1. What is the difference between method overloading and overriding in Java? Explain with an example. Can you overload or override static methods?

Method overloading in Java is when a class has multiple methods with the same name but different parameter lists. This allows for having methods with the same name but different functionality based on the number or type of arguments passed to the method. For example:

```
public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }
  public double add(double a, double b) {
    return a + b;
  }
  public int add(int a, int b, int c) {
    return a + b + c;
  }
}
```

Method overriding in Java is when a subclass has a method with the same name and parameter list as a method in the superclass. This allows the subclass to provide its own implementation of the method, which will be used instead of the superclass's method when the subclass object is used. For example:

```
public class Animal {
  public void makeNoise() {
    System.out.println("Some generic animal noise");
  }
}

public class Dog extends Animal {
  @Override
  public void makeNoise() {
    System.out.println("Bark!");
  }
}
```

Static methods can be overloaded, but they cannot be overridden. This is because static methods are bound to the class, not to individual objects, and therefore cannot be replaced with a subclass's implementation.

2. Write a code to display repeated characters in the given String. Also, print the count of how many times each character is repeated.

```
public class DuplicateCharacters {
     public static void main(String[] args) {
        String string1 = "Great responsibility";
        int count;
```

```
        //Converts given string into character array
        char string[] = string1.toCharArray();

        System.out.println("Duplicate characters in a given string: ");
        //Counts each character present in the string
        for(int i = 0; i <string.length; i++) {
            count = 1;
            for(int j = i+1; j <string.length; j++) {
                if(string[i] == string[j] && string[i] != ' ') {
                    count++;
                    //Set string[j] to 0 to avoid printing visited character
                    string[j] = '0';
                }
            }
            //A character is considered as duplicate if count is greater than 1
            if(count > 1 && string[i] != '0')
                System.out.println(string[i]);
        }
    }
}
```

3. What is the map in Java? What are the different types of maps? What is the difference between Hashtable and HashMap? How can you make the HashMap synchronized?

In Java, the **Map interface** is a part of the java.util package and provides a collection of key-value pairs. It maps keys to values, allowing you to retrieve a value using a key.

There are several different implementations of the Map interface in Java, including:

HashMap: This is an unordered map that stores key-value pairs in a hash table. It allows null values and null keys.

TreeMap: This is an ordered map that stores key-value pairs in a red-black tree. It does not allow null keys, but does allow null values.

LinkedHashMap: This is an ordered map that maintains the insertion order of the elements. It allows null values and null keys.

ConcurrentHashMap: This is a thread-safe version of HashMap that can be used in multi-threaded environments.

To make a HashMap synchronized, you can use the Collections.synchronizedMap method to wrap the HashMap in a synchronized map. Here's an example:

```
Map<String, String> map = new HashMap<>();
Map<String, String> syncMap = Collections.synchronizedMap(map);
```

Alternatively, you can use the **ConcurrentHashMap** class, which is a thread-safe version of HashMap.

```
Map<String, String> syncMap = new ConcurrentHashMap<>();
```

4. What is the difference between the int and the Integer? What are their default values? Where they are used? What about the performance?

int is a primitive data type in Java, while **Integer** is a wrapper class for the int data type.

The default value of an int variable is 0. The default value of an Integer variable is null.

int variables are generally used when you need to store and manipulate small values that do not require a lot of memory. They are also used when you need to perform arithmetic operations on them.

Integer variables are generally used when you need to store and manipulate larger values or when you need to store a null value. They are also used when you need to pass an int value as an object, such as when you need to use it as a method argument or when you need to store it in a collection that stores objects (e.g., List, Set, etc.).

In terms of performance, int variables are generally faster than Integer variables because they are stored on the stack and do not require the overhead of object creation. However, the difference in performance is generally not significant unless you are working with very large datasets or in a high-concurrency environment.

In summary, int is generally used for simple, performance-critical tasks, while Integer is used for tasks that require more flexibility or when an int value needs to be treated as an object.

5. What is the significance of final, finally, and finalize? Are they related?

In Java, **final**, **finally**, and **finalize** are three different keywords that have different meanings and uses:

final: The final keyword can be used to modify variables, methods, and classes. When used with a variable, final indicates that the value of the variable cannot be changed once it is initialized. For example:

```
final int x = 10;
x = 20; // This will cause a compile-time error
```

When used with a method, final indicates that the method cannot be overridden by subclasses. For example:

```
public class MyClass {
  public final void myMethod() {
    // Method code goes here
  }
}
public class MySubClass extends MyClass {
  public void myMethod() { // This will cause a compile-time error
    // Method code goes here
  }
}
```

When used with a class, final indicates that the class cannot be subclassed. For example:

```
public final class MyClass {
  // Class code goes here
}

public class MySubClass extends MyClass { // This will cause a compile-time error
  // Class code goes here
}
```

finally: The finally block is used in conjunction with the try and catch blocks in a try-catch-finally construct. The finally block is always executed, regardless of whether an exception is thrown or caught. It is typically used to clean up resources, such as closing open files or releasing connections.

finalize: The finalize method is a method that is defined in the Object class and is called by the garbage collector when it determines that an object is no longer reachable. The finalize method can be overridden in a subclass to perform cleanup tasks, such as releasing resources or closing open files. However, it is generally not recommended to rely on the finalize method for resource cleanup because it is not guaranteed to be called and because it can significantly impact the performance of the garbage collector.

In summary, final, finally, and finalize are unrelated and have different meanings and uses in Java. final is used to modify variables, methods, and classes, finally is used in a try-catch-finally construct to clean up resources, and finalize is a method that is called by the **garbage collector** for cleanup tasks

6. Explain the inheritance with Java code examples. Why multiple class inheritance is not supported in Java? Does this problem arise with Java 8 interface inheritance?

Inheritance in Java is the mechanism by which one class (called the subclass or child class) can inherit the properties and methods of another class (called the superclass or parent class). This allows you to create new classes that are built on top of existing classes, without having to rewrite the code that is already present in the existing classes. Here is an example of inheritance in Java:

```
public class Animal {
  protected String name;

  public Animal(String name) {
    this.name = name;
  }

  public void eat() {
    System.out.println(name + " is eating.");
  }
}

public class Cat extends Animal {
  public Cat(String name) {
    super(name);
  }
```

```
  public void meow() {
    System.out.println(name + " is meowing.");
  }
}

public class Main {
  public static void main(String[] args) {
    Cat cat = new Cat("Kitty");
    cat.eat(); // Output: "Kitty is eating."
    cat.meow(); // Output: "Kitty is meowing."
  }
}
```

In this example, the Cat class inherits the name field and the eat method from the Animal class. It also has its own method, meow, which is specific to the Cat class.

Multiple class inheritance, where a subclass can inherit from multiple superclasses, is not supported in Java. This is because multiple inheritance can lead to conflicts when multiple superclasses have methods with the same name and signature. For example, consider the following classes:

```
public class A {
  public void foo() {
    // Method code goes here
  }
}

public class B {
  public void foo() {
    // Method code goes here
  }
}

public class C extends A, B {
  // Class code goes here
}
```

In this example, the C class inherits from both A and B, which both have a foo method. This can lead to confusion when calling the foo method on an instance of C, as it is not clear which foo method should be called.

To overcome this problem, Java supports **single class inheritance**, where a subclass can only inherit from a single superclass. However, it allows a class to implement multiple interfaces, which can provide some of the benefits of multiple inheritance.

For example, consider the following interfaces:

```
public interface A {
  void foo();
}
```

```
public interface B {
  void foo();
}

public class C implements A, B {
  public void foo() {
    // Method code goes here
  }
}
```

In this example, the C class implements both A and B, which both have a foo method. However, since C is only implementing the interfaces, it does not have the same problem of multiple inheritance as it would if it were extending multiple classes.

7. What are the advantages of Java over C++? In which case you would prefer C++?

There are several advantages of Java over C++:

Java is easier to learn and use than C++, as it has a more intuitive syntax and fewer low-level details to worry about. **Java** is more portable than C++, as it runs on a virtual machine that abstracts away the underlying hardware and operating system. This means that Java code can run on any device that has a compatible Java runtime, without the need to recompile the code for each specific platform. **Java** has a built-in garbage collector that automatically reclaims memory that is no longer used by the program. This makes it easier to write memory-efficient programs in Java, as you do not have to worry about manually allocating and freeing memory. **Java** has a large, active community and a rich ecosystem of libraries and tools that make it easier to develop, test, and deploy applications. However, there are also some cases where C++ may be a better choice:

C++ is generally faster than Java, as it is a compiled language and does not incur the overhead of running on a virtual machine. This makes it a good choice for applications that require the highest possible performance, such as games or scientific simulations. **C++** has a larger set of low-level features, such as direct memory access and support for inline assembly, which can be useful for certain types of systems programming tasks. **C++** has a longer history and has been widely used for a variety of applications, which means that it has a larger codebase and a more established set of best practices and tools. In general, Java is a good choice for most applications that do not require the highest possible performance or the lowest-level control. It is particularly well-suited for web and enterprise applications, as well as Android mobile development. C++ is a good choice for applications that require the highest performance or the lowest-level control, such as games or operating systems.

8. Explain the difference between Error and Exception. Explain the difference between checked and unchecked exceptions.

In Java, an **Error is a subclass of Throwable** that indicates a serious problem that a reasonable application should not try to catch. An Error is typically thrown when a critical system error occurs, such as when the Java Virtual Machine (JVM) runs out of memory or when the system is in an inconsistent state.

An **Exception is also a subclass of Throwable**, but it indicates a less serious problem that an application might be able to recover from. An Exception is typically thrown when an error occurs that is not caused by a

critical system error, such as when an input file is not found or when an invalid argument is passed to a method.

In Java, exceptions are divided into two categories: **checked exceptions and unchecked exceptions**

Checked exceptions are exceptions that are checked at compile-time. This means that if a method throws a checked exception, the calling code must either handle the exception or declare that it throws the exception. This is enforced by the compiler, which will generate a compile-time error if the calling code does not handle or declare the exception.

Checked exceptions are typically used to indicate recoverable errors, such as input/output errors or resource exhaustion errors.

Here is an example of how to handle a checked exception:

```
try {
  // Code that might throw an exception goes here
} catch (IOException e) {
  // Handle the exception here
}
```

Here is an example of how to declare that a method throws a checked exception:

```
public void myMethod() throws IOException {
  // Code that might throw an exception goes here
}
```

Unchecked exceptions, on the other hand, are not checked at compile-time. This means that the calling code does not have to handle or declare unchecked exceptions. Unchecked exceptions are typically used to indicate programming errors, such as null pointer exceptions or index out-of-bounds exceptions.

Here is an example of an unchecked exception:

```
int[] arr = new int[5];
int x = arr[10]; // This will throw an ArrayIndexOutOfBoundsException
```

In summary, an **Error** is a serious problem that should not be caught, while an Exception is a less serious problem that an application might be able to recover from. Checked exceptions are enforced by the compiler and must be handled or declared, while unchecked exceptions are not checked at compile-time and do not have to be handled or declared.

9. What are fail-fast and fail-safe iterators? Give a code example.

In Java, an **iterator** is an object that allows you to iterate over a collection of elements, such as a list or a set. Iterators are used to access the elements of a collection one by one, without exposing the underlying implementation of the collection.

There are two types of iterators in Java: **fail-fast iterators** and **fail-safe iterators.**

Fail-fast iterators are iterators that throw a ConcurrentModificationException if the collection is modified while the iterator is in use. This is to prevent the iterator from returning inconsistent or invalid results due to the collection being modified by another thread.

Fail-fast iterators are typically used with collections that are not designed to be modified concurrently, such as ArrayList, Vector, and HashMap.

Here is an example of using a fail-fast iterator:

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
  String element = iterator.next();
  System.out.println(element);
  if (element.equals("B")) {
    list.remove("B"); // This will throw a ConcurrentModificationException
  }
}
```

Fail-safe iterators, on the other hand, do not throw a ConcurrentModificationException if the collection is modified while the iterator is in use. Instead, they work on a snapshot of the collection and do not reflect any changes made to the collection after the iterator was created.

Fail-safe iterators are typically used with collections that are designed to be modified concurrently, such as ConcurrentHashMap and CopyOnWriteArrayList.

Here is an example of using a fail-safe iterator:

```
Map<String, String> map = new ConcurrentHashMap<>();
map.put("A", "apple");
map.put("B", "banana");
map.put("C", "cherry");

Iterator<Map.Entry<String, String>> iterator = map.entrySet().iterator();
while (iterator.hasNext()) {
  Map.Entry<String, String> entry = iterator.next();
  System.out.println(entry.getKey() + ": " + entry.getValue());
  map.put("D", "date"); // This will not affect the iterator
}
```

In summary, fail-fast iterators throw a ConcurrentModificationException if the collection is modified while the iterator is in use, while fail-safe iterators do not throw an exception and work on a snapshot of the collection.

Fail-fast iterators are used with collections that are not designed to be modified concurrently, while fail-safe iterators are used with collections that are designed to be modified concurrently.

10.What is the difference between ObjectOutputStream and DataOutputStream? What is the significance of the Serializable interface?

In Java, **ObjectOutputStream** and **DataOutputStream** are two classes that are used for writing data to an output stream. They both inherit from the OutputStream class and provide methods for writing various data types, such as int, long, float, double, and String.

The main difference between **ObjectOutputStream** and **DataOutputStream** is that ObjectOutputStream can write Java objects to an output stream, while DataOutputStream can only write primitive data types and strings.

ObjectOutputStream works by serializing the object, which means that it converts the object's state into a byte stream that can be written to an output stream. The byte stream can later be deserialized back into an object using an ObjectInputStream.

To be serializable, a class must implement the Serializable interface, which is a marker interface that does not have any methods. This indicates to the ObjectOutputStream that the class's state can be serialized and deserialized.

Here is an example of using ObjectOutputStream to write an object to an output stream:

```
public class Person implements Serializable {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  // Getters and setters go here
}

// ...

Person person = new Person("John", 30);

try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.dat"))) {
  out.writeObject(person);
} catch (IOException e) {
  e.printStackTrace();
}
```

Here is an example of using DataOutputStream to write primitive data types and strings to an output stream:

```
try (DataOutputStream out = new DataOutputStream(new
FileOutputStream("data.dat"))) {
  out.writeInt(123);
  out.writeLong(456L);
  out.writeFloat(3.14f);
  out.writeDouble(2.718);
  out.writeUTF("Hello, world!");
} catch (IOException e) {
  e.printStackTrace();
}
```

In summary, **ObjectOutputStream** is used to write Java objects to an output stream, while **DataOutputStream** is used to write primitive data types and strings to an output stream. ObjectOutputStream requires the object to be serializable, while DataOutputStream does not have this requirement.

In Java, the **Serializable interface** is a marker interface that is used to indicate that a class's state can be serialized and deserialized. Serialization is the process of converting an object's state into a byte stream that can be written to an output stream, while deserialization is the process of converting the byte stream back into an object.

To make a class serializable, you simply need to make it implement the Serializable interface:

```
public class MyClass implements Serializable {
  // Class definition goes here
}
```

The **Serializable** interface does not have any methods, so you do not have to implement any methods when you make your class implement it.

Once a class is serializable, you can use an ObjectOutputStream to write the object to an output stream, and an ObjectInputStream to read the object back from the input stream.

Here is an example of serializing and deserializing an object using ObjectOutputStream and ObjectInputStream:

```
public class MyClass implements Serializable {
  // Class definition goes here
}

// ...

MyClass object = new MyClass();

// Serialize the object
try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("object.dat"))) {
  out.writeObject(object);
```

```
} catch (IOException e) {
  e.printStackTrace();
}

// Deserialize the object
try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("object.dat"))) {
  MyClass deserializedObject = (MyClass) in.readObject();
} catch (IOException | ClassNotFoundException e) {
  e.printStackTrace();
}
```

The **Serializable** interface is useful because it allows you to easily save and restore the state of an object by writing and reading it to and from an output stream. It is also useful for transferring objects over a network or for storing objects in a database.

One important thing to note is that the Serializable interface does not include any mechanism for versioning. This means that if you change the class definition after serializing an object, you may not be able to deserialize the object correctly. To handle this situation, you can use the Externalizable interface, which allows you to control the serialization process and implement versioning.

11.How to search a user-defined class object into an ArrayList? Explain with a code example.

To search for an object of a user-defined class in an ArrayList, you can use the contains() method provided by the ArrayList class. This method checks if the specified element is present in the list.

Here is an example of how you can use the contains() method to search for an object in an ArrayList:

```
import java.util.ArrayList;

class Student {
   private String name;
   private int age;

   // Constructor and other methods go here...
}

public class Main {
   public static void main(String[] args) {
      // Create an ArrayList to store Student objects
      ArrayList<Student> students = new ArrayList<>();

      // Add some Student objects to the list
      students.add(new Student("Alice", 20));
      students.add(new Student("Bob", 21));
      students.add(new Student("Charlie", 22));

      // Create a Student object to search for
      Student s = new Student("Bob", 21);

      // Search for the Student object in the list
```

```
        if (students.contains(s)) {
            System.out.println("Student found in the list!");
        } else {
            System.out.println("Student not found in the list!");
        }
    }
}
```

In this example, we have a Student class with a name and age field, and a constructor to create a Student object. We create an ArrayList of Student objects and add some students to the list. Then we create a Student object to search for, and use the contains() method to check if it is present in the list.

Note that the contains() method uses the equals() method to compare elements in the list. Therefore, you need to override the equals() method in your Student class for it to work correctly. You can do this by adding the following code to your Student class:

```
@Override
public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof Student)) return false;
    Student s = (Student) o;
    return s.name.equals(name) && s.age == age;
}
```

This implementation of the equals() method compares the name and age fields of two Student objects to determine if they are equal.

12.What is the role of @FunctionalInterface? Explain important built-in functional interfaces.

The **@FunctionalInterface** annotation is used to specify that an interface is a functional interface, which is an interface that has exactly one abstract method. A functional interface can be used as the target type for a lambda expression or method reference.

Functional interfaces are useful in Java because they enable you to create simple, concise code that represents a single function. They are used extensively in the Java Streams API, which provides a way to process collections of data in a declarative, functional style.

Here are some important built-in functional interfaces in Java:

java.util.function.Consumer: Represents an operation that accepts a single input argument and returns no result. java.util.function.Supplier: Represents a supplier of results. java.util.function.Function<T, R>: Represents a function that takes an argument of type T and returns a result of type R. java.util.function.Predicate: Represents a predicate (a boolean-valued function) of one argument. java.util.function.BiFunction<T, U, R>: Represents a function that takes two arguments and returns a result. Here is an example of how you can use a functional interface to create a lambda expression:

```
import java.util.function.Function;
```

```
public class Main {
   public static void main(String[] args) {
      // Create a Function that takes a string and returns its length
      Function<String, Integer> stringLengthFunction = s -> s.length();

      // Use the apply() method to apply the function to a string
      int length = stringLengthFunction.apply("Hello");
      System.out.println("Length: " + length);
   }
}
```

In this example, we create a Function that takes a string and returns its length. We then use the apply() method to apply the function to a string and print the result.

13.What are the characteristics of Stream? Produce a stream of 1 to 100 numbers, filter all even numbers, calculate their squares and sum them all.

A **stream** in Java is a sequence of elements that supports various operations for processing elements. Some characteristics of streams are:

A stream does not store its elements. It simply provides a way to process them. A stream does not modify the source of elements. It operates on the elements and produces a new stream or a result. A stream is lazy. It does not perform any operations until a terminal operation is called. Here is an example of how you can create a stream of numbers from 1 to 100, filter all even numbers, calculate their squares, and sum them all:

```
import java.util.stream.IntStream;

public class Main {
   public static void main(String[] args) {
      int sum = IntStream.rangeClosed(1, 100)
                         .filter(n -> n % 2 == 0)
                         .map(n -> n * n)
                         .sum();
      System.out.println("Sum: " + sum);
   }
}
```

In this example, we use the rangeClosed() method of the IntStream class to create a stream of integers from 1 to 100. Then we use the filter() method to keep only the even numbers in the stream, the map() method to calculate the squares of the numbers, and the sum() method to add them all together. The result is printed to the console.

Note that the sum() method is a terminal operation, which means it triggers the execution of the stream pipeline and terminates the stream. All intermediate operations such as filter() and map() are lazily executed, and their results are passed to the terminal operation.

14.What are the characteristics of the equals() and hashCode() method? Which Java collection needs these methods to be overridden?

The **equals()** and **hashCode()** methods are used to determine the equality of objects and to create hash-based collections. These methods are defined in the Object class, which is the superclass of all Java classes.

Here are some characteristics of the equals() and hashCode() methods:

The equals() method is used to compare the contents of two objects. It returns true if the objects are considered equal, and false otherwise. The hashCode() method is used to generate a numerical value, called a hash code, for an object. The hash code is used to index objects in hash-based collections such as HashMap and HashSet. The equals() and hashCode() methods should be consistent with each other. This means that if two objects are considered equal according to the equals() method, they should have the same hash code. It is important to override the equals() and hashCode() methods correctly in your classes if you want to use them as keys in hash-based collections. The HashMap and HashSet classes, for example, use the hashCode() method to determine where to store an object in the collection and the equals() method to compare objects for equality.

To override the equals() and hashCode() methods correctly, you should follow these guidelines:

Override the equals() method to compare the contents of two objects, rather than their references. Override the hashCode() method to generate a hash code that depends on the contents of the object. If you override the equals() method, you should also override the hashCode() method. Here is an example of how you can override the equals() and hashCode() methods in a class:

```
class Person {
   private String name;
   private int age;

   // Constructor and other methods go here...

   @Override
   public boolean equals(Object o) {
      if (o == this) return true;
      if (!(o instanceof Person)) return false;
      Person p = (Person) o;
      return p.name.equals(name) && p.age == age;
   }

   @Override
   public int hashCode() {
      int result = 17;
      result = 31 * result + name.hashCode();
      result = 31 * result + age;
      return result;
   }
}
```

In this example, we override the equals() method to compare the name and age fields of two Person objects. We also override the hashCode() method to generate a hash code based on the name and age fields of the object.

15.How to create a thread in Java using Lambda expression? What are other ways to create the threads? What is relation between start() and run() methods?

To create a thread in Java using a lambda expression, you can use the Runnable interface and pass a lambda expression as the implementation of the run() method. Here is an example of how you can create a thread using a lambda expression:

```
public class Main {
   public static void main(String[] args) {
      // Create a thread using a lambda expression
      Thread t = new Thread(() -> {
         // Code to run in the thread goes here
      });

      // Start the thread
      t.start();
   }
}
```

In this example, we create a Thread object and pass a lambda expression as the implementation of the run() method. The lambda expression defines the code that will be executed in the thread. We then call the start() method on the Thread object to start the thread.

There are several other ways to create threads in Java:

You can create a class that implements the Runnable interface and pass an instance of that class to the Thread constructor. You can create a class that extends the Thread class and override the run() method. The start() method is used to start a new thread and execute the code in the run() method. The run() method is the entry point for the new thread, and it defines the code that will be executed in the thread.

The start() method is different from the run() method in that it creates a new thread and starts it, while the run() method simply executes the code in the current thread. When you call the start() method on a Thread object, it creates a new thread and starts it, and the run() method is executed in the new thread. If you call the run() method directly, it simply executes the code in the current thread, without creating a new thread.

Here is an example of how you can create a thread by extending the Thread class:

```
public class MyThread extends Thread {
   @Override
   public void run() {
      // Code to run in the thread goes here
   }
}

public class Main {
   public static void main(String[] args) {
      // Create a MyThread object
      MyThread t = new MyThread();
```

```
        // Start the thread
        t.start();
    }
}
```

In this example, we create a MyThread class that extends the Thread class and overrides the run() method. We then create an instance of MyThread and call the start() method to start the thread.

Q: What is a Wrapper Class? What is their use?

A: A Wrapper Class is a class that wraps around a primitive data type and provides additional functionality such as converting data types, formatting data, and implementing data structures. The main use of Wrapper Classes is to allow primitive data types to be used in Java's collection classes.

Q: Which are the methods of java.lang.Object class? Which are the native methods of Object class?

A: The methods of the java.lang.Object class include toString(), equals(), hashCode(), and clone(). The native methods of the Object class include registerNatives(), getClass(), hashCode(), wait(), notify(), notifyAll(), and finalize().

Q: What is the need for a package? Which types are allowed to declare in a package?

A: A package is used to organize related classes and interfaces in Java. It helps in reducing naming conflicts and makes it easier to locate and manage classes. The following types can be declared in a package: classes, interfaces, enumerations, and annotations.

Q: Why can we not declare multiple public classes in a single Java file?

A: In Java, only one public class can be declared in a single file, and the name of the file must match the name of the public class. This is because the Java compiler needs to know which class to use as the entry point when executing the program.

Q: What is the difference between import and static import?

A: The import statement is used to make classes and interfaces available in other classes. The static import statement is used to import the static members of a class directly into the importing class.

Q: How can we pass an argument to the method by reference? Explain with an example.

A: In Java, it is not possible to pass arguments to a method by reference, as Java only supports pass-by-value. However, we can achieve similar functionality by passing an object to a method, and modifying the state of the object within the method.

For example, consider the following code:

```
Copy code
public class PassByReferenceExample {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println("Value before method call: " + obj.value);
        changeValue(obj);
```

```
        System.out.println("Value after method call: " + obj.value);
    }

    public static void changeValue(MyClass obj) {
        obj.value = 10;
    }
}

class MyClass {
    public int value = 5;
}
```

In this example, we pass an instance of MyClass to the changeValue() method, and modify the value of the object within the method.

Q: What is the difference between checked and unchecked exceptions?

A: Checked exceptions are those which are checked at compile time and the code must handle or declare them. Examples of checked exceptions include IOException and SQLException. Unchecked exceptions, on the other hand, are not checked at compile time and are thrown at runtime. Examples of unchecked exceptions include NullPointerException and ArrayIndexOutOfBoundsException.

Q: What are the advantages and disadvantages of generics?

A: The advantages of using generics include type safety, code reuse, and performance improvements. The disadvantages include increased complexity and decreased readability.

Q: What is the difference between Comparable and Comparator?

A: Comparable is an interface that allows objects to be compared with other objects of the same class. The comparison logic is implemented in the compareTo() method. Comparator is an interface that allows objects to be compared with other objects of a different class. The comparison logic is implemented in the compare() method.

What is the difference between ArrayList and Vector? How to use ListIterator?

Both ArrayList and Vector are used to store a collection of elements. The main differences between them are:

Synchronization: Vector is synchronized while ArrayList is not, which means that multiple threads can access an ArrayList simultaneously, but only one thread can access a Vector at a time.

Growth rate: Vector increases its size by 100% while ArrayList increases its size by 50% when it reaches its maximum capacity.

Performance: ArrayList is faster than Vector since it is not synchronized.

ListIterator is an interface that provides methods to traverse a list in both forward and backward directions, to modify the list during iteration, and to obtain the current position in the list. Here's an example of how to use it:

```
List<String> list = new ArrayList<>();
list.add("apple");
list.add("banana");
list.add("cherry");

ListIterator<String> iterator = list.listIterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}

while (iterator.hasPrevious()) {
    System.out.println(iterator.previous());
}
```

What is fail-fast and fail-safe iterator? In Java, an iterator is used to traverse a collection of elements. There are two types of iterators: fail-fast and fail-safe.

Fail-fast iterators: These iterators throw a ConcurrentModificationException if a collection is modified while it is being traversed. They are designed to fail quickly to prevent the modification of the collection from causing problems in other parts of the code. Examples of fail-fast iterators are the iterators of ArrayList, HashSet, and HashMap.

Fail-safe iterators: These iterators create a copy of the collection before traversing it, so any modifications to the original collection will not affect the iterator. They are designed to be safe in multi-threaded environments. Examples of fail-safe iterators are the iterators of ConcurrentSkipListMap and CopyOnWriteArrayList.

What is serialization and deserialization? What is the significance of SerialVersionUID? Serialization is the process of converting an object into a stream of bytes, which can be saved to a file, sent over a network, or stored in a database. Deserialization is the reverse process of converting a stream of bytes back into an object. In Java, objects can be serialized by implementing the Serializable interface.

The serialVersionUID is a unique identifier for a serialized object. It is used to ensure that the deserialization process is compatible with the serialization process. If the serialVersionUID of the serialized object is different from the serialVersionUID of the deserialized object, an InvalidClassException will be thrown.

What is the relation between Thread start() and run() method? In Java, threads can be created by extending the Thread class or implementing the Runnable interface. The start() method is used to start a new thread, which calls the run() method. The run() method contains the code that will be executed by the thread. The start() method creates a new thread and runs it in a separate call stack, while the run() method executes on the current call stack.

When should we create a thread by implementing Runnable and extending Thread class? We can create a thread in Java by extending the Thread class or implementing the Runnable interface. It is recommended to implement the Runnable interface since it separates the task from the thread, which is a better design practice. Additionally, by implementing Runnable, the class can extend another class or implement another interface.

What is functional interface? Which functional interfaces are predefined and where they are used? What is significance of filter(), map(), flatMapO and reduceO operations? In which scenarios they are used ?

A functional interface is an interface that contains exactly one abstract method, and is typically used to define a lambda expression. In Java 8 and later versions, functional interfaces are extensively used to implement functional programming concepts in Java.

Java provides several predefined functional interfaces, including:

Predicate: takes an argument and returns a boolean value. Function: takes an argument and returns a result. Supplier: takes no arguments and returns a result. Consumer: takes an argument and returns no result.

The significance of filter(), map(), flatMap(), and reduce() operations are as follows:

filter(): is used to filter the elements of a stream based on a given predicate.

map(): is used to transform the elements of a stream into a new stream by applying a function to each element.

flatMap(): is used to transform a stream of streams into a single stream.

reduce(): is used to reduce the elements of a stream to a single value by applying a given function.

These operations are commonly used in functional programming paradigms and are particularly useful in scenarios where large amounts of data need to be processed efficiently. For example, they can be used for data manipulation in big data or machine learning applications, as well as for processing collections of data in general.

Lambda Expression a lambda expression in Java is a way to write a small piece of code that can be used later. It's like writing a recipe or a set of instructions for someone else to follow.

When you write a lambda expression, you start by saying what kind of inputs you need (like "two numbers"), and then you say what you want to do with those inputs (like "add them together").

Lambda expressions are useful because they allow you to write code that can be easily reused in different parts of your program. They can be passed around like objects, so you can use them in different methods and classes without having to rewrite the same code over and over again.

```
(int a, int b) -> a + b

Calculator calculator = (int a, int b) -> a + b;

int result = calculator.calculate(3, 4); // result = 7
```