Kris Moreira  
Technical Recuriter  
703-570-8981  
[kris.moreira@revature.com](mailto:kris.moreira@revature.com)

Core Java

### Interview Questions

* What is new in Java 8 with interfaces? : default method which is a concrete method
* How can I access non-static fields inside of static methods?

What is the main method in Java? main method is the entry point of any java program. You can only change the name of String array argument.

What are control statements why are they useful? Break, continue : **Control flow statements** **break** up the flow of execution, enabling your program to conditionally **execute** particular blocks of code.

What kinds of variables can I use with switch statements? **byte**, **short**, **char**, and **int** primitive data types. It also works with enumerated types, the **String** class, and wrapper class: **Character**, **Byte**, **Short**, and **Integer**

**Objects** have states and behaviors. An **object** is an instance of a class, has a set of methods.

**State and behavior**: state tells us about the type or the value of that object whereas behavior tells us about the operations or things that the object can perform.

**A class** describes a set of objects with same behavior, a blueprint for objects.

An **object reference** specifies the location of an object. The null reference refers to no object.

Primitive type variables stores values. Object variables store reference.

**Primitive data types**

* **int** a; // number type - 4 bytes
* **boolean** b; // T/F
* **char** c; // %afd343' - 2 bytes
* **double** d; // decimal - 8 bytes
* **float** f; // flating decimal - 4 bytes
* **long** w; // large numer - 8 bytes
* **short** g; // small numbers - 2 bytes
* **byte** h; // //1byte

**Immutable: Mutable** object – You can change the states and fields after the object is created. For **examples**: StringBuilder , **java**.util.Date and etc. ... Immutable object – You cannot change anything after the object is created. For **examples**: String, boxed primitive objects like Integer, Long and etc.

A **constructor in Java** is a block of code similar to a method that's called when an instance of an object is created. A **constructor** doesn't have a return type. ... Unlike methods, **constructors** are not considered members of a class.

**Inheritance:** is a mechanism in which one object acquires all the properties and behaviors of a parent object.

**Abstraction** : is a process of hiding the implementation details from the user. Оnly the functionality will be provided to the user. In **Java**, **abstraction** is achieved using abstract classes and interfaces

Abstract Classes: We cannot create instance for an abstract class. We can be able to create instance for its subclass only.

public abstract class MyAbstractClass{

}

public abstract int getLength();

1.Main **difference** is methods of a **Java interface** are implicitly **abstract** and cannot have implementations. A **Java abstract class** can have instance methods that implements a default behavior. 2.Variables declared in a **Java interface** is by default final. An **abstract class** may contain non-final variables.

**Interface:**

**public** **interface** Measurable {

**double** getMeasure();

default void talk(“Hello”);

}

**Encapsulation**: is the act of providing a public interface and hiding the implementation details.

* Object fields
* Private instance vars,
* Public getter (accessor methods) & setters(mutator methods)

**Polymorphisms** Ability for objs to behave in various manners

* Method overloading: same name, different signature (same class)
* Method overriding: same name, different implementation (inherited classes)

Covariant types ie. Human h = new Adult();

**Package:** A package is a namespace that organizes a set of related classes and interfaces. You can think of packages as being similar to different folders on your computer. Sub packages have no special connection to parent packages. i.e com.ex.\* will not import com.ex.sub.ClassName

**OOP**: is a **programming** language model organized around **objects** rather than "actions" and data rather than logic.

**A Collection** groups together elements and allows them to be retrieved later. Java collection is a hierarchy of interface types and classes for collecting objects. List, Queue, ArrayList, Stack, LinkedList.

Why use java?

1. write once, run anywhere

2. portable

3. free

.widely supported

.open source

rich API

automatic memory management

JVM: compile java, java virtual machine

JRE: java runtime environment, execute

JDK: Java development kit

//[access modifier] [non-access modifier] class [ClassName]{}

strick type

case sensitive

java c helloworld.java compile

java HelloWorld execute

java HelloWorld "these are parameters"

**Wrapper class**

Auto-boxing: Conversion between primitive types and the corresponding wrapper class is automatic.

if you assign a double value to a Double variable, the number is automatically put into a box.

Double wrapper = 29.95;

Conversely, wrapper values are automatically unboxed to primitive types.

double x = wrapper;

**Casting:** to convert a value to a different type.

double balance = 12.75;

int dollar = (int) (balance + tax); lost faction

dollar = 12;

or int ronded = (int) Math.round(balance);

rounded = 13;

**Operators**

are special symbols performing operations on one, two, or three operands & returning a result.

Java operators are classified into categories:

Assignment =, +=, ==, \*, /, %

Arithmetic +, -, / , \*, %, x++, --xx

Relational ==, !=, >

Logical/short circuit (&&, ||)

Bitwise (&, |, )

Shift >>, <<

Ternary (exp? if true : if false)

Type comparison (instance of)

**Strings Class**

java.lang.String

Java.lang is only package in Java that we can access classes of directly without the need of import

Special objects in Java

Not the same thing as Character[]

String literal =" " can filled w >= char

String are immutable, cannot change its contents after created.

String can be initialized in 2 ways

String st="hello", String str2 = new String("hi");

gets stored in the String pool: String st="hello",

gets stored as object in the heap : String str2 = new String("hi");

**String builder**-mutable Strings: fast, not synchronized

**String buffered**: thread safe, mutable Strings, slower but synchronized.

**Memory in Java: Stack memory, heap space**

Java heap space is used by Java runtime to allocate memory to objs & JRE classes. Garbage collection is used to free memory from any unreferenced objs. Contains the String pool.

Java stack memory: used for execution of a thread, or a single chronological process. It also contains local (method specific) values & referenced to objs in the heap being referred to in that method.

Stack memory is always referenced in LIFO order. Whenever a method is invoked, a block is created in the stack memory related to stack trace. As soon as the method ends, the block becomes unused & available for the next method. Stack memory is smaller in size than heap space.

What is the **String pool**? An area in heap memory where Java store literal String values.

**Access modifiers/levels**

Determine where, from outside of a class, an entity can be accessed.

Public – accessible everywhere

Protected – accessible within same package & from subclasses

Private-only accessible within the class

it is declared in [no modifier] aka “default” or “package” – only accessible within same package.

An object’s instance variables store the data required for excuting its methods. An object stores its data in instance variables. An instance of a class is an object of the class.

**Variable scopes:** scope of a variable is the part of the program in which it is visible.

. Instance / object

. static / class

.local / method

. block scope

variable scopes refer to the lifetime of accessibility of a variable.

there are 4 scopes

1. static/class (var, methods, nested classes) – belongs to the class, not to any object of the class, and accessible from outside of the class without an instance ie. Class.x or Class.method();

Regarding static variables, these values are shared throughout any instances of the class.

class Test {static int count=0;}

\*

\* from outside test;

\*

\* Test c = new Test();

\* Test c2 = new Test();

\* c.count = 10;

\* .then c2.count would also be 10;

\*

2. object/instance - the particular instance of the objects

\* state and behavior

\* Class c = new Class();

\* c.x; or c.method();

\*

3. method/local - parameters for method and any variable declare within the method

4. block/loop - any variable defined in curly braces or in the () before loop

An **array** stores a sequence of values of the same type.

An **Array List** stores a sequence of values whose size can change.

**Static methods**: methods that are not invoked on objects. Math.pow, primitive types are not objects.

**Instance methods**: methods that are invoked on objects.

**Exception & errors**: things that go wrong at runtime – unlike syntax errors & other issues that will prevent successful compilation.

**Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**

**Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise.

*Throws(after method)* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

**Unit Testing**

Most granular form of software testing.

Evaluating smallest executable pieces of code (methods)

Performed by programmer – not team of external testers, should always think “TDD” – Test Driven Development.

Junit – popular Java Unit Testing framework, must use external jar for org.junit packages.

**Annotations:**

**@Test – indicates that the following method is a unit test**

**@Before @ After – run before/after each @Test method**

**@BeforeClass/@AfterClass - run before/after all @Test methods in the class are executed.**

**Assert methods – methods used to define expected behavior & indicate success/failure.**

**@Runwith, @SuiteCases : Test Suit**

**@Override**

**@Functional interface**

Interfaces

* An abstract type that is used to define behavior that subclasses must implement “Contract”
* Classes may implement multiple interfaces -> methods are implicitly public & abstract.
* Since Java 1.8, methods CAN be implemented in interfaces, they must be labeled with the default keyword.
  + All variables are implicitly public static final.
  + “Abstract” keyword – apply to a method to require it to be overridden in subclass.
  + Related functional interfaces marker interfaces, multiple inheritance.

**Abstract Classes**

* May have both concrete & abstract methods.
* Cannot be instantiated.

Final keyword:

-Final class – cannot be extended.

-Final method – cannot be overridden. CAN be overloaded.

-Final variables – cannot be reassigned constant values. i.e final static int MAX\_VALUE = 100;

* **Threading** multiple flows of control, each thread gets its own **stack**
* Thread class
  + worker
* Runnable interface
  + job

-Thread – in the context of Java, a thread is the path followed when executing a program. All Java programs have at least one thread known as the main thread, which is created by the JVM at the program’s start, when the main() method is invoked.

-Multithreading – multiple flows of controls / threads of execution in program execution.

-Each thread gets its own stack.

-Create a separate thread by extending the Thread class or by implementing the Runnable interface overriding the run() method of the Runnable interface, then calling the start() method of the Thread class.

Thread class -> worker

Start();

Runnable interface -> job

Run();

JVM Thread scheduler – as a developer, you don’t have complete control over thread execution, But you can influence via thread priority (1-10).

Thread States: New, RUNNABLE, BLOCKED, WAITING, TIMED-WAITING, & TERMINATED.

Types of threads:

* User-defined: main + any other created threads.
* Daemon – background processes. i.e Garbage collector.

Related: synchronized keyword, deadlock, starvation, produce, consumer problem.

Design patterns are established general solutions to commonly occurring software development problems.

The singleton design pattern is a pattern which creates an object in memory only once throughout the lifetime of the application to be shared across multiple classes.

\* To make a singleton you must make a private static variable of the class's type, make a private constructor, and make a public static getInstance() method, which will return the single instance of the class created.

\* There are two types of singletons

\* Lazy Singleton

\* Eager Singleton

The factory design pattern is a pattern which creates objects where the precise type may not be known until runtime (create a reference to a parent class or interface but instantiate with the particular implementing class based on some sort of user input)

\* To make a factory we need:

\* -abstract data type

\* -classes which inherit the abstract data type

\* -static method that returns a concrete instance of the abstract data type

|  |
| --- |
| public class DessertFactory { |
|  | /\* |
|  | \* The factory design pattern is a pattern which |
|  | \* creates objects where the precise type may not |
|  | \* be known until runtime (create a reference to a |
|  | \* parent class or interface but instantiate with |
|  | \* the particular implementing class based on some |
|  | \* sort of user input) |
|  | \* |
|  | \* To make a factory we need: |
|  | \* - abstract data type |
|  | \* - classes which inherit the abstract data type |
|  | \* - static method that returns a concrete instance |
|  | \* of the abstract data type |
|  | \*/ |
|  |  |
|  | public static Dessert getDessert(String type) { |
|  | switch(type) { |
|  | case "cake": |
|  | return new Cake(); |
|  | case "cookie": |
|  | return new Cookie(); |
|  | case "brownie": |
|  | return new Brownie(); |
|  | default: |
|  | throw new DessertNotFoundException(); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | //Good practice to throw an exception if the desired concrete class is not found |
|  | class DessertNotFoundException extends RuntimeException{} |
|  |  |
|  | //abstract type |
|  | interface Dessert{ |
|  | String makeMe(); |
|  | } |
|  |  |
|  | class Cake implements Dessert{ |
|  |  |
|  | @Override |
|  | public String makeMe() { |
|  | return "bake a cake with lots of sugar and things"; |
|  | } |
|  | } |
|  |  |
|  | class Cookie implements Dessert{ |
|  |  |
|  | @Override |
|  | public String makeMe() { |
|  | return "chocolate chip"; |
|  | } |
|  | } |
|  |  |
|  | class Brownie implements Dessert{ |
|  |  |
|  | @Override |
|  | public String makeMe() { |
|  | return "yum"; |
|  | } |
|  |  |
|  | } |

|  |
| --- |
| public class Singleton { |
|  | /\* |
|  | \* The Singleton design pattern is a pattern which creates |
|  | \* an object in memory only once throughout the lifetime of |
|  | \* the application to be shared across multiple classes |
|  | \* |
|  | \* To make a Singleton you must make a private static variable |
|  | \* of the class's type, make a private constructor, and make a |
|  | \* public static getInstance() method, which will return the single |
|  | \* instance of the class created. |
|  | \* |
|  | \* There are two types of singletons. |
|  | \* Lazy Singleton |
|  | \* Eager Singleton |
|  | \*/ |
|  |  |
|  | private static Singleton singleton = new Singleton(); |
|  | private String name; //instance var to demonstrate single instance |
|  |  |
|  | private Singleton() { |
|  | System.out.println("instantiating eager singleton"); |
|  | } |
|  |  |
|  | public static Singleton getInstance() { |
|  | return singleton; |
|  | } |
|  |  |
|  | public String getName() { |
|  | return name; |
|  | } |
|  |  |
|  | public void setName(String name) { |
|  | this.name = name; |
|  | } |
|  |  |
|  | public static void test() { |
|  | System.out.println("in eager singleton"); |
|  | } |
|  | } |
|  |  |
|  | //Lazily instantiate single instance of this class |
|  | class LazySingleton{ |
|  |  |
|  | //JUST declaration, NOT instantiation |
|  | private static LazySingleton instance; |
|  |  |
|  | private LazySingleton() { |
|  | System.out.println("Lazily instantiating singleton"); |
|  | } |
|  |  |
|  | public static LazySingleton getInstance() { |
|  | if(instance == null) { |
|  | instance = new LazySingleton(); |
|  | } |
|  | return instance; |
|  | } |
|  |  |
|  | public static void test() { |
|  | System.out.println("in lazy singleton class"); |
|  | } |

What make String class immutable? caching, security, synchronization, and performance

Caching the *String* literals and reusing them saves a lot of heap space because different *String* variables refer to the same object in the *String* pool.

The *String* is widely used in Java applications to store sensitive pieces of information like usernames, passwords, connection URLs, network connections,

Being immutable automatically makes the *String* thread safe since they won’t be changed when accessed from multiple threads.

== checks if the memory address is the same

Equals check if same content.

* What is Maven? **Maven** is a build automation tool used primarily for **Java** projects. **Maven** addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. Java build tool
* manage dependencies
* build our project

POM: Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

**Reflection** is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime. The required classes for **reflection** are provided under **java**.lang.reflect package. ... Through **reflection** we can invoke methods at runtime irrespective of the access specifier used with them.

class Person implements Comparable<Person> {

String name;

int age;

@Override

public int compareTo(Person other) {

return this.age - other.age;

}

}

public class Main {

public static void main(String[] args) {

List<Person> list = new ArrayList<>();

list.add(new Person("john", 28));

list.add(new Person("amy", 25));

list.add(new Person("austin", 33));

System.out.println(list);

Collections.sort(list);

System.out.println(list);

}

}

class PersonNameComparator implements Comparator<Person> {

@Override

public int compare(Person one, Person two) {

return one.name.compareTo(two.name);

}

}

public class Main {

public static void main(String[] args) {

List<Person> list = new ArrayList<>();

list.add(new Person("john", 28));

list.add(new Person("amy", 25));

list.add(new Person("austin", 33));

System.out.println(list);

Collections.sort(list, new PersonNameComparator());

System.out.println(list);

}

}

Collection:

|  |
| --- |
| public class UnderstandingCollections { |
|  |  |
|  | /\* |
|  | \* In Java, a Collection is a group of individual objects represented |
|  | \* as a single unit. We use them to store and organize our groups of |
|  | \* objects. Java's collection Framework is extensive and has several |
|  | \* classes and interfaces with different properties and behaviors to |
|  | \* group our entities together. |
|  | \* |
|  | \*/ |
|  |  |
|  | public static void main(String[] args) { |
|  | arrayExample(); |
|  | arrayListExample(); |
|  | setExample(); |
|  | mapExample(); |
|  | queueExample(); |
|  | } |
|  |  |
|  | public static void arrayExample() { |
|  | // arrays can be created in two different ways |
|  | String[] stringArr = {"you can","create an array", "like this"}; |
|  | String[] stringArr2 = new String[] {"or","like","this"}; |
|  | // the size of an array cannot be changed, but it can be iterated over |
|  | for (String s : stringArr) { |
|  | System.out.print(s+" "); |
|  | } |
|  | for (String s : stringArr2) { |
|  | System.out.print(s+" "); |
|  | } |
|  | System.out.println(""); |
|  | } |
|  |  |
|  | public static void arrayListExample() { |
|  | // an arrayList can add elements dynamically, growing and shrinking as needed |
|  | List<String> list = new ArrayList<>(); |
|  | list.add("first string"); |
|  | list.add("second string"); |
|  | list.add(new String("third string")); |
|  | // you can manipulate and access elements of arrayLists |
|  | // with .add, .remove, .get, .size |
|  | int size = list.size(); |
|  | String first = list.get(0); |
|  | System.out.println("1: "+first); |
|  | String alsoFirst = list.remove(0); |
|  | System.out.println("are they the same?" + alsoFirst == first); |
|  | int sizeAfterRemoval = list.size(); |
|  | System.out.println("this should be false: "+ (sizeAfterRemoval == size)); |
|  | } |
|  |  |
|  | public static void setExample() { |
|  | // hashsets are sets (no duplicates) backed by a hash table |
|  | // there is no guarantee about the order of elements in a hashset |
|  | Set<String> set = new HashSet<>(); |
|  | System.out.println("adding a string to the set"); |
|  | set.add("hello"); |
|  | System.out.println("attempting to add identical string"); |
|  | set.add("hello"); |
|  | // the set size should still be 1 because duplicates are not allowed |
|  | System.out.println("set size: "+set.size()); |
|  | set.add("different string"); |
|  | System.out.println("after adding different string, the set size is now: "+set.size()); |
|  | } |
|  |  |
|  | public static void mapExample() { |
|  | // Maps map keys to values, in this case the specific implementation is a hashmap |
|  | Map<Integer, String> map = new HashMap<>(); |
|  | System.out.println("entering values into map"); |
|  | map.put(1, "hello"); |
|  | map.put(2, "goodbye"); |
|  | System.out.println("retrieving values from map:"); |
|  | String val1 = map.get(1); |
|  | String val2 = map.get(2); |
|  | System.out.println("val1: "+val1); |
|  | System.out.println("val2: "+val2); |
|  | } |
|  |  |
|  | public static void queueExample() { |
|  | Queue<String> q = new ArrayDeque<>(); |
|  | q.add("first in the queue"); |
|  | q.add("second in queue"); |
|  | Iterator<String> i = q.iterator(); |
|  | while (i.hasNext()) { |
|  | System.out.println(i.next()); |
|  | } |
|  | } |
|  |  |
|  | } |