatile

In a *multi-threaded* application, every thread maintains a copy of variable from 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.

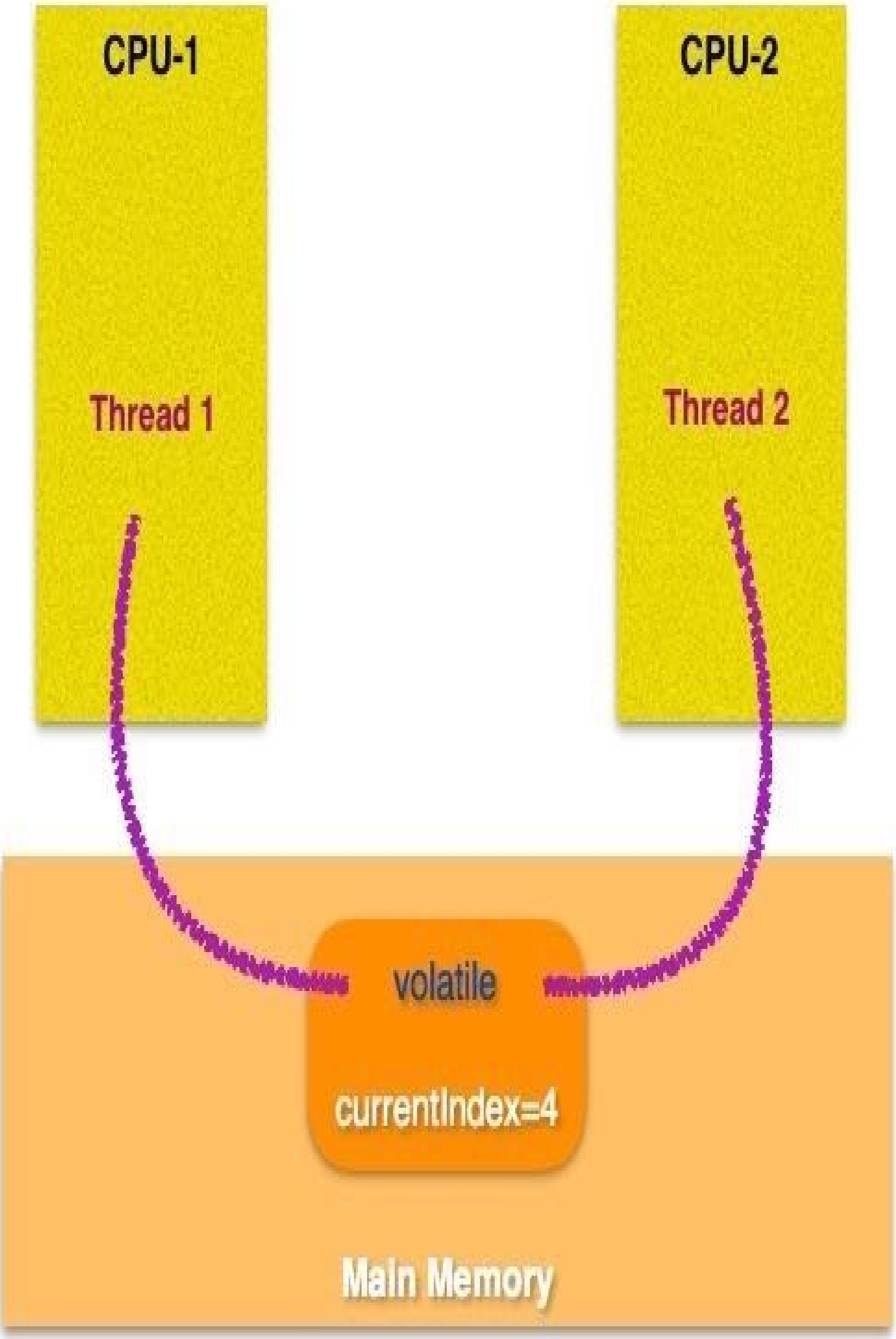
A field marked *volatile* is stored and read directly from the main memory. As *volatile* field is stored in the main memory, all the threads have visibility to most updated copy of the *volatile* field’s value, irrespective of which thread modified it.

Consider a class *Ledger,* which has a member currentIndex to keep track of 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;
}
```


CPU-1
Thread 1
CPU-2
Thread 2
volatile
currentIndex=4
Main Memory

Questions

- What is volatile field?
- Explain the problem that volatile field solves?
- Where does the volatile field gets stored?

static vs volatile vs synchronised

static Variable

static variables are used in context of class objects where only one copy of static variable exists irrespective of how many objects of the class are created.

But if there are multiple threads 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 context of threads, where only one variable exists irrespective of how many threads or objects accessing it and everyone always get 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 modifier 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 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 the objects on another thread?
- If both static and volatile variables are shared across objects, then what’s the problem a volatile variable solves?
- How are volatile and static variables different from synchronized, as even the synchronized monitor guards the memory object?

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 access to that object.
- Life of ThreadLocal objects is tied to the thread it’s created for, unless there are other variables referencing the same object.
- To prevent leak, it’s a good practice to remove ThreadLocal object using remove() method.

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

Questions

- What is thread local storage?
- What is the kind of information should you store in Thread Local Storage?
- Why you don’t need to synchronize access to the objects that are stored in ThreadLocal Storage?

- Why should you call remove method on 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 specified interval.

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

- *wait* is called on the object's monitor; whereas, *sleep* is called on *thread*.
- Waiting object can be notified; whereas, sleeping thread cannot.
- Sleeping thread cannot release a lock; whereas, waiting object can.
- To wake a sleeping thread you need 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 waiting object to wake but not the sleeping?
- You need direct handle to wake a sleeping thread; 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 current thread cannot proceed, until the target thread on which it depends, 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 thread’s join method?
- Why do you need to join two threads?

Atomic Classes

Atomic classes provides ability to perform atomic operations on 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 atomic classes?
- Why you don't require synchronizing access to an object of 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 the three are not synchronized together, so any thread context switch that happens in between, will result into undesired result.*
- 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 variable is declared as Volatile. Volatile just guarantees happen-before reads.*

Lock

- *Locking* is a mechanism to control access to the shared resources in a multi-threaded environment.
- *ReentrantLock* class implements *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, 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 locking mechanism in a multi-threaded environment?
- Do you need to acquire and release lock in same block of code?
- Why should you prefer using tryLock() instead of lock()?

ReadWriteLock

- *ReadWriteLock* maintains pair of associated locks, one for writing and the other for read-only operations.
- Only one thread can acquire 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 scenario would you prefer ReadWriteLock to any other locking mechanism?

Synchronisers

Synchronizers synchronizes multiple threads to protect a *Critical Section*.

Sync

Point ***Threads***

```
  |      ______
  |        _______-
  |     _____-
```

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

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

- A *Barrier* is called 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 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 happen if one of thread dies, for which other threads were waiting at the Barrier?
- For what kind of work would you use Barrier?

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

Questions

- Explain Semaphore synchronizer?
- What happens if you call release() more number of time than acquire()?
- What happens when you call acquire, but permit is not available?
- How can you acquire 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 *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 *arriveAndAwaitAdvance()* method to wait for all parties to arrive before proceeding to the next phase.
- *Phasers* monitors 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 *arriveAndDeregister()* method, from moving to the next phase.

Questions

- Explain Phasor synchronizer?
- Can the number of parties registered with Phasor dynamically change over time?
- Can you reuse the same Phaser object again?
- Can you monitor count of registered parties with Phasor using some external object?
- What is the difference between Semaphore and Phaser, with respect to number of parties that can register with it?

Exchanger

- *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 thread are required with the Exchanger object?
- What is the primary purpose of Exchanger synchronizer?
- Does Exchanger synchronizer uses 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.
- *Latch* cannot be reused.
- *CountDownLatch* class provides implementation for *Latch*.
- All threads wait calling *await()* method till *countDown()* is called as many times 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();

// awaiting 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 mechanism of signalling a job completion to Latch?

Executor Framework

Executor framework provides an infrastructure to execute set of related tasks on thread.

It takes care to manage the following.

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

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

Questions

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

Executor Service

- *ExecutorService* inherits from *Executor* interface providing 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* object 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 single thread, use *newSingleThreadExecutor()* method.

Handling Results

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

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

- *submit()* 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 executor
      Future<String> task =
            exec.submit(
                  new Callable<String>() {
                     @Override
                     public String call()
```

```
                                throws Exception {
                              //some logic
                              return null;
                          }
                      });
        // waits for result
        String result =
                task.get();
        // Shutdown executor
        exec.shutdown();
    }
}
```

- If there is any 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 task to run in future.
- Methods 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 processor to configure thread pool.*

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

Questions

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

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

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

- *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* - task that yields result.
    - *RecursiveAction* - task that does not yield result.
- Both, *RecursiveTask* and *RecursiveAction* provides abstract *compute()* method to be implemented by the class, whose object represents the *ForkJoin* task.

Questions

- Explain Fork-Join framework in Java?
- How Fork-Join framework helps 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 behaviour of an application.

Purpose of reflection

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

- Reflection facilitates modular software development by investigating code and libraries at runtime, to plugin classes and components.
- Annotations are read using reflection. *JUnit* uses reflection to discover methods to setup and test.
- Debuggers uses 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 use of reflection to access *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 does debuggers use reflection?
- How does reflection helps in creating modular software architecture?
- Can you access private members using reflection?
- Give some examples, where reflection is used by 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’s 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 an 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 violates encapsulation?
- How does reflection violates 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 Internet of Things, along with JSON’s ability to be easily parsed to JavaScript object, it has become preferred data format for integration with the web. Following are the few primary factors behind this.

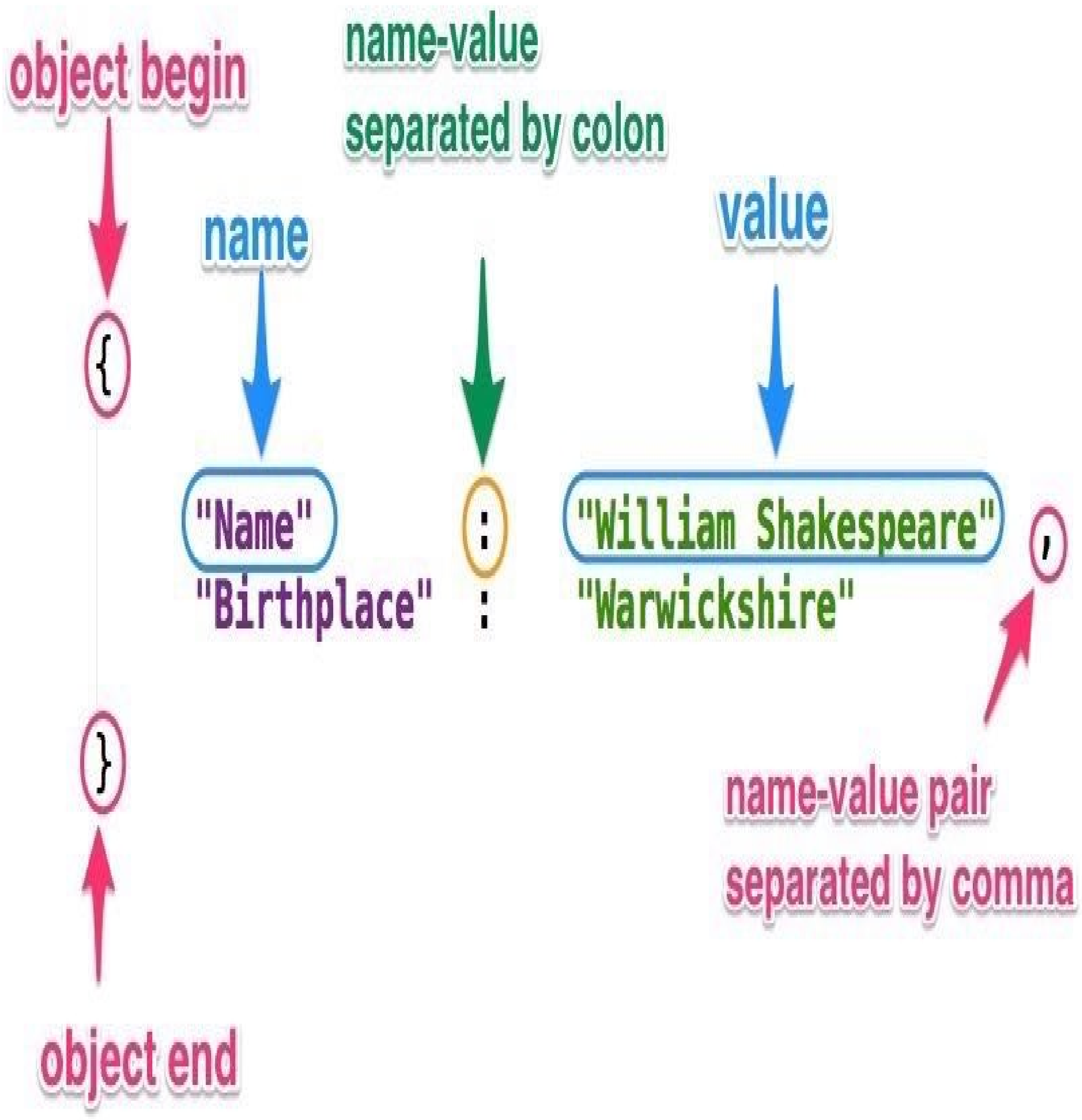
- It’s interoperable, as it’s restricted to primitive data types only.
- It’s lightweight and less verbose than XML.
- It’s very easy to serialize and transmit structured data over network.
- Almost all modern languages support it.
- JavaScript parser in all the popular web browsers supports it.

JSON Structures

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

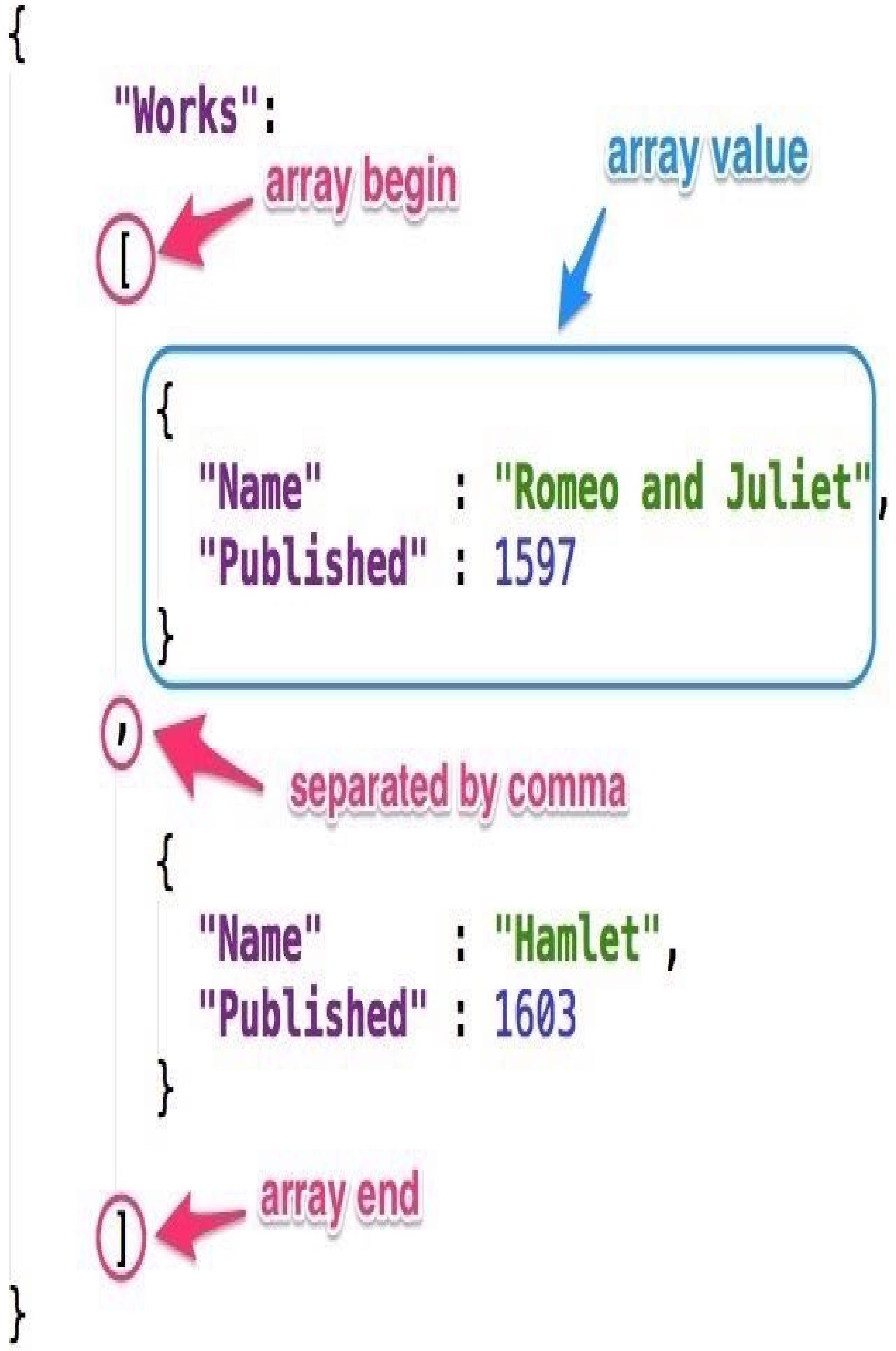
JSON Object

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.



```
{
    "Works":
    [
      {
        "Name"      : "Romeo and Juliet",
        "Published" : 1597
      }
    ,
      {
        "Name"      : "Hamlet",
        "Published" : 1603
      }
    ]
}
```

Questions

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

MEMORY MANAGEMENT

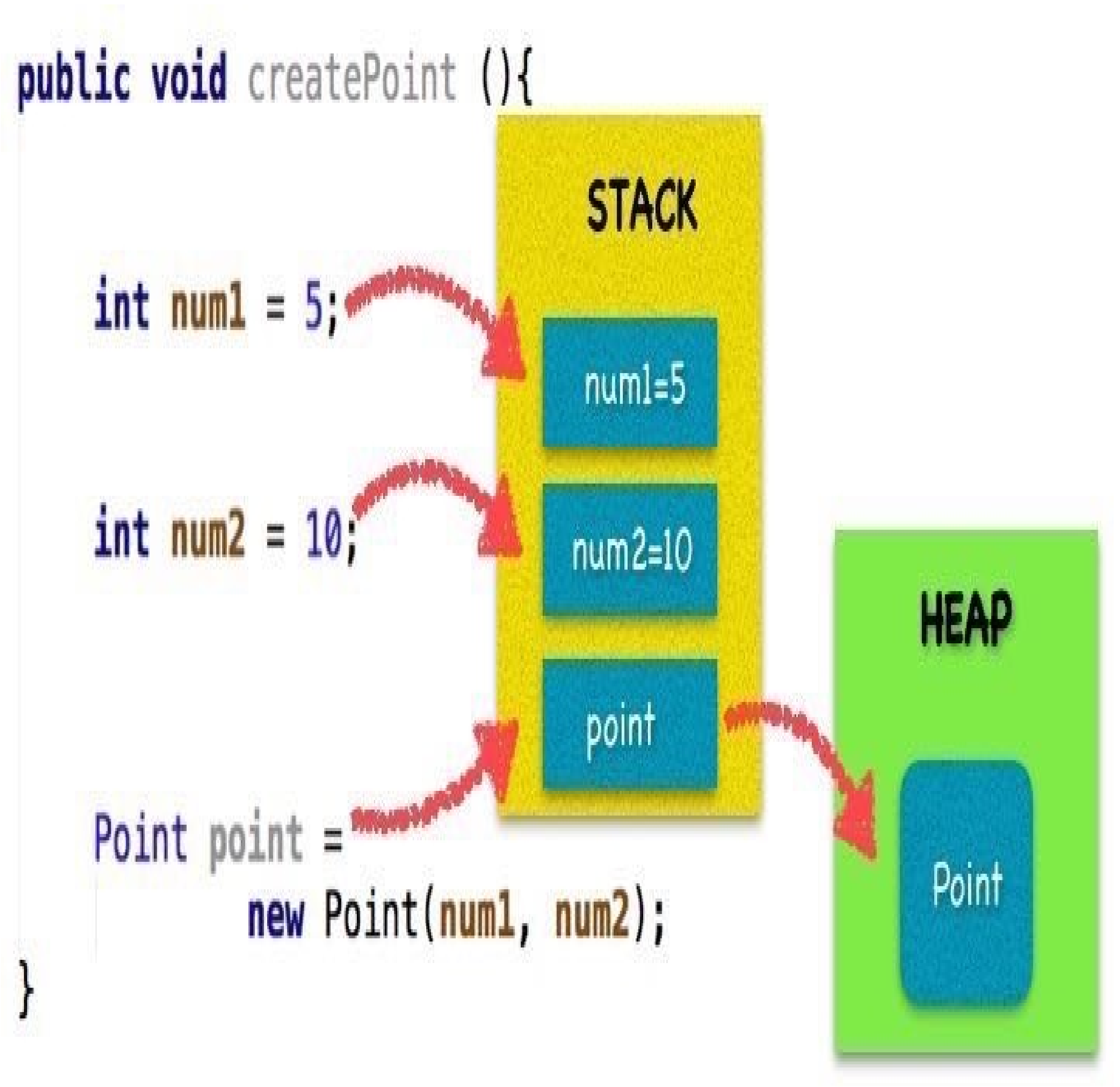
Stack vs Heap

- *Stack* is memory associated with each system thread when it’s created; whereas, heap is shared by all the threads in an application.
- For each function the thread visits, a block of memory is allotted on the top of 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 heap are destroyed manually.
- When the thread exists, 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, it’s as simple as adjusting pointers. 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. Heap memory is shared across multiple threads in application; synchronization with other thread has performance consequences.
- 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 *createPoint* is invoked.

- **num1** and **num2** variables are stored in stack.
- Reference **point** variable is stored in stack.
- Object of class *Point* is stored in heap.



```java
public void createPoint (){

    int num1 = 5;

    int num2 = 10;

    Point point =
            new Point(num1, num2);
}
```

Questions

- What is stack?
- What is heap?
- How is memory allocated and de-allocated in stack, during its lifecycle?
- When does the stack memory gets released?
- When does the heap memory gets released?
- Why memory allocation in stack is faster as compared to 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 stack?

Heap Fragmentation

Heap fragmentation happens when a Java application allocates and de-allocated small and large blocks of memory over a period of time, 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 long *Garbage Collection* cycle as JVM is forced to compact the heap. *Avoiding allocating large block of memory, by increasing heap size, etc, can control heap fragmentation*.

Questions

- What is heap fragmentation?
- Why does heap fragmentation happens?
- Who has the responsibility to reduce heap fragmentation?
- How can you control heap fragmentation?
- What happens when there is very high level of heap fragmentation?
- Why does heap fragmentation slows 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 sub-interface *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 is serialization and de-serialization?
- Which interface needs to be implemented by the type that wants 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 un-referenced objects from the memory and also move the remaining objects together to release contiguous block of memory. When Garbage collection happens, all the running threads in the application are suspended during the collection cycle. Garbage Collector run on a Daemon thread.

Moving all the surviving objects together *reduces memory fragmentation*, which improves the speed of memory allocation.

During 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 Garbage Collection cycle?
- Why Garbage Collection is considered expensive process?
- Why after Garbage Collection cycle, usually the performance of the application improves?
- How does Garbage Collection prevents OutOfMemoryException?
- Which type of thread is used for Garbage Collection: Daemon or user thread?
- What is memory fragmentation?
- Why does application seems to slow down when Garbage Collection happens?
- Can you explicitly request a Garbage Collection?
- Which method 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.*

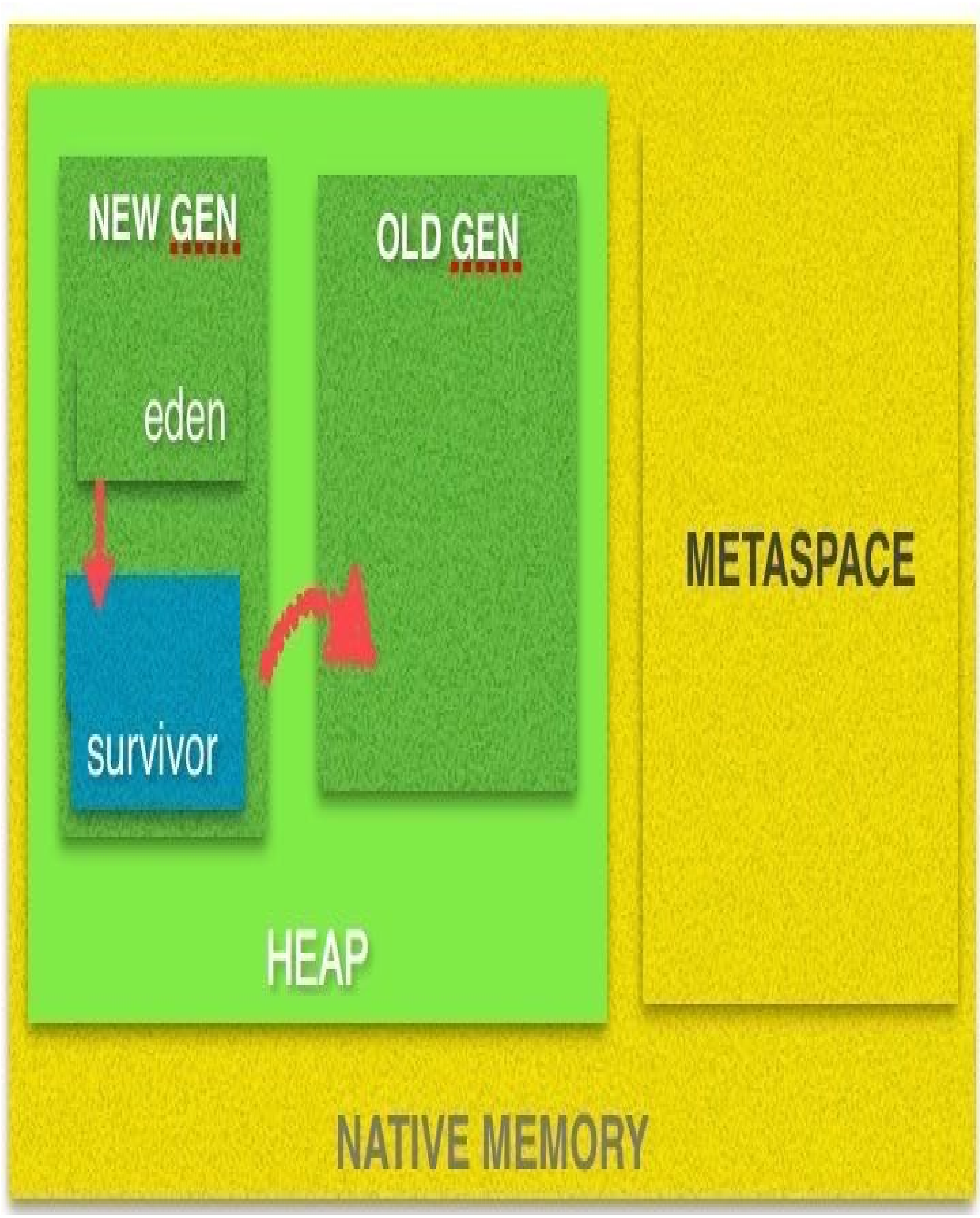
Stack Memory

Stack memory is associated with each system thread and used during execution of the thread. *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 memory space.
- *Old Generation*
- *Metaspace*



When new generation is filled, it triggers garbage collection. Objects that survive this GC cycle in *Eden* are moved to the *Survivor* space. Similarly after few cycles of GC, the surviving objects keeps moving to *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 old generation; so if lots of Garbage Collection happens in old generation, it may lead to timeout error in the application.

Questions

- How is Heap memory divided?
- Explain different generations of heap memory? How objects are moved across generations?
- How is garbage collection cycle triggered?
- What are Eden and Survivor spaces in New Generation memory?
- When does the objects in Eden moves to Survivor space?
- What is Metaspace?
- What type of objects are stored in Metaspace?
- Why Metaspace has virtually unlimited space?
- In which generation does the Garbage Collection cycle runs slowest? Why?
- Which generation is more frequently garbage collected?

Weak vs Soft vs Phantom Reference

If an object in memory has a reference then the garbage collector will not collect it. This principle is not true for Weak and Soft references.

Weak Reference

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 technical purpose to track memory usage.

Questions

- What is weak reference?
- What is soft reference?
- What is Phantom Reference?
- What are the reference types that gets 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 yourselves.
- Unit test helps you to sleep well at night.

Other important reasons

- Unit tests provide immediate and continuous feedback on the success/failure for the changes made to the code.
- Units test increases the confidence to make big changes without worrying about breaking any existing feature.
- Unit test helps you to understand the internals of the code and design.
- Unit tests also serves as documentation on various coding scenarios.
- Unit test saves time in 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 saves time in 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 unit 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 or a feature is completely implemented.

Regression

Regression testing are 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 generally performed after updating or deploying the software, to verify that the changes are made as per the requirements.

Questions

- What is unit testing?

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

Testing Private Members

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

Usage

The code below demonstrates use of reflection to access 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, its not advisable to access private fields using reflection; you have following alternate options to test code in a private method:

- Test the private method through 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 you shouldn't use reflection to test private members?
- What are the alternate ways to test private members?

Mocking

Primary responsibility of a unit test is to verify the conditional logic in the class code, which should run super fast for an immediate feedback. To enable an immediate feedback, it’s necessary that class has no external dependencies; which is not practical in object oriented software development, where you need to communicate with the external objects to perform File IO, manage database access, communicate with web service, etc. So the basic idea of *mocking* is to *replace these external dependencies with mock objects*, to isolate the object under test.

Benefits of mocking

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

Questions

- What are mock objects?
- What are the benefits of using mock objects?
- Why mock objects helps to speed up the tests?
- How does mocking helps to achieve faster development cycle?
- What are the types of dependencies that you should replace mock objects with?

JAVA

TOOLS

Git

Git is source code management systems. Its *distributed version control system (DVCS),* which facilitates multiple developers and teams to work in parallel. Git's primary emphasis is on providing speed while maintaining data integrity.

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

Let's discuss few terms that are frequently used:

Repository - is directory that contains all the project files.

Clone - it creates a working copy of local repository.

Branch - is created to encapsulate the development of new features or to fix bugs.

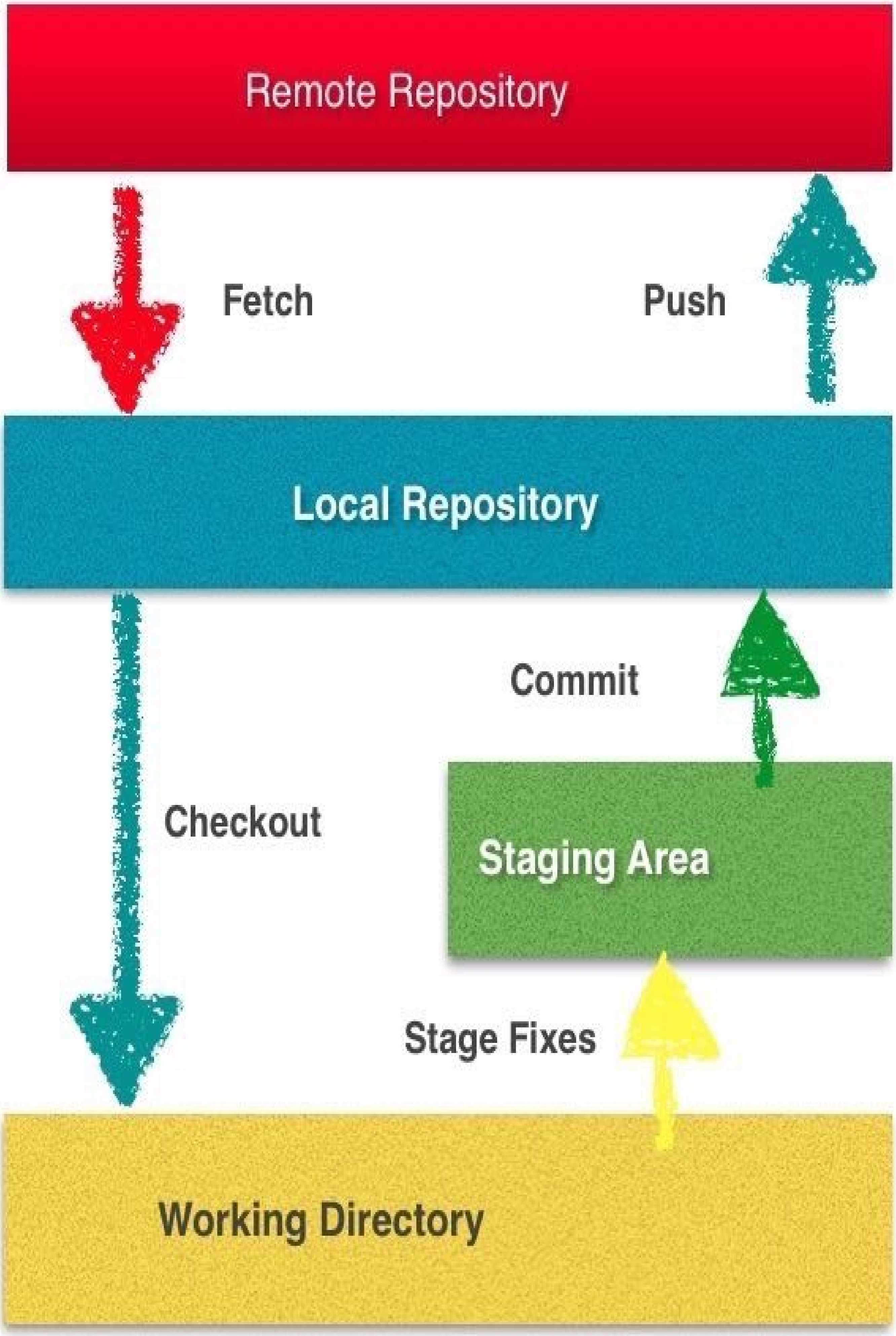
HEAD - points to the last commit.

Commit - commits changes to HEAD and not to 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

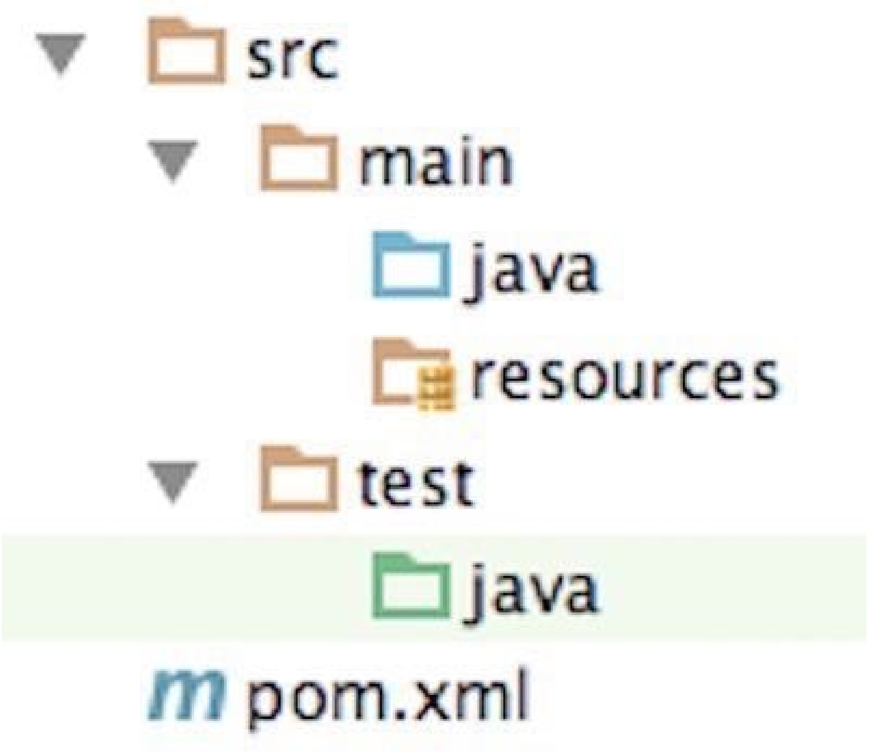

Remote Repository
Fetch
Push
Local Repository
Commit
Checkout
Staging Area
Stage Fixes
Working Directory

Questions

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

Maven

Maven is software project management framework that manages project build, dependency resolution, testing, deployment, reporting, etc. Maven is based on conventions rather than elaborate configurations; that is, if the project is laid out as per the prescribed conventions, as depicted in the image below, then it will be able to find the resources to perform different build operations.



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

Ant

Ant is powerful XML based scripting tool for automating the build process. An automated build infrastructure is very important element in the Continuous Integration cycle.

Ant provides support for things like code compilation, testing and packaging, which can be defined as a series of task. Unlike Maven, Ant is driven by configuration and not convention. Similar to *goals* in Maven, you can define *Targets* in Ants, which are series of tasks. Ant depends on file *build.xml* (you can specify different name too) to execute the targets defined.

Questions

- What is Ant tool used for?
- What all things does Ant support?
- What are differences between Ant and Maven?
- What is the meaning of Targets in Ant?

Jenkins

Continuous Integration

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

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

Jenkins

Jenkins is open source web based tool to perform continuous integration. Jenkins provides configuration options to configure and execute all the above-mentioned activities. Configuration options that are available 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 Continuous Integration framework should support?
- Explain capabilities of Jenkins to support Continuous Integration?
- How do you integrate third party tools with Jenkins?