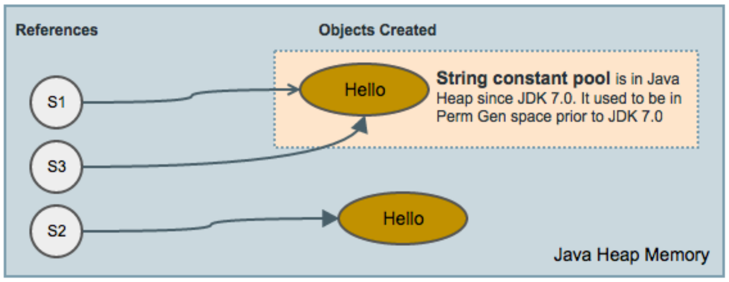
♥♦ Q1-Q10: Top 50+ Core Java Interview Questions and Answers

Java interview questions and answers with detailed diagrams & working code to fast-track your Java interview preparation.

Q1. What is the difference between "==" and "equals(…)" in comparing Java String objects?
A1. When you use "==" (i.e. shallow comparison), you are actually comparing the two object references to see if they point to tl
same object. When you use "equals(…)", which is a "deep comparison" that compares the actual string values. For example:

```
public class StringEquals {
 public static void main(String[ ] args) {
   String s1 = "Hello";
   String s2 = new String(s1);
   String s3 = "Hello";

   System.out.println(s1 + " equals " + s2 + "--> " +  s1.equals(s2)); //true

   System.out.println(s1 + " == " + s2 + " --> " + (s1 == s2)); //false
   System.out.println(s1 + " == " + s3+ " --> " + (s1 == s3)); //true
 }
}

```

The variable **s1** refers to the String instance created by "Hello". The object referred to by **s2** is created with s1 as an initializer, tl
the contents of the two String objects are identical, but they are 2 distinct objects having 2 distinct references s1 and s2. This
means that s1 and s2 do not refer to the same object and are, therefore, not ==, but *equals( )* as they have the same value "Hel
The s1 == s3 is true, as they both point to the same object due to internal caching. The references s1 and s3 are **interned** and
points to the same object in the string pool.


References
Objects Created
S1
S3
S2
Hello
Hello
String constant pool is in Java Heap since JDK 7.0. It used to be in Perm Gen space prior to JDK 7.0
Java Heap Memory

Create a String object as a literal without the "new" keyword for caching

In Java 6 — all interned strings were stored in the **PermGen** – the fixed size part of heap mainly used for storing loaded classes
and string pool.

In Java 7 – the string pool was relocated to the **heap**. So, you are not restricted by the limited size.

How about comparing the other objects like Integer, Boolean, and custom objects like "Pet"? Object **equals** Vs "**==**", and pass by
reference Vs **value**.

Q2. Can you explain how Strings are interned in Java?
A2. String class is designed with the **Flyweight design pattern** in mind. Flyweight is all about re-usability without having to crea
too many objects in memory.

[**Further Reading:** Flyweight pattern and improve memory usage & performance]

A pool of Strings is maintained by the String class. When the *intern( )* method is invoked, equals(..) method is invoked to determine if the String already exist in the pool. If it does then the String from the pool is returned instead of creating a new object. If not already in the string pool, a new String object is added to the pool and a reference to this object is returned. For a two given strings s1 & s2, *s1.intern( )* == *s2.intern( )* only if *s1.equals(s2)* is true.

Two String objects are created by the code shown below. Hence s1 == s2 returns false.

```
//Two new objects are created. Not interned and not recommended.
String s1 = new String("A");
String s2 = new String("A");

```

s1.intern() == s2.intern() returns **true**, but you have to remember to make sure that you actually do intern() all of the strings tha you're going to compare. It's easy to forget to intern() all strings and then you can get confusingly incorrect results. Also, why unnecessarily create more objects?

Instead use string literals as shown below to intern automatically:

```
String s1 = "A";
String s2 = "A";

```

s1 and s2 point to the same String object in the pool. Hence s1 == s2 returns true.

Since interning is automatic for String literals String s1 = "A", the *intern( )* method is to be used on Strings constructed with new String("A").

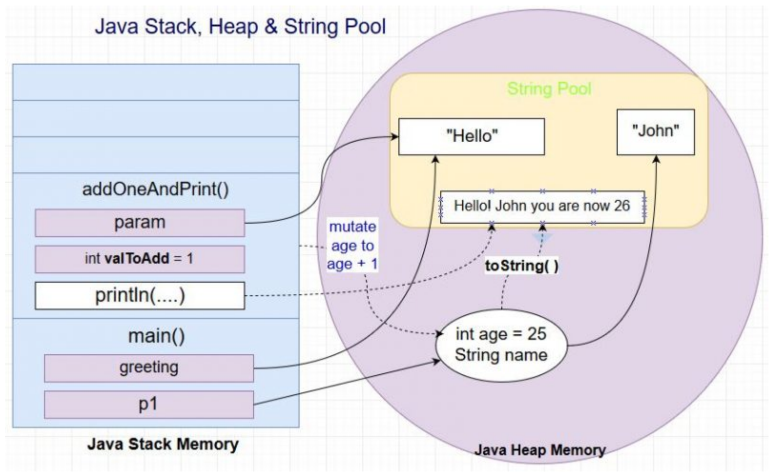
Q3. Can you describe what the following code does and what parts of memory the local variables, objects, and references to th objects occupy in Java?

```

public class Person {

    //instance variables
    private String name;
    private int age;

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

    public static void main(String[] args) {
        Person p1 = new Person("John", 25); // p1 is local object reference
        String greeting = "Hello";          //greeting is local object reference
        p1.addOneAndPrint(greeting);
    }

    public void addOneAndPrint(String param) {
        int valToAdd = 1;                      // local variable
        this.age  = this.age + valToAdd;       // instance variable is mutated to 26
        System.out.println(param + "! " + this); // calls toString() method
    }

    @Override
    public String toString() {
        return name + " you are now " + age;
    }
}


```

A3. The above code outputs "Hello! John you are now 26". The following diagram depicts how the different variables & objects stored.

Stack: local variables (both primitive & reference) and method parameters.

Heap: objects (Strings will be in **String Pool** within the heap)



Java Stack, Heap, and String Pool

Q4. Why String class has been made immutable in Java?
A4. For performance, thread-safety & error proneness/security.

1. Performance: Immutable objects are ideal for representing values of abstract data (i.e. value objects) types like numbers, enumerated types, etc. If you need a different value, create a different object. In Java, *Integer*, *Long*, *Float*, *Character*, *BigIntege* and *BigDecimal* are all immutable objects. Optimization strategies like caching of hashcode, caching of objects, object pooling, can be easily applied to improve performance. If Strings were made mutable, string pooling would not be possible as changing string with one reference will lead to the wrong value for the other references.

[**Further reading:** Java immutable objects interview Q&As | Builder pattern and immutability in Java]

2. Thread safety as immutable objects are inherently thread safe as they cannot be modified once created. They can only be us as read only objects. They can easily be shared among multiple threads for better scalability.

3. Less error proneness : as immutable objects cannot be modified once created, and can be safely shared around.

Q. Why is a char array i.e char[] preferred over String to store a password?
A. String is immutable in Java and stored in the String pool. Once it is created it stays in the pool until garbage collected. This ha greater risk of **1)** someone producing a memory dump to find the password **2)** the application inadvertently logging password readable string.

If you use a char[] instead, you can override it with some dummy values once done with it, and also logging the char[] like "[C@5829428e" is not as bad as logging it as String "password123".

Shouldn't there be more FAQs on Java strings? : 10 FAQ Java String class interview questions and answers as it is one of the v frequently used data types.

Let's move on to the next set of FAQ Core Java interview questions and answers focusing on the java modifiers final, finally and finalize.

Q5. In Java, what purpose does the key words **final**, **finally**, and **finalize** fulfill?
A5. '**final**' makes a variable reference not changeable, makes a method not overridable, and makes a class not inheritable. Lear more about the drill down question on 6 Java modifiers that interviewers like to quiz you on…

'**finally**' is used in a try/catch statement to almost always execute the code. Even when an exception is thrown, the finally block executed. This is used to close non-memory resources like file handles, sockets, database connections, etc till Java 7. This is is r longer true in Java 7.

Java 7 has introduced the *AutoCloseable* interface to avoid the unsightly try/catch/finally(within finally try/catch) blocks to close resource. It also prevents potential resource leaks due to not properly closing a resource.

//Pre Java 7

```
BufferedReader br = null;
try {
  File f = new File("c://temp/simple.txt");
  InputStream is = new FileInputStream(f);
  InputStreamReader isr = new InputStreamReader(is);
  br = new BufferedReader(isr);

  String read;

  while ((read = br.readLine()) != null) {
     System.out.println(read);
  }
} catch (IOException ioe) {
  ioe.printStackTrace();
} finally {
  //Hmmm another try catch. unsightly
  try {
    if (br != null) {
       br.close();
    }
  } catch (IOException ex) {
    ex.printStackTrace();
 }
}

```

Java 7 – try can have AutoCloseble types. *InputStream* and *OutputStream* classes now implements the Autocloseable interface

```
try (InputStream is = new FileInputStream(new File("c://temp/simple.txt"));
 InputStreamReader isr = new InputStreamReader(is);
 BufferedReader br2 = new BufferedReader(isr);) {
   String read;

   while ((read = br2.readLine()) != null) {
      System.out.println(read);
   }
 }

 catch (IOException ioe) {
   ioe.printStackTrace();
 }

```

try can now have multiple statements in the parenthesis and each statement should create an object which implements the ne *java.lang.AutoClosable* interface. The ***AutoClosable*** interface consists of just one method. void close() throws *Exception* {}. Each *AutoClosable* resource created in the try statement will be automatically closed without requiring a finally block. If an exceptior thrown in the try block and another Exception is thrown while closing the resource, the first Exception is the one eventually thrown to the caller. Think of the *close( )* method as implicitly being called as the last line in the try block. If using Java 7 or later editions, use *AutoCloseable* statements within the try block for more concise & readable code.

'finalize' is called when an object is garbage collected. You rarely need to override it. It should not be used to release non-mem resources like file handles, sockets, database connections, etc because Java has only a finite number of these resources and yo do not know when the Garbage Collection (i.e. GC) is going to kick in to release these non-memory resources through the *fina)* method.

[Further reading: Java Garbage Collection interview Q&As to ascertain your depth of Java knowledge]

So, final and finally are used very frequently in your Java code, but the key word finalize is hardly or never used.

Q6. What will be the output of the following code & why?

```
public static int getSomeNumber( ){
  try{
    return 2;
  } finally {
    return 1;
  }
}

```

A6. 1 is returned because 'finally' has the right to override any exception/returned value by the try..catch block. It is a bad pract to return from a finally block as it can suppress any exceptions thrown from a try..catch block. For example, the following code not throw an exception.

```
public static int getSomeNumber( ){
  try{
     throw new RuntimeException( );
  } finally {
     return 12;
  }
 }

```

Q7. What can prevent execution of a code in a finally block?
A7. **#1.** An end-less loop.

```
public static void main(String[ ] args) {
 try {
   System.out.println("This line is printed .....");
   //endless loop
   while(true){
     //...
   }
 }
 finally{
   System.out.println("Finally block is reached."); // won't reach
 }
}

```

#2. *System.exit(1)* statement.

```
public class Temp {

 public static void main(String[ ] args) {
   try {
     System.out.println("This line is printed .....");
     System.exit(1);
   }
   finally{
     System.out.println("Finally block is reached.");// won't reach
   }
 }
}

```

#3. Thread death or turning off the power to CPU.
#4. An exception arising in a finally block itself.
#5. Process p = Runtime.getRuntime( ).exec("kill -9 " +....);

Q8. Can you describe "method overloading" versus "method overriding"? Does it happen at **compile time** or **runtime**?
A8. Overloading deals with multiple methods in the same class with the **same name** but **different method signatures**. Both the below methods have the same method names but different method signatures, which mean the methods are overloaded.

```
public class {
  public static void evaluate(String param1);    // overloaded method #1
  public static void evaluate(int param1);       // overloaded method #2
}

```

This happens at compile-time. This is also called **compile-time polymorphism** because the compiler must decide which metho run based on the data types of the arguments. If the compiler were to compile the statement:

```
evaluate(“My Test Argument passed to param1”);
```

it could see that the argument was a “string” literal, and generates byte code that called method #1.

If the compiler were to compile the statement:

```
evaluate(5);
```

it could see that the argument was an “int”, and generates byte code that called method #2.

Overloading lets you define the **same operation in different ways for different data**.

Overriding deals with two methods, one in the **parent class** and the other one in the **child class** and has the **same name** and **sa** **signatures**. Both the below methods have the **same method names and the signatures** but the method in the subclass “B” overrides the method in the superclass (aka the parent class) “A”.

Parent class

```
public class A {
 public int compute(int input) { //method #3
   return 3 * input;
 }
}
```

Child class

```
public class B extends A {
 @Override
 public int compute(int input) { //method #4
   return 4 * input;
 }
}
```

This happens at runtime. This is also called **runtime polymorphism** because the compiler does not and cannot know which method to call. Instead, the JVM must make the determination while the code is running.

The method compute(..) in subclass “B” overrides the method compute(..) in super class “A”. If the compiler has to compile the following method,

```
public int evaluate(A reference, int arg2) {
  int result = reference.compute(arg2);
}
```

The compiler would not know whether the input argument ‘**reference**‘ is of type “A” or type “B”. This must be determined durin runtime whether to call method #3 or method #4 depending on what type of object (i.e. instance of Class A or instance of Clas is assigned to the input variable “reference”.

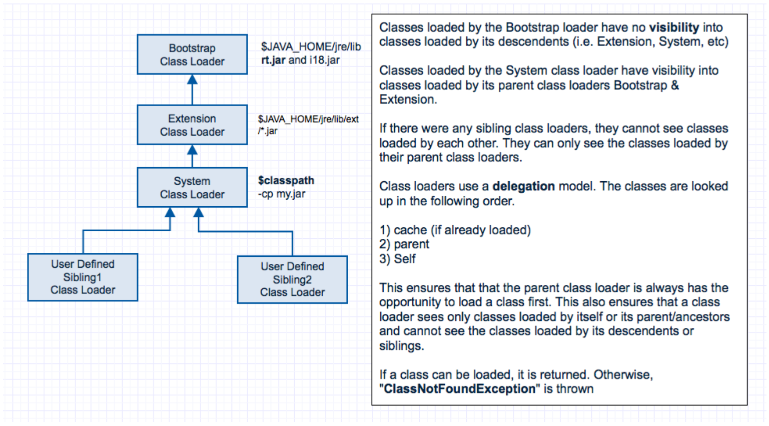
```
A obj1 = new B( );
A obj2 = new A( );
evaluate(obj1, 5); // 4 * 5 = 20. method #4 is invoked as stored object is of type B
evaluate(obj2, 5); // 3 * 5 = 15. method #3 is invoked as stored object is of type A
```

Overriding lets you define **the same operation in different ways for different object types**. It is determined by the “stored” obje type, and NOT by the “referenced” object type.

It is important to understand **compile-time** vs **runtime** from core and enterprise Java interview questions & answers perspectiv
More detailed discussion: Compile-time Vs Run-time Interview Q&As

Q. Are overriding & polymorphism applicable static methods as well?
A. No. If you try to override a static method, it is known as hiding or shadowing.

Q9. What do you know about class loading? Explain Java class loaders? If you have a class in a package, what do you need to d
run it? Explain dynamic class loading?
A9. Class loaders are hierarchical. Classes are introduced into the JVM as they are referenced by name in a class that is already running in the JVM. So, how is the very first class loaded? The very first class is specially loaded with the help of static *main( )* method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running. A class loader creates a namespace. All JVMs include at least one class loader that is embedded within the JVM called t
primordial (or bootstrap) class loader. The JVM has hooks in it to allow user defined class loaders to be used in place of primor
class loader. Let us look at the class loaders created by the JVM.


Bootstrap Class Loader
$JAVA_HOME/jre/lib rt.jar and i18.jar
Extension Class Loader
$JAVA_HOME/jre/lib/ext /*.jar
System Class Loader
$classpath -cp my.jar
User Defined Sibling1 Class Loader
User Defined Sibling2 Class Loader

Classes loaded by the Bootstrap loader have no **visibility** into classes loaded by its descendents (i.e. Extension, System, etc)

Classes loaded by the System class loader have visibility into classes loaded by its parent class loaders Bootstrap & Extension.

If there were any sibling class loaders, they cannot see classes loaded by each other. They can only see the classes loaded by their parent class loaders.

Class loaders use a **delegation** model. The classes are looked up in the following order.

1) cache (if already loaded)
2) parent
3) Self

This ensures that that the parent class loader is always has the opportunity to load a class first. This also ensures that a class loader sees only classes loaded by itself or its parent/ancestors and cannot see the classes loaded by its descendents or siblings.

If a class can be loaded, it is returned. Otherwise, "**ClassNotFoundException**" is thrown

Java Class Loader Basics

Class loaders are hierarchical and use a delegation model when loading a class. Class loaders request their parent to load the class first before attempting to load it themselves. When a class loader loads a class, the child class loaders in the hierarchy wil never reload the class again. Hence uniqueness is maintained. Classes loaded by a child class loader have visibility into classes loaded by its parents up the hierarchy but the reverse is not true as explained in the above diagram.

Q10. Explain static vs. dynamic class loading?
A10. Classes are **statically loaded** with Java's "new" operator.

```
class MyClass {
 public static void main(String args[]) {
   Car c = new Car( );
 }
}

```

Dynamic loading is a technique for programmatically invoking the functions of a **classloader** at **runtime**. Let us look at how to l
classes dynamically.

```
//static method which returns a Class
Class clazz = Class.forName ("com.Car");  // The value "com.Car" can be evaluated at runtime & passed in via a varia

```

The above static method "forName" & the below instance (i.e. non-static method) "loadClass"

```
ClassLoader classLoader = MyClass.class.getClassLoader();
Class clazz = classLoader.loadClass(("com.Car"); //Non-static method that returns a Class

```

return the class object associated with the class name.Once the class is dynamically loaded the following method returns an instance of the loaded class. It's just like creating a class object with no arguments.

```
// A non-static method, which creates an instance of a
// class (i.e. creates an object).
Car myCar = (Car) clazz.newInstance ( );

```

The string class name like "com.Car" can be supplied dynamically at runtime. Unlike the static loading, the dynamic loading will decide whether to load the class "com.Car" or the class "com.Jeep" at runtime based on a runtime condition.

```

public void process(String classNameSupplied) {
   Object vehicle = Class.forName (classNameSupplied).newInstance();
   //......
}

```

Static class loading throws NoClassDefFoundError if the class is NOT FOUND whereas the dynamic class loading throws ClassNotFoundException if the class is NOT FOUND.

Q. What is the difference between the following approaches?

```
Class.forName("com.SomeClass");
```

and

```
classLoader.loadClass("com.SomeClass");
```

A.

Class.forName("com.SomeClass")

— Uses the caller's classloader and initializes the class (runs static intitializers, etc.)

classLoader.loadClass("com.SomeClass")

— Uses the "supplied class loader", and initializes the class **lazily** (i.e. on first use). So, if you use this way to load a JDBC driver, i won't get registered, and you won't be able to use JDBC.

The "java.lang.API" has a method signature that takes a boolean flag indicating whether to initialize the class on loading or not, and a class loader reference.

```

forName(String name, boolean initialize, ClassLoader loader)

```

So, invoking

```
Class.forName("com.SomeClass")
```

is same as invoking

```
forName("com.SomeClass", true, currentClassLoader)
```

Q. What are the different ways to create a “ClassLoader” object?
A.

```

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
ClassLoader classLoader = MyClass.class.getClassLoader();      // Assuming in class "MyClass"
ClassLoader classLoader = getClass().getClassLoader();         // works in any class

```

Q. How to load property file from classpath?
A. **getResourceAsStream()** is the method of java.lang.Class. This method finds the resource by implicitly delegating to this objec
class loader.

```

final Properties properties = new Properties();
try (final InputStream stream = this.getClass().getResourceAsStream("myapp.properties")) {
    properties.load(stream);
    /* or properties.loadFromXML(...) */
}

```

Note: “Try with AutoCloseable resources” syntax introduced with Java 7 is used above.

Q. What is the benefit of loading a property file from classpath?
A. It is portable as your file is relative to the classpath. You can deploy the “jar” file containing your “property” file to **any locatic**
where the JVM is.

Loading it from outside the classpath is NOT portable

```

final Properties properties = new Properties();
final String dir = System.getProperty("user.dir");

try (final InputStream stream =  new FileInputStream(dir + "/myapp/myapp.properties")) {
    properties.load(stream);
}

```

As the above code is NOT portable, you must document very clearly in the installation or deployment document as to where th
property file is loaded from because if you deploy your “jar” file to another location, it might not already have the path “dir” anc
“myapp” configured.

So, loading it via the classpath is recommended as it is a portable solution.

Question to ponder

Q. What tips would you give to someone who is experiencing a class loading or “Class Not Found” exception?
A. “***ClassNotFoundException***” could be quite tricky to troubleshoot. When you get a *ClassNotFoundException*, it means the JVM
traversed the entire classpath and not found the class you’ve attempted to reference.

About Arulkumaran Kumaraswamipillai
Mechanical Engineering to Java freelancer since 2003. Published Java/JEE books via Amazon.com in 2005, and
sold 35K+ copies. Books are outdated and replaced with this online Java training. join my LinkedIn group.