Is Data Passed by Reference or by Value in Java?

- Passing by value – it means that we pass a copy of an object as a parameter into a method.
- Passing by reference – it means that we pass a reference to an object as a parameter into a method.
- When we pass primitives to a method, its value is copied into a new variable.
- When it comes to objects, the value of the reference is copied into a new variable.
- So we can say that Java is a strictly pass-by-value language.

Other modifiers:

- **Static** fields or methods are class members, whereas non-static ones are object members.
- When **final** is used on a field, it means that the field reference cannot be changed.
- When **final** is applied to a class or a method, it assures us that that class or method cannot be extended or overridden.
- When classes are **abstract**, they can't be instantiated.
- When we use **synchronized** keyword, we make Java use a monitor lock to provide synchronization on a given code fragment.
- **Volatile** declares that the field value must be read from and written to main memory – bypassing the CPU cache.

Difference between Heap and Stack?

- There are two parts of memory where all variables and objects are stored by the JVM. The first is the stack and the second is the heap.

Stack:

- A stack is a place where the JVM reserves blocks for local variables and additional data.
- The stack is a LIFO (last in first out) structure. It means that whenever a method is called, a new block is reserved for local variables and object references.
- Each new method invocation reserves the next block.
- When methods finish their execution, blocks are released in the reversed manner they were started.
- Every new thread has its own stack.

HEAP:

- We should be aware that the stack has much less memory space than the heap.
- And when a stack is full, the JVM will throw a StackOverflowError.
- It's likely to occur when there is a bad recursive call and the recursion goes too deep.
- Every new object is created on the Java heap which is used for a dynamic allocation.
- There is a garbage collector which is responsible for

Which Access Modifiers Are Available in Java and What Is Their Purpose?

There are four access modifiers in Java:

- Private
  - The private modifier assures that class members won't be accessible outside the class.
- default (package)
  - If we use default visibility our class or its members will be accessible only inside the package of our class.
- Protected
  - The protected modifier allows subclasses to access the protected members of a superclass, even if they are not within the same package.
- Public
  - Public modifier can be used together with class keyword and all class members. It makes classes and class members accessible in all packages and by all classes.

Difference Between JDK, JRE, and JVM?

JDK:

- JDK stands for Java Development Kit, which is a set of tools necessary for developers to write applications in Java. There are three types of JDK environments:
  - Standard Edition – development kit for creating portable desktop or server application.
  - Enterprise Edition – an extension to the Standard Edition with support for distributed computing or web services
  - Micro Edition – development platform for embedded and mobile applications
- There are plenty of tools included in the JDK which help programmers with writing, debugging or maintaining applications.
- The most popular ones are a compiler (javac), an interpreter (java), an archiver (jar) and a documentation generator (javadoc).

JRE:

- JRE is a Java Runtime Environment. It's a part of the JDK, but it contains the minimum functionality to run Java applications.
- It consists of a Java Virtual Machine, core classes, and supporting files. For example, it doesn't have any compiler.

JVM:

- JVM is the acronym for Java Virtual Machine, which is a virtual machine able to run programs compiled to bytecode.
- It's described by the JVM specification, as it's

erasing unused objects which are divided into young (nursery) and old spaces.
- Memory access to the heap is slower than access to the stack.
- The JVM throws an OutOfMemoryError when the heap is full.

important to ensure interoperability between different implementations.
- The most important function of a JVM is to enable users to deploy the same Java application into different operating systems and environments without worrying about what lies underneath.

Difference Between the Comparable and Comparator Interfaces?

- Sometimes when we write a new class, we would like to be able to compare objects of that class. It's especially helpful when we want to use sorted collections.
- There are two ways we can do this:
  - with the Comparable interface or
  - with the Comparator interface.

Comparable:

- First, let's look at the **Comparable** interface:

```
public interface Comparable<T> {
   int compareTo(T var1);
}
```

- We should implement that interface by the class whose objects we want to sort.
- It has the compareTo() method and returns an integer. It can return three values: -1, 0, and 1 which means that this object is less than, equal to or greater than the compared object.
- It's worth mentioning that the overridden compareT0() method should be consistent with the equals() method.

Comparator:

- On the other hand, we can use the Comparator interface. It can be passed to the sort() methods of the Collection interface or when instantiating sorted collections.
- That's why it's mostly used to create a one-time sorting strategy.
- What's more, it's also useful when we use a third-party class which doesn't implement the Comparable interface.
- Like the compareTo() method, the overridden compare() methods should be consistent with the equals() method, but they may optionally allow comparison with nulls.

What Is the void Type and When Do We Use It?

- Every time we write a method in Java, it must have a return type. If we want the method to return no value, we can use the void keyword.
- We should also know that there is a Void class. It's a placeholder class that may be used, for example, when working with generics. The Void class can neither be instantiated nor extended.

What Are the Methods of the Object Class and What Do They Do?

- It's important to know what methods the Object class contains and how they work.
- It's also very helpful when we want to override those methods:
- clone() – returns a copy of this object
- equals() – returns true when this object is equal to the object passed as a parameter
- finalize() – the garbage collector calls this method while it's cleaning the memory
- getClass() – returns the runtime class of this object
- hashCode() – returns a hash code of this object. We should be aware that it should be consistent with the equals() method
- notify() – sends a notification to a single thread waiting for the object's monitor
- notifyAll() – sends a notification to all threads waiting for the object's monitor
- toString() – returns a string representation of this object
- wait() – there are three overloaded versions of this method. It forces the current thread to wait the specified amount of time until another thread calls notify() or notifyAll() on this object.

What Is an Enum and How We Can Use It?

- Enum is a type of class that allows developers to specify a set of predefined constant values.
- To create such a class we have to use the enum keyword.
- Let's imagine an enum of days of the week:

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
```

Another interesting advantage of Enums is that they are thread-safe and so they are popularly used as singletons.

Singleton Pattern:

- Normally, implementing a class using the Singleton pattern is quite non-trivial.
- Enums provide an easy and quick way of implementing singletons.

```
FRIDAY, SATURDAY
}
```

- To iterate over all constants we can use the static values() method.
- What's more, enums enable us to define members such as properties and methods like regular classes.
- Although it's a special type of class, we can't subclass it. An enum can, however, implement an interface.

Strategy Pattern:

- Conventionally the Strategy pattern is written by having an interface that is implemented by different classes.
- Adding a new strategy meant adding a new implementation class.
- With enums, this is achieved with less effort, adding a new implementation means defining just another instance with some implementation.
- The code snippet below shows how to implement the Strategy pattern:

```
public enum PizzaDeliveryStrategy {
   EXPRESS {
      @Override
      public void deliver(Pizza pz) {
         System.out.println("Pizza will be delivered in express
mode");
      }
   },
   NORMAL {
      @Override
      public void deliver(Pizza pz) {
         System.out.println("Pizza will be delivered in normal
mode");
      }
   };

   public abstract void deliver(Pizza pz);
}
```

What Is a NullPointerException?

- The NullPointerException is probably the most common exception in the Java world. It's an unchecked exception and thus extends RuntimeException. We shouldn't try to handle it.
- This exception is thrown when we try to access a variable or call a method of a null reference, like when:
  - invoking a method of a null reference
  - setting or getting a field of a null reference
  - checking the length of a null array reference
  - setting or getting an item of a null array reference
  - throwing null

- In addition to that, since the enum class implements the Serializable interface under the hood, the class is guaranteed to be a singleton by the JVM, which unlike the conventional implementation where we have to ensure that no new instances are created during deserialization.
- In the code snippet below, we see how we can implement singleton pattern:

```
public enum PizzaDeliverySystemConfiguration {
   INSTANCE;
   PizzaDeliverySystemConfiguration() {
      // Initialization configuration which involves
      // overriding defaults like delivery strategy
   }

   private PizzaDeliveryStrategy deliveryStrategy =
PizzaDeliveryStrategy.NORMAL;

   public static PizzaDeliverySystemConfiguration getInstance()
{
      return INSTANCE;
   }

   public PizzaDeliveryStrategy getDeliveryStrategy() {
      return deliveryStrategy;
   }
}
```

Comparing enum types using == operator:

- Since enum types ensure that only one instance of the constants exists in the JVM, we can safely use "==" operator to compare two variables; moreover the "==" operator provides compile-time and run-time safety.

What is a JAR?

- JAR is a shortcut for Java archive. It's an archive file packaged using the ZIP file format.
- We can use it to include the class files and auxiliary resources that are necessary for applications.
- It has many features:
  - Security – we can digitally sign JAR files
  - Compression – while using a JAR, we can compress files for efficient storage
  - Portability – we can use the same JAR file across multiple platforms
  - Versioning – JAR files can hold metadata about the files they contain
  - Sealing – we can seal a package within a JAR file. This means that all classes from one package must be included in the same JAR file
  - Extensions – we can use the JAR file format to package modules or extensions for existing software

What Are Two Types of Casting in Java? Which Exception May Be Thrown While Casting? How Can We Avoid It?

- We can distinguish two types of casting in Java.
- We can do upcasting which is casting an object to a supertype or downcasting which is casting an object to a subtype.

Upcasting:

- Upcasting is very simple, as we always can do that. For example, we can upcast a String instance to the Object type:
  Object str = "string";

Downcasting:

- Alternatively, we can downcast a variable. It's not as safe as upcasting as it involves a type check. If we incorrectly cast an object, the JVM will throw a ClassCastExcpetion at runtime.
- Fortunately, we can use the instanceof keyword to prevent invalid casting:

```
Object o = "string";
String str = (String) o; // it's ok
Object o2 = new Object();
String str2 = (String) o2; // ClassCastException thrown
if (o2 instanceof String) { // returns false
  String str3 = (String) o;
}
```

What Is JIT?

- JIT stands for "just in time". It's a component of the JRE that runs in the runtime and increases the performance of the application. Specifically, it's a compiler which runs just after the program's start.
- This is different from the regular Java compiler which compiles the code long before the application is started. JIT can speed up the application in different ways.
- For example, the JIT compiler is responsible for compiling bytecode into native instructions on the fly to improve performance. Also, it can optimize the code to the targeted CPU and operating system.
- Additionally, it has access to many runtime statistics which may be used for recompilation for optimal performance. With this, it can also do some global code optimizations or rearrange code for better cache utilization.

What Is Reflection in Java?

- Reflection is a very powerful mechanism in Java. Reflection is a mechanism of Java language which enables programmers to examine or modify the internal state of the program (properties, methods, classes etc.) at runtime. The java.lang.reflect package provides all required components for using reflection.
- When using this feature, we can access all possible

Why Is String an Immutable Class?

- We should know that String objects are treated differently than other objects by the JVM.
- One difference is that String objects are immutable.
- It means that we can't change them once we have created them.
- There are several reasons why they behave that way:
  - They are stored in the string pool which is a special part of the heap memory. It's responsible for saving a lot of space.
  - The immutability of the String class guarantees that it's hash code won't change. Due to that fact, Strings can be effectively used as keys in hashing collections. We can be sure that we won't overwrite any data because of a change in hash codes.
  - They can be used safely across several threads. No thread can change the value of a String object, so we get thread safety for free.
- Strings are immutable to avoid serious security issues. Sensitive data such as passwords could be changed by an unreliable source or another thread.

What Is the Difference Between Dynamic Binding and Static Binding?

- Binding in Java is a process of associating a method call with the proper method body.
- We can distinguish two types of binding in Java: static and dynamic.
- The main difference between static binding and dynamic binding is that static binding occurs at compile time and dynamic binding at runtime.

Static binding:

- Static binding uses class information for binding. It's responsible for resolving class members that are private or static and final methods and variables. Also, static binding binds overloaded methods.

Dynamic binding:

- Dynamic binding, on the other hand, uses object information to resolve bindings. That's why it's responsible for resolving virtual and overridden methods.

What Is a Classloader?

- The classloader is one of the most important components in Java. It's a part of the JRE.
- Simply put, the classloader is responsible for loading classes into the JVM. We can distinguish three types of classloaders:
  - Bootstrap classloader – it loads the core Java classes. They are located in the

fields, methods, constructors that are included within a class definition. We can access them irrespective of their access modifier. It means that for example, we are able to access private members. To do that, we don't have to know their names. All we have to do is to use some static methods of Class.

- It's worth to know that there is a possibility to restrict access via reflection. To do that we can use the Java security manager and the Java security policy file. They allow us to grant permissions to classes.
- When working with modules since Java 9, we should know that by default, we aren't able to use reflection on classes imported from another module. To allow other classes to use reflection to access the private members of a package we have to grant the "Reflection" Permission.

What Is the Purpose of the Serializable Interface?

- We can use the Serializable interface to enable serializability of a class, using Java's Serialization API.
- Serialization is a mechanism for saving the state of an object as a sequence of bytes while deserialization is a mechanism for restoring the state of an object from a sequence of bytes.
- The serialized output holds the object's state and some metadata about the object's type and types of its fields.
- We should know that subtypes of serializable classes are also serializable. However, if we want to make a class serializable, but its supertype is non-serializable we have to do two things:
    - implement the Serializable interface
    - assure that a no argument constructor is present in the superclass.

Describe the Collections type hierarchy. What are the main interfaces, and what are the differences between them?

- The Iterable interface represents any collection that can be iterated using the for-each loop.
- The Collection interface inherits from Iterable and adds generic methods for checking if an element is in a collection, adding and removing elements from the collection, determining its size etc.
- The List, Set, and Queue interfaces inherit from the Collection interface.

List:

- List is an ordered collection, and its elements can be accessed by their index in the list.

Set:

- Set is an unordered collection with distinct elements, similar to the mathematical notion of a set.

Queue:

<JAVA_HOME>/jre/lib directory
    - Extension classloader – it loads classes located in <JAVA_HOME>/jre/lib/ext or in the path defined by the java.ext.dirs property
    - System classloader – it loads classes on the classpath of our application
- A classloader loads classes "on demand".
- It means that classes are loaded after they are called by the program.
- What's more, a classloader can load a class with a given name only once.
- However, if the same class is loaded by two different class loaders, then those classes fail in an equality check.

What Is the Difference Between Static and Dynamic Class Loading?

- Static class loading takes place when we have source classes available at compile time. We can make use of it by creating object instances with the new keyword.
- Dynamic class loading refers to a situation when we can't provide a class definition at the compile time. Yet, we can do that at runtime. To create an instance of a class, we have to use the Class.forName() method: Class.forName("oracle.jdbc.driver.OracleDriver")

Is there a destructor in Java?

- In Java, the garbage collector automatically deletes the unused objects to free up the memory.
- Developers have no need to mark the objects for deletion, which is error-prone.
- So it's sensible Java has no destructors available.

- In case the objects hold open sockets, open files, or database connections, the garbage collector is not able to reclaim those resources.
- We can release the resources in close method and use try-finally syntax to call the method afterward before Java 7, such as the I/O classes FileInputStreamand FileOutputStream.

- **As of Java 7**, we can implement interface AutoCloseable and use try-with-resources statement to write shorter and cleaner code.
- But it's possible the API users forget to call the close method, so the finalize method and Cleaner class comes into existence to act as the safety net. But please be cautioned they are not equivalent to the destructor.
- It's not assured both the finalize method and Cleaner class will run promptly.
- They even get no chance to run before the JVM exits. Although we could call System.runFinalization to suggest that JVM run the finalize methods of any objects pending for finalization, it's still

- Queue is a collection with additional methods for adding, removing and examining elements, useful for holding elements prior to processing.

Map:

- Map interface is also a part of the collection framework, yet it does not extend Collection. This is by design, to stress the difference between collections and mappings which are hard to gather under a common abstraction. The Map interface represents a key-value data structure with unique keys and no more than one value for each key.

Describe various implementations of the Map interface and their use case differences?

HashMap:

- One of the most often used implementations of the Map interface is the HashMap. It is a typical hash map data structure that allows accessing elements in constant time, or O(1), but does not preserve order and is not thread-safe.

LinkedHashMap:

- To preserve insertion order of elements, you can use the LinkedHashMap class which extends the HashMap and additionally ties the elements into a linked list, with foreseeable overhead.

TreeMap:

- The TreeMap class stores its elements in a red-black tree structure, which allows accessing elements in logarithmic time, or O(log(n)). It is slower than the HashMap for most cases, but it allows keeping the elements in order according to some Comparator.

ConcurrentHashMap:

- The ConcurrentHashMap is a thread-safe implementation of a hash map. It provides full concurrency of retrievals (as the get operation does not entail locking) and high expected concurrency of updates.

Hashtable:

- The Hashtable class has been in Java since version 1.0. It is not deprecated but is mostly considered obsolete. It is a thread-safe hash map, but unlike ConcurrentHashMap, all its methods are simply synchronized, which means that all operations on this map block, even retrieval of independent values.

How is HashMap implemented in Java? How does its implementation use hashCode and equals methods of

non-deterministic.

- Moreover, the finalize method can cause performance issues, deadlocks, etc. We can find more information by looking at one of our articles: A Guide to the finalize Method in Java.

- **As of Java 9**, Cleaner class is added to replace the finalize method because of the downsides it has. As a result, we have better control over the thread which does the cleaning actions.
- But the java spec points out the behavior of cleaners during System.exit is implementation specific and Java provides no guarantees whether cleaning actions will be invoked or not.

Explain the difference between LinkedList and ArrayList.

ArrayList:

- ArrayList is an implementation of the List interface that is based on an array. ArrayList internally handles resizing of this array when the elements are added or removed. You can access its elements in constant time by their index in the array. However, inserting or removing an element infers shifting all consequent elements which may be slow if the array is huge and the inserted or removed element is close to the beginning of the list.

LinkedList:

- LinkedList is a doubly-linked list: single elements are put into Node objects that have references to previous and next Node. This implementation may appear more efficient than ArrayList if you have lots of insertions or deletions in different parts of the list, especially if the list is large.

- In most cases, however, ArrayList outperforms LinkedList.
- Even elements shifting in ArrayList, while being an O(n) operation, is implemented as a very fast System.arraycopy() call.
- It can even appear faster than the LinkedList's O(1) insertion which requires instantiating a Node object and updating multiple references under the hood.
- LinkedList also can have a large memory overhead due to the creation of multiple small Node objects.

What is the difference between HashSet and TreeSet?

- Both HashSet and TreeSet classes implement the Set interface and represent sets of distinct elements. Additionally, TreeSet implements the NavigableSet interface. This interface defines methods that take advantage of the ordering of elements.
- HashSet is internally based on a HashMap, and TreeSet is backed by a TreeMap instance, which defines their

objects? What is the time complexity of putting and getting an element from such structure?

- The HashMap class represents a typical hash map data structure with certain design choices.
- The HashMap is backed by a resizable array that has a size of power-of-two.
- When the element is added to a HashMap, first its hashCode is calculated (an int value). Then a certain number of lower bits of this value are used as an array index. This index directly points to the cell of the array (called a bucket) where this key-value pair should be placed. Accessing an element by its index in an array is a very fast O(1) operation, which is the main feature of a hash map structure.
- A hashCode is not unique, however, and even for different hashCodes, we may receive the same array position. This is called a collision. There is more than one way of resolving collisions in the hash map data structures. In Java's HashMap, each bucket actually refers not to a single object, but to a red-black tree of all objects that landed in this bucket (prior to Java 8, this was a linked list).
- So when the HashMap has determined the bucket for a key, it has to traverse this tree to put the key-value pair in its place. If a pair with such key already exists in the bucket, it is replaced with a new one.
- To retrieve the object by its key, the HashMap again has to calculate the hashCode for the key, find the corresponding bucket, traverse the tree, call equals on keys in the tree and find the matching one.
- HashMap has O(1) complexity, or constant-time complexity, of putting and getting the elements. Of course, lots of collisions could degrade the performance to O(log(n)) time complexity in the worst case, when all elements land in a single bucket. This is usually solved by providing a good hash function with a uniform distribution.
- When the HashMap internal array is filled (more on that in the next question), it is automatically resized to be twice as large. This operation infers rehashing (rebuilding of internal data structures), which is costly, so you should plan the size of your HashMap beforehand.

properties:

  - HashSet does not keep elements in any particular order.
  - Iteration over the elements in a HashSet produces them in a shuffled order.
  - TreeSet, on the other hand, produces elements in order according to some predefined Comparator.

Describe special collections for enums. What are the benefits of their implementation compared to regular collections?

- EnumSet and EnumMap are special implementations of Set and Map interfaces correspondingly. You should always use these implementations when you're dealing with enums because they are very efficient.
- An EnumSet is just a bit vector with "ones" in the positions corresponding to ordinal values of enums present in the set. To check if an enum value is in the set, the implementation simply has to check if the corresponding bit in the vector is a "one", which is a very easy operation.
- Similarly, an EnumMap is an array accessed with enum's ordinal value as an index. In the case of EnumMap, there is no need to calculate hash codes or resolve collisions.

What is the difference between fail-fast and fail-safe iterators?

- Iterators for different collections are either fail-fast or fail-safe, depending on how they react to concurrent modifications. The concurrent modification is not only a modification of collection from another thread but also modification from the same thread but using another iterator or modifying the collection directly.
- Fail-fast iterators (those returned by HashMap, ArrayList, and other non-thread-safe collections) iterate over the collection's internal data structure, and they throw ConcurrentModificationException as soon as they detect a concurrent modification.
- Fail-safe iterators (returned by thread-safe collections such as ConcurrentHashMap, CopyOnWriteArrayList) create a copy of the structure they iterate upon. They guarantee safety from concurrent modifications. Their drawbacks include excessive memory consumption and iteration over possibly out-of-date data in case the collection was modified.

What is the purpose of the initial capacity and load factor parameters of a HashMap? What are their default values?

- The initialCapacity argument of the HashMap constructor affects the size of the internal data structure of the HashMap, but reasoning about the actual size of a map is a bit tricky. The HashMap's internal data structure is an array with the

How can you use Comparable and Comparator interfaces to sort collections?

- The Comparable interface is an interface for objects that can be compared according to some order. Its single method is compareTo, which operates on two values: the object itself and the argument object of the same type. For instance, Integer, Long, and other

power-of-two size. So the initialCapacity argument value is increased to the next power-of-two (for instance, if you set it to 10, the actual size of the internal array will be 16).

- The load factor of a HashMap is the ratio of the element count divided by the bucket count (i.e. internal array size). For instance, if a 16-bucket HashMap contains 12 elements, its load factor is 12/16 = 0.75. A high load factor means a lot of collisions, which in turn means that the map should be resized to the next power of two. So the loadFactor argument is a maximum value of the load factor of a map. When the map achieves this load factor, it resizes its internal array to the next power-of-two value.
- The initialCapacity is 16 by default, and the loadFactor is 0.75 by default, so you could put 12 elements in a HashMap that was instantiated with the default constructor, and it would not resize. The same goes for the HashSet, which is backed by a HashMap instance internally.
- Consequently, it is not trivial to come up with initialCapacity that satisfies your needs. This is why the Guava library has Maps.newHashMapWithExpectedSize() and Sets.newHashSetWithExpectedSize() methods that allow you to build a HashMap or a HashSet that can hold the expected number of elements without resizing.

numeric types implement this interface. String also implements this interface, and its compareTo method compares strings in lexicographical order.

- The Comparable interface allows to sort lists of corresponding objects with the Collections.sort() method and uphold the iteration order in collections that implement SortedSet and SortedMap. If your objects can be sorted using some logic, they should implement the Comparable interface.
- The Comparable interface usually is implemented using natural ordering of the elements. For instance, all Integer numbers are ordered from lesser to greater values. But sometimes you may want to implement another kind of ordering, for instance, to sort the numbers in descending order. The Comparator interface can help here.
- The class of the objects you want to sort does not need to implement this interface. You simply create an implementing class and define the compare method which receives two objects and decides how to order them. You may then use the instance of this class to override the natural ordering of the Collections.sort() method or SortedSet and SortedMap instances.
- As the Comparator interface is a functional interface, you may replace it with a lambda expression, as in the following example. It shows ordering a list using a natural ordering (Integer‘s Comparable interface) and using a custom iterator (Comparator<Integer> interface).

Describe the place of the Object class in the type hierarchy. Which types inherit from Object, and which don’t? Do arrays inherit from Object? Can a lambda expression be assigned to an Object variable?

- The java.lang.Object is at the top of the class hierarchy in Java. All classes inherit from it, either explicitly, implicitly (when the extends keyword is omitted from the class definition) or transitively via the chain of inheritance.
- However, there are eight primitive types that do not inherit from Object, namely boolean, byte, short, char, int, float, long and double.
- According to the Java Language Specification, arrays are objects too. They can be assigned to an Object reference, and all Object methods may be called on them.
- Lambda expressions can’t be assigned directly to an Object variable because Object is not a functional interface. But you can assign a lambda to a functional interface variable and then assign it to an Object variable(or simply assign it to an Object variable by casting it to a functional interface at the same time).

Explain the difference between primitive and reference types.

- Reference types inherit from the top java.lang.Object class and are themselves inheritable (except final classes). Primitive types do not inherit and cannot be subclassed.
- Primitively typed argument values are always passed via the stack, which means they are passed by value, not by reference. This has the following implication: changes made to a primitive argument value inside the method do not propagate to the actual argument value.
- Primitive types are usually stored using the underlying hardware value types.
- For instance, to store an int value, a 32-bit memory cell can be used. Reference types introduce the overhead of object header which is present in every instance of a reference type.
- The size of an object header can be quite significant in relation to a simple numeric value size. This is why the primitive types were introduced in the first place — to save space on object overhead. The downside is that not everything in Java technically is an object — primitive values do not inherit from Object class.

Describe the different primitive types and the amount of memory they occupy.

Java has 8 primitive types:

- boolean — logical true/false value. The size of boolean is not defined by the JVM specification and can vary in different implementations.
- byte — signed 8-bit value,
- short — signed 16-bit value,
- char — unsigned 16-bit value,
- int — signed 32-bit value,
- long — signed 64-bit value,
- float — 32-bit single precision floating point value corresponding to the IEEE 754 standard,
- double — 64-bit double precision floating point value corresponding to the IEEE 754 standard.

What are the restrictions on the members (fields and methods) of an interface type?

- An interface can declare fields, but they are implicitly declared as public, static and final, even if you don't specify those modifiers.
- Consequently, you can't explicitly define an interface field as private. In essence, an interface may only have constant fields, not instance fields.
- All methods of an interface are also implicitly public. They also can be either (implicitly) abstract, or default.

Does Java have multiple inheritance?

- Java does not support the multiple inheritance for classes, which means that a class can only inherit from a single superclass.
- But you can implement multiple interfaces with a single class, and some of the methods of those interfaces may be defined as default and have an implementation. This allows you to have a safer way of mixing different functionality in a single class.

What are the wrapper classes? What is autoboxing?

- For each of the eight primitive types in Java, there is a wrapper class that can be used to wrap a primitive value and use it like an object. Those classes are, correspondingly, Boolean, Byte, Short, Character, Integer, Float, Long, and Double. These wrappers can be useful, for instance, when you need to put a primitive value into a generic collection, which only accepts reference objects.

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(5));
```

- To save the trouble of manually converting primitives back and forth, an automatic conversion known as autoboxing/auto-unboxing is provided by the Java compiler.

What is the difference between an abstract class and an interface? What are the use cases of one and the other?

- An abstract class is a class with the abstract modifier in its definition. It can't be instantiated, but it can be subclassed. The interface is a type described with interface keyword. It also cannot be instantiated, but it can be implemented.
- The main difference between an abstract class and an interface is that a class can implement multiple interfaces, but extend only one abstract class.
- An abstract class is usually used as a base type in some class hierarchy, and it signifies the main intention of all classes that inherit from it.
- An abstract class could also implement some basic methods needed in all subclasses. For instance, most map collections in JDK inherit from the AbstractMap class which implements many methods used by subclasses (such as the equals method).
- An interface specifies some contract that the class agrees to. An implemented interface may signify not only the main intention of the class but also some additional contracts.
- For instance, if a class implements the Comparable interface, this means that instances of this class may be compared, whatever the main purpose of this class is.

What is the difference between an inner class and a static nested class?

- Simply put – a nested class is basically a class defined inside another class.

Nested classes:

- Nested classes fall into two categories with very different properties. An inner class is a class that can't be instantiated without instantiating the enclosing class first, i.e. any instance of an inner class is implicitly bound to some instance of the enclosing class.
- Here's an example of an inner class – you can see that it can access the reference to the outer class instance in the form of OuterClass1.this construct:

```
public class OuterClass1 {
   public class InnerClass {
      public OuterClass1 getOuterInstance() {
         return OuterClass1.this;
      }
   }
}
```

- To instantiate such inner class, you need to have an instance of an outer class:

```
OuterClass1 outerClass1 = new OuterClass1();
OuterClass1.InnerClass innerClass = outerClass1.new
InnerClass();
```

```
List<Integer> list = new ArrayList<>();
list.add(5);
int value = list.get(0);
```

Describe the difference between equals() and ==

- The == operator allows you to compare two objects for “sameness” (i.e. that both variables refer to the same object in memory). It is important to remember that the new keyword always creates a new object which will not pass the == equality with any other object, even if they seem to have the same value:
- The equals() method is defined in the java.lang.Object class and is, therefore, available for any reference type. By default, it simply checks that the object is the same via the == operator. But it is usually overridden in subclasses to provide the specific semantics of comparison for a class.

Static nested class:

- Static nested class is quite different. Syntactically it is just a nested class with the static modifier in its definition.
- In practice, it means that this class may be instantiated as any other class, without binding it to any instance of the enclosing class:

```
public class OuterClass2 {
   public static class StaticNestedClass {
   }
}
```

- To instantiate such class, you don’t need an instance of outer class:

```
OuterClass2.StaticNestedClass staticNestedClass = new
OuterClass2.StaticNestedClass();
```

Suppose you have a variable that references an instance of a Class type. How do you check that an object is an instance of this class?

- You cannot use instanceof keyword in this case because it only works if you provide the actual class name as a literal.
- Thankfully, the Class class has a method isInstance that allows checking if an object is an instance of this class:

What is an anonymous class? Describe its use case.

- Anonymous class is a one-shot class that is defined in the same place where its instance is needed. This class is defined and instantiated in the same place, thus it does not need a name.
- Before Java 8, you would often use an anonymous class to define the implementation of a single method interface, like Runnable. In Java 8, lambdas are used instead of single abstract method interfaces. But anonymous classes still have use cases, for example, when you need an instance of an interface with multiple methods or an instance of a class with some added features.

What is a Generic Type Parameter?

- Type is the name of a class or interface. As implied by the name, a generic type parameter is when a type can be used as a parameter in a class, method or interface declaration.
- Let’s start with a simple example, one without generics, to demonstrate this:

```
public interface Consumer {
   public void consume(String parameter)
}
```

- In this case, the method parameter type of the consume() method is String. It is not parameterized and not configurable.
- Now let’s replace our String type with a generic type that we will call T. It is named like this by convention:

```
public interface Consumer<T> {
   public void consume(T parameter)
}
```

- When we implement our consumer, we can provide the type that we want it to consume as an argument. This is a generic type parameter:

```
public class IntegerConsumer implements
Consumer<Integer> {
   public void consume(Integer parameter)
```

What Are Some Advantages of Using Generic Types?

- One advantage of using generics is avoiding costs and provide type safety. This is particularly useful when working with collections. Let’s demonstrate this:
- Let’s demonstrate this:

```
List list = new ArrayList();
list.add("foo");
Object o = list.get(1);
String foo = (String) foo;
```

- In our example, the element type in our list is unknown to the compiler. This means that the only thing that can be guaranteed is that it is an object. So when we retrieve our element, an object is what we get back. As the authors of the code, we know it’s a String, but we have to cast our object to one to fix the problem explicitly. This produces a lot of noise and boilerplate.
- Next, if we start to think about the room for manual error, the casting problem gets worse. What if we accidentally had an integer in our list?

```
}
```

- In this case, now we can consume integers. We can swap out this type for whatever we require.

What is Type Erasure?

- It's important to realize that generic type information is only available to the compiler, not the JVM. In other words, type erasure means that generic type information is not available to the JVM at runtime, only compile time.
- The reasoning behind major implementation choice is simple – preserving backward compatibility with older versions of Java. When a generic code is compiled into bytecode, it will be as if the generic type never existed. This means that the compilation will:
  - Replace generic types with objects.
  - Replace bounded types (More on these in a later question) with the first bound class
- Insert the equivalent of casts when retrieving generic objects.
- It's important to understand type erasure. Otherwise, a developer might get confused and think they'd be able to get the type at runtime:

```
public foo(Consumer<T> consumer) {
  Type type = consumer.getGenericTypeParameter()
}
```

- The above example is a pseudo code equivalent of what things might look like without type erasure, but unfortunately, it is impossible. Once again, the generic type information is not available at runtime.

How Does a Generic Method Differ From a Generic Type?

- A generic method is where a type parameter is introduced to a method, living within the scope of that method. Let's try this with an example:

```
public static <T> T returnType(T argument) {
  return argument;
}
```

- We've used a static method but could have also used a non-static one if we wished. By leveraging type inference (covered in the next question), we can invoke this like any ordinary method, without having to specify any type arguments when we do so.

What is a Bounded Type Parameter?

- So far all our questions have covered generic types arguments which are unbounded. This means that our generic type arguments could be any type that we want.
- When we use bounded parameters, we are restricting the types that can be used as generic type arguments.
- As an example, let's say we want to force our generic type always to be a subclass of animal:

```
list.add(1)
Object o = list.get(1);
String foo = (String) foo;
```

In this case, we would get a ClassCastException at runtime, as an Integer cannot be cast to String.

- Now, let's try repeating ourselves, this time using generics:

```
List<String> list = new ArrayList<>();
list.add("foo");
String o = list.get(1);    // No cast
Integer foo = list.get(1); // Compilation error
```

- As we can see, by using generics we have a compile type check which prevents ClassCastExceptions and removes the need for casting.
- The other advantage is to avoid code duplication. Without generics, we have to copy and paste the same code but for different types. With generics, we do not have to do this. We can even implement algorithms which apply to generic types.

If a Generic Type is Omitted When Instantiating an Object, will the Code Still Compile?

- As generics did not exist before Java 5, it is possible not to use them at all. For example, generics were retrofitted to most of the standard Java classes such as collections. If we look at our list from question one, then we will see that we already have an example of omitting the generic type:

```
List list = new ArrayList();
```

- Despite being able to compile, it's still likely that there will be a warning from the compiler. This is because we are losing the extra compile time check that we get from using generics.
- The point to remember is that while backward compatibility and type erasure make it possible to omit generic types, it is bad practice.

What is Type Inference?

- Type inference is when the compiler can look at the type of a method argument to infer a generic type. For example, if we passed in T to a method which returns T, then the compiler can figure out the return type. Let's try this out by invoking our generic method from the previous question:

```
Integer inferredInteger = returnType(1);
String inferredString = returnType("String");
```

- As we can see, there's no need for a cast, and no need to pass in any generic type argument. The argument type only infers the return type.

```
public abstract class Cage<T extends Animal> {
   abstract void addAnimal(T animal)
}
```

- By using extends, we are forcing T to be a subclass of animal. We could then have a cage of cats:
  Cage<Cat> catCage;

- But we could not have a cage of objects, as an object is not a subclass of an animal:
  Cage<Object> objectCage; // Compilation error

- One advantage of this is that all the methods of animal are available to the compiler. We know our type extends it, so we could write a generic algorithm which operates on any animal. This means we don't have to reproduce our method for different animal subclasses:

```
public void firstAnimalJump() {
   T animal = animals.get(0);
   animal.jump();
}
```

What is an Upper Bounded Wildcard?

- An upper bounded wildcard is when a wildcard type inherits from a concrete type. This is particularly useful when working with collections and inheritance.
- Let's try demonstrating this with a farm class which will store animals, first without the wildcard type:

```
public class Farm {
  private List<Animal> animals;
  public void addAnimals(Collection<Animal>
newAnimals) {
    animals.addAll(newAnimals);
  }
}
```

- If we had multiple subclasses of animal, such as cat and dog, we might make the incorrect assumption that we can add them all to our farm:
  farm.addAnimals(cats); // Compilation error
  farm.addAnimals(dogs); // Compilation error

- This is because the compiler expects a collection of the concrete type animal, not one it subclasses.
- Now, let's introduce an upper bounded wildcard to our add animals method:
  public void addAnimals(Collection<? extends Animal> newAnimals)

- Now if we try again, our code will compile. This is because we are now telling the compiler to accept a collection of any subtype of animal.

When Would You Choose to Use a Lower Bounded Type vs. an Upper Bounded Type?

Is it Possible to Declared a Multiple Bounded Type Parameter?

- Declaring multiple bounds for our generic types is possible. In our previous example, we specified a single bound, but we could also specify more if we wish:

  public abstract class Cage<T extends Animal & Comparable>

- In our example, the animal is a class and comparable is an interface. Now, our type must respect both of these upper bounds. If our type were a subclass of animal but did not implement comparable, then the code would not compile. It's also worth remembering that if one of the upper bounds is a class, it must be the first argument.

What is a Wildcard type?

- A wildcard type represents an unknown type. It's detonated with a question mark as follows:
  public static consumeListOfWildcardType(List<?> list)

- Here, we are specifying a list which could be of any type. We could pass a list of anything into this method.

What is an Unbounded Wildcard?

- An unbounded wildcard is a wildcard with no upper or lower bound, that can represent any type.
- It's also important to know that the wildcard type is not synonymous to object. This is because a wildcard can be any type whereas an object type is specifically an object (and cannot be a subclass of an object). Let's demonstrate this with an example:
  List<?> wildcardList = new ArrayList<String>();
  List<Object> objectList = new ArrayList<String>(); // Compilation error

- Again, the reason the second line does not compile is that a list of objects is required, not a list of strings. The first line compiles because a list of any unknown type is acceptable.

What is a Lower Bounded Wildcard?

- A lower bounded wildcard is when instead of providing an upper bound, we provide a lower bound by using the super keyword. In other words, a lower bounded wildcard means we are forcing the type to be a superclass of our bounded type. Let's try this with an example:

```
public static void addDogs(List<? super Animal> list) {
  list.add(new Dog("tom"))
}
```

- When dealing with collections, a common rule for selecting between upper or lower bounded wildcards is PECS. PECS stands for producer extends, consumer super.
- This can be easily demonstrated through the use of some standard Java interfaces and classes.
- Producer extends just means that if you are creating a producer of a generic type, then use the extends keyword. Let's try applying this principle to a collection, to see why it makes sense:

```
public static void makeLotsOfNoise(List<? extends
Animal> animals) {
   animals.forEach(Animal::makeNoise);
}
```

- Here, we want to call makeNoise() on each animal in our collection. This means our collection is a producer, as all we are doing with it is getting it to return animals for us to perform our operation on. If we got rid of extends, we wouldn't be able to pass in lists of cats, dogs or any other subclasses of animals. By applying the producer extends principle, we have the most flexibility possible.
- Consumer super means the opposite to producer extends. All it means is that if we are dealing with something which consumers elements, then we should use the super keyword. We can demonstrate this by repeating our previous example:

```
public static void addCats(List<? super Animal>
animals) {
   animals.add(new Cat());
}
```

- We are only adding to our list of animals, so our list of animals is a consumer. This is why we use the super keyword. It means that we could pass in a list of any superclass of animal, but not a subclass. For example, if we tried passing in a list of dogs or cats then the code would not compile.
- The final thing to consider is what to do if a collection is both a consumer and a producer. An example of this might be a collection where elements are both added and removed. In this case, an unbounded wildcard should be used.

- By using super, we could call addDogs on a list of objects:

```
ArrayList<Object> objects = new ArrayList<>();
addDogs(objects);
```

- This makes sense, as an object is a superclass of animal. If we did not use the lower bounded wildcard, the code would not compile, as a list of objects is not a list of animals.
- If we think about it, we wouldn't be able to add a dog to a list of any subclass of animal, such as cats, or even dogs. Only a superclass of animal. For example, this would not compile:

```
ArrayList<Cat> objects = new ArrayList<>();
addDogs(objects);
```

Are There Any Situations Where Generic Type Information is Available at Runtime?

- There is one situation where a generic type is available at runtime. This is when a generic type is part of the class signature like so:

```
public class CatCage implements Cage<Cat>
```

- By using reflection, we get this type parameter:

```
(Class<T>) ((ParameterizedType) getClass()

.getGenericSuperclass()).getActualTypeArguments()[0]
;
```

What does the statement "memory is managed in Java" mean?

- Memory is the key resource an application requires to run effectively and like any resource, it is scarce. As such, its allocation and deallocation to and from applications or different parts of an application require a lot of care and consideration.
- However, in Java, a developer does not need to explicitly allocate and deallocate memory – the JVM and more specifically the Garbage Collector – has the duty of handling memory allocation so that the developer doesn't have to.
- This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

What is Garbage Collection and what are its advantages?

- Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

Are there any disadvantages of Garbage Collection?

- Yes. Whenever the garbage collector runs, it has an effect on the application's performance. This is because all other threads in the application have to be

- An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.
- The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

What are stack and heap? What is stored in each of these memory structures, and how are they interrelated?

- The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap defined in currently executing methods.
- This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all local variables after exiting the method. Every thread has its own stack.
- The heap is a large bulk of memory intended for allocation of objects. When you create an object with the new keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

Describe in detail how generational garbage collection works

- To properly understand how generational garbage collection works, it is important to first remember how Java heap is structured to facilitate generational garbage collection.
- The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.
- The young generation hosts most of the newly created objects. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only “promoted” to the old generation space after they have attained a certain “age”.
- The term “age” in generational garbage collection refers to the number of collection cycles the object has survived.
- The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).
- The old generation hosts objects that have lived in memory longer than a certain “age”. The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less stopped to allow the garbage collector thread to effectively do its work.
- Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

What is the meaning of the term “stop-the-world”?

- When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete.
- Depending on the needs of an application, “stop the world” garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

What is generational garbage collection and what makes it a popular garbage collection approach?

- Generational garbage collection can be loosely defined as the strategy used by the garbage collector where the heap is divided into a number of sections called generations, each of which will hold objects according to their “age” on the heap.
- Whenever the garbage collector is running, the first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned.
- As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived.
- With generational garbage collection, objects are grouped according to their “age” in terms of how many garbage collection cycles they have survived. This way, the bulk of the work spread across various minor and major collection cycles.
- Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

When does an object become eligible for garbage collection? Describe how the GC collects an eligible object?

- An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.
- The most straightforward case of an object becoming eligible for garbage collection is if all its references are

frequently than in the young generation.

- The permanent generation or more commonly called, PermGen, contains metadata required by the JVM to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.
- First, any new objects are allocated to the Eden space. Both survivor spaces start out empty. When the Eden space fills up, a minor garbage collection is triggered. Referenced objects are moved to the first survivor space. Unreferenced objects are deleted.
- During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1).
- In addition, objects from the last minor GC in the first survivor space (S0) have their age incremented and are moved to S1. Once all surviving objects have been moved to S1, both S0 and Eden space are cleared. At this point, S1 contains objects with different ages.
- At the next minor GC, the same process is repeated. However this time the survivor spaces switch. Referenced objects are moved to S0 from both Eden and S1. Surviving objects are aged. Eden and S1 are cleared.
- After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.
- This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space. For each major GC, there are several minor GCs.

Describe strong, weak, soft and phantom references and their role in garbage collection.

- Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to minimize latency and maximize throughput, in critical applications. Much as it is impossible to explicitly control when garbage collection is triggered in the JVM, it is possible to influence how it occurs as regards the objects we have created.
- Java provides us with reference objects to control the relationship between the objects we create and the garbage collector.
- By default, every object we create in a Java program is null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.
- Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

How do you trigger garbage collection from Java code?

- You, as Java programmer, can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.
- Before removing an object from memory garbage collection thread invokes finalize()method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.
- Additionally, there are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it's not guaranteed that garbage collection will happen.

What happens when there is not enough heap space to accommodate storage of new objects?

- If there is no memory space for creating a new object in Heap, Java Virtual Machine throws OutOfMemoryError or more specifically java.lang.OutOfMemoryError heap space.

Is it possible to «resurrect» an object that became eligible for garbage collection?

- When an object becomes eligible for garbage collection, the GC has to run the finalize method on it. The finalize method is guaranteed to run only once, thus the GC flags the object as finalized and gives it a rest until the next cycle.
- In the finalize method you can technically "resurrect" an object, for example, by assigning it to a static field. The object would become alive again and non-eligible for garbage collection, so the GC would not collect it during the next cycle.
- The object, however, would be marked as finalized, so when it would become eligible again, the finalize method would not be called. In essence, you can turn this "resurrection" trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

strongly referenced by a variable:
StringBuilder sb = new StringBuilder();

- In the above snippet, the new keyword creates a new StringBuilder object and stores it on the heap. The variable sb then stores a strong reference to this object. What this means for the garbage collector is that the particular StringBuilder object is not eligible for collection at all due to a strong reference held to it by sb. The story only changes when we nullify sb like this:
  sb = null;
- After calling the above line, the object will then be eligible for collection.
- We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside java.lang.ref package.
- A soft reference can be created to the above object like this:
  StringBuilder sb = new StringBuilder();
  SoftReference<StringBuilder> sbRef = new SoftReference<>(sb);
  sb = null;
- In the above snippet, we have created two references to the StringBuilder object. The first line creates a strong reference sb and the second line creates a soft reference sbRef. The third line should make the object eligible for collection but the garbage collector will postpone collecting it because of sbRef.

- The story will only change when memory becomes tight and the JVM is on the brink of throwing an OutOfMemory error. In other words, objects with only soft references are collected as a last resort to recover memory.

- A weak reference can be created in a similar manner using WeakReference class. When sb is set to null and the StringBuilder object only has a weak reference, the JVM's garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

- A phantom reference is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was collected.

Suppose we have a circular reference (two objects that reference each other). Could such pair of objects become eligible for garbage collection and why?

- Yes, a pair of objects with a circular reference can become eligible for garbage collection.
- This is because of how Java's garbage collector handles circular references.
- It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static field).
- If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

How are strings represented in memory?

- A String instance in Java is an object with two fields: a char[] value field and an int hash field. The value field is an array of chars representing the string itself, and the hash field contains the hashCode of a string which is initialized with zero, calculated during the first hashCode() call and cached ever since. As a curious edge case, if a hashCode of a string has a zero value, it has to be recalculated each time the hashCode() is called.
- Important thing is that a String instance is immutable: you can't get or modify the underlying char[] array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical String objects in your source code, they are all represented by a single instance at runtime.

What is a StringBuilder and what are its use cases? What is the difference between appending a string to a StringBuilder and concatenating two strings with a + operator? How does StringBuilder differ from StringBuffer?

- StringBuilder allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the String class which is immutable.
- When concatenating two String instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. StringBuilder allows handling string manipulations much more efficiently.
- StringBuffer is different from StringBuilder in that it is thread-safe. If you need to manipulate a string in a single thread, use StringBuilder instead.

Describe the meaning of the final keyword when applied to a class, method, field or a local variable.

- The final keyword has multiple different meanings

What is a default method?

- Prior to Java 8, interfaces could only have abstract methods, i.e. methods without a body. Starting with

when applied to different language constructs:

- A final class is a class that cannot be subclassed
- A final method is a method that cannot be overridden in subclasses
- A final field is a field that has to be initialized in the constructor or initializer block and cannot be modified after that
- A final variable is a variable that may be assigned (and has to be assigned) only once and is never modified after that

What are static class members?

- Static fields and methods of a class are not bound to a specific instance of a class.
- Instead, they are bound to the class object itself.
- The call of a static method or addressing a static field is resolved at compile time because, contrary to instance methods and fields, we don't need to walk the reference and determine an actual object we're referring to.

May a class be declared abstract if it does not have any abstract members? What could be the purpose of such class?

- Yes, a class can be declared abstract even if it does not contain any abstract members.
- As an abstract class, it cannot be instantiated, but it can serve as a root object of some hierarchy, providing methods that can be useful to its implementations.

Java 8, interface methods can have a default implementation. If an implementing class does not override this method, then the default implementation is used. Such methods are suitably marked with a default keyword.

- One of the prominent use cases of a default method is adding a method to an existing interface. If you don't mark such interface method as default, then all existing implementations of this interface will break. Adding a method with a default implementation ensures binary compatibility of legacy code with the new version of this interface.
- A good example of this is the Iterator interface which allows a class to be a target of the for-each loop. This interface first appeared in Java 5, but in Java 8 it received two additional methods, forEach, and spliterator. They are defined as default methods with implementations and thus do not break backward compatibility:

```
public interface Iterable<T> {
   Iterator<T> iterator();

   default void forEach(Consumer<? super T> action) {
/* */ }

   default Spliterator<T> spliterator() { /* */ }
```

What is Constructor Chaining?

- Constructor chaining is a way of simplifying object construction by providing multiple constructors that call each other in sequence.
- The most specific constructor may take all possible arguments and may be used for the most detailed object configuration. A less specific constructor may call the more specific constructor by providing some of its arguments with default values. At the top of the chain, a no-argument constructor could instantiate an object with default values.
- Here's an example with a class that models a discount in percents that are available within a certain amount of days. The default values of 10% and 2 days are used if we don't specify them when using a no-arg constructor:

```
public class Discount {
   private int percent;
   private int days;

   public Discount() {
      this(10);
   }

   public Discount(int percent) {
```

What is overriding and overloading of methods? How are they different?

- Overriding of a method is done in a subclass when you define a method with the same signature as in superclass. This allows the runtime to pick a method depending on the actual object type that you call the method on. Methods toString, equals, and hashCode are overridden quite often in subclasses.
- Overloading of a method happens in the same class. Overloading occurs when you create a method with the same name but with different types or number of arguments. This allows you to execute a certain code depending on the types of arguments you provide, while the name of the method remains the same.
- Here's an example of overloading in the java.io.Writer abstract class. The following methods are both named write, but one of them receives an int while another receives a char array.

```
public abstract class Writer {
   public void write(int c) throws IOException {
      // ...
   }

   public void write(char cbuf[]) throws IOException {
      // ...
```

```
        this(percent, 2);
    }

    public Discount(int percent, int days) {
        this.percent = percent;
        this.days = days;
    }
}
```

What is an immutable class, and how can you create one?

- An instance of an immutable class cannot be changed after it's created. By changing we mean mutating the state by modifying the values of the fields of the instance. Immutable classes have many advantages: they are thread-safe, and it is much easier to reason about them when you have no mutable state to consider.
- To make a class immutable, you should ensure the following:
  - All fields should be declared private and final; this infers that they should be initialized in the constructor and not changed ever since;
  - The class should have no setters or other methods that mutate the values of the fields;
  - All fields of the class that were passed via constructor should either be also immutable, or their values should be copied before field initialization (or else we could change the state of this class by holding on to these values and modifying them);
  - The methods of the class should not be overridable; either all methods should be final, or the constructor should be private and only invoked via static factory method.

What is a marker interface? What are the notable examples of marker interfaces in Java?

- A marker interface is an interface without any methods. It is usually implemented by a class or extended by another interface to signify a certain property. The most widely known marker interfaces in standard Java library are the following:
  - Serializable is used to explicitly express that this class can be serialized;
  - Cloneable allows cloning objects using the clone method (without Cloneable interface in place, this method throws a CloneNotSupportedException);
  - Remote is used in RMI to specify an interface which methods could be called remotely.

What is a var-arg? What are the restrictions on a var-arg? How can you use it inside the method body?

- Var-arg is a variable-length argument for a method. A method may have only one var-arg, and it has to come

```
    }
}
```

Can you override a static method?

- No, you can't. By definition, you can only override a method if its implementation is determined at runtime by the type of the actual instance (a process known as the dynamic method lookup).
- The static method implementation is determined at compile time using the type of the reference, so overriding would not make much sense anyway. Although you can add to subclass a static method with the exact same signature as in superclass, this is not technically overriding.

How do you compare two enum values: with equals() or with ==?

- Actually, you can use both.
- The enum values are objects, so they can be compared with equals(), but they are also implemented as static constants under the hood, so you might as well compare them with ==.
- This is mostly a matter of code style, but if you want to save character space (and possibly skip an unneeded method call), you should compare enums with ==.

What is an initializer block? What is a static initializer block?

- An initializer block is a curly-braced block of code in the class scope which is executed during the instance creation. You can use it to initialize fields with something more complex than in-place initialization one-liners.
- Actually, the compiler just copies this block inside every constructor, so it is a nice way to extract common code from all constructors.
- A static initializer block is a curly-braced block of code with the static modifier in front of it. It is executed once during the class loading and can be used for initializing static fields or for some side effects.

What is a singleton and how can it be implemented in Java?

- Singleton is a pattern of object-oriented programming. A singleton class may only have one instance, usually globally visible and accessible.
- There are multiple ways of creating a singleton in Java. The following is the most simple example with a static field that is initialized in-place. The initialization is thread-safe because static fields are guaranteed to be initialized in a thread-safe manner. The constructor is private, so there is no way for outer code to create more than one instance of the class.

```
public class SingletonExample {
    private static SingletonExample instance = new
SingletonExample();
```

last in the list of arguments. It is specified as a type name followed by an ellipsis and an argument name. Inside the method body, a var-arg is used as an array of specified type.

- Here's an example from the standard library — the Collections.addAll method that receives a collection, a variable number of elements, and adds all elements to the collection:

```
public static <T> boolean addAll(
  Collection<? super T> c, T... elements) {
    boolean result = false;
    for (T element : elements)
        result |= c.add(element);
    return result;
}
```

Can you access an overridden method of a superclass? Can you access an overridden method of a super-superclass in a similar way?

- To access an overridden method of a superclass, you can use the super keyword. But you don't have a similar way of accessing the overridden method of a super-superclass.
- As an example from the standard library, LinkedHashMap class extends HashMap and mostly re-uses its functionality, adding a linked list over its values to preserve iteration order. LinkedHashMap re-uses the clear method of its superclass and then clears head and tail references of its linked list:

```
public void clear() {
    super.clear();
    head = tail = null;
}
```

How can you handle an exception?

- By using a try-catch-finally statement:

```
try {
    // ...
} catch (ExceptionType1 ex) {
    // ...
} catch (ExceptionType2 ex) {
    // ...
} finally {
    // ...
}
```

- The block of code in which an exception may occur is enclosed in a try block. This block is also called "protected" or "guarded" code.
- If an exception occurs, the catch block that matches the exception being thrown is executed, if not, all catch blocks are ignored.

```
    private SingletonExample() {}

    public static SingletonExample getInstance() {
        return instance;
    }
}
```

- But this approach could have a serious drawback — the instance would be instantiated when this class is first accessed. If initialization of this class is a heavy operation, and we would probably like to defer it until the instance is actually needed (possibly never), but at the same time keep it thread-safe. In this case, we should use a technique known as double-checked locking.

What is an exception?

- An exception is an abnormal event that occurs during the execution of a program and disrupts the normal flow of the program's instructions.

What is the purpose of the throw and throws keywords?

- The throws keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method:

```
public void simpleMethod() throws Exception {
    // ...
}
```

- The throw keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition:

```
if (task.isTooComplicated()) {
    throw new TooComplicatedException("The task is too complicated");
}
```

How can you catch multiple exceptions?

- There are three ways of handling multiple exceptions in a block of code.
- The first is to use a catch block that can handle all exception types being thrown:

```
try {
    // ...
} catch (Exception ex) {
    // ...
}
```

- You should keep in mind that the recommended practice is to use exception handlers that are as

- The finally block is always executed after the try block exits, whether an exception was thrown or not inside it.

What is the difference between a checked and an unchecked exception?

- A checked exception must be handled within a try-catch block or declared in a throws clause; whereas an unchecked exception is not required to be handled nor declared.
- Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively.
- All exceptions are checked exceptions, except those indicated by Error, RuntimeException, and their subclasses.

What is the difference between an exception and error?

- An exception is an event that represents a condition from which is possible to recover, whereas error represents an external situation usually impossible to recover from.
- All errors thrown by the JVM are instances of Error or one of its subclasses, the more common ones include but are not limited to:
- OutOfMemoryError – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available
- StackOverflowError – occurs when the stack space for a thread has run out, typically because an application recurses too deeply
- ExceptionInInitializerError – signals that an unexpected exception occurred during the evaluation of a static initializer
- NoClassDefFoundError – is thrown when the classloader tries to load the definition of a class and couldn't find it, usually because the required class files were not found in the classpath
- UnsupportedClassVersionError – occurs when the JVM attempts to read a class file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java
- Although an error can be handled with a try statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown.

Why would you want to subclass an exception?

- If the exception type isn't represented by those that already exist in the Java platform, or if you need to provide more information to client code to treat it in a more precise manner, then you should create a custom exception.
- Deciding whether a custom exception should be accurate as possible.
- Exception handlers that are too broad can make your code more error-prone, catch exceptions that weren't anticipated, and cause unexpected behavior in your program.

- The second way is implementing multiple catch blocks:

```
try {
   // ...
} catch (FileNotFoundException ex) {
   // ...
} catch (EOFException ex) {
   // ...
}
```

- Note that, if the exceptions have an inheritance relationship; the child type must come first and the parent type later. If we fail to do this, it will result in a compilation error.

- The third is to use a multi-catch block:

```
try {
   // ...
} catch (FileNotFoundException | EOFException ex) {
   // ...
}
```

- This feature, first introduced in Java 7; reduces code duplication and makes it easier to maintain.

What is exception chaining?

- Occurs when an exception is thrown in response to another exception. This allows us to discover the complete history of our raised problem:

```
try {
   task.readConfigFile();
} catch (FileNotFoundException ex) {
   throw new TaskException("Could not perform task",
ex);
}
```

What is a stacktrace and how does it relate to an exception?

- A stack trace provides the names of the classes and methods that were called, from the start of the application to the point an exception occurred.
- It's a very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it.

What are some advantages of exceptions?

- Traditional error detection and handling techniques often lead to spaghetti code hard to maintain and difficult to read. However, exceptions enable us to separate the core logic of our application from the details of what to do when something unexpected

checked or unchecked depends entirely on the business case. However, as a rule of thumb; if the code using your exception can be expected to recover from it, then create a checked exception otherwise make it unchecked.

- Also, you should inherit from the most specific Exception subclass that closely relates to the one you want to throw. If there is no such class, then choose Exception as the parent.

Can you throw any exception inside a lambda expression's body?

- When using a standard functional interface already provided by Java, you can only throw unchecked exceptions because standard functional interfaces do not have a "throws" clause in method signatures:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> {
   if (i == 0) {
      throw new IllegalArgumentException("Zero not
allowed");
   }
   System.out.println(Math.PI / i);
});
```

- However, if you are using a custom functional interface, throwing checked exceptions is possible:

```
@FunctionalInterface
public static interface CheckedFunction<T> {
   void apply(T t) throws Exception;
}

public void processTasks(
  List<Task> taks, CheckedFunction<Task>
checkedFunction) {
   for (Task task : taks) {
      try {
         checkedFunction.apply(task);
      } catch (Exception e) {
         // ...
      }
   }
}

processTasks(taskList, t -> {
   // ...
   throw new Exception("Something happened");
});
```

Is there any way of throwing a checked exception from a method that does not have a throws clause?

- Yes. We can take advantage of the type erasure performed by the compiler and make it think we are happens.
- Also, since the JVM searches backward through the call stack to find any methods interested in handling a particular exception; we gain the ability to propagate an error up in the call stack without writing additional code.
- Also, because all exceptions thrown in a program are objects, they can be grouped or categorized based on its class hierarchy. This allows us to catch a group exceptions in a single exception handler by specifying the exception's superclass in the catch block.

What are the rules we need to follow when overriding a method that throws an exception?

- Several rules dictate how exceptions must be declared in the context of inheritance.
- When the parent class method doesn't throw any exceptions, the child class method can't throw any checked exception, but it may throw any unchecked.
- Here's an example code to demonstrate this:

```
class Parent {
   void doSomething() {
      // ...
   }
}

class Child extends Parent {
   void doSomething() throws
IllegalArgumentException {
      // ...
   }
}
```

- The next example will fail to compile since the overriding method throws a checked exception not declared in the overridden method:

```
class Parent {
   void doSomething() {
      // ...
   }
}

class Child extends Parent {
   void doSomething() throws IOException {
      // Compilation error
   }
}
```

- When the parent class method throws one or more checked exceptions, the child class method can throw any unchecked exception; all, none or a subset of the declared checked exceptions, and even a greater number of these as long as they have the same scope

throwing an unchecked exception, when, in fact; we're throwing a checked exception:

```
public <T extends Throwable> T
sneakyThrow(Throwable ex) throws T {
   throw (T) ex;
}

public void methodWithoutThrows() {
   this.<RuntimeException>sneakyThrow(new
Exception("Checked Exception"));
}
```

What are annotations? What are their typical use cases?

- Annotations are metadata bound to elements of the source code of a program and have no effect on the operation of the code they operate.
- Their typical uses cases are:
    - Information for the compiler – with annotations, the compiler can detect errors or suppress warnings
    - Compile-time and deployment-time processing – software tools can process annotations and generate code, configuration files, etc.
    - Runtime processing – annotations can be examined at runtime to customize the behavior of a program

Describe some useful annotations from the standard library.

- There are several annotations in the java.lang and java.lang.annotation packages, the more common ones include but not limited to:
    - @Override – marks that a method is meant to override an element declared in a superclass. If it fails to override the method correctly, the compiler will issue an error
    - @Deprecated – indicates that element is deprecated and should not be used. The compiler will issue a warning if the program uses a method, class, or field marked with this annotation
    - @SuppressWarnings – tells the compiler to suppress specific warnings. Most commonly used when interfacing with legacy code written before generics appeared
    - @FunctionalInterface – introduced in Java 8, indicates that the type declaration is a functional interface and whose implementation can be provided using a Lambda Expression

How can you create an annotation?

or narrower.

- Here's an example code that successfully follows the previous rule:

```
class Parent {
   void doSomething() throws IOException,
ParseException {
      // ...
   }

   void doSomethingElse() throws IOException {
      // ...
   }
}

class Child extends Parent {
   void doSomething() throws IOException {
      // ...
   }

   void doSomethingElse() throws
FileNotFoundException, EOFException {
      // ...
   }
}
```

- Note that both methods respect the rule. The first throws fewer exceptions than the overridden method, and the second, even though it throws more; they're narrower in scope.
- However, if we try to throw a checked exception that the parent class method doesn't declare or we throw one with a broader scope; we'll get a compilation error:

```
class Parent {
   void doSomething() throws FileNotFoundException {
      // ...
   }
}

class Child extends Parent {
   void doSomething() throws IOException {
      // Compilation error
   }
}
```

- When the parent class method has a throws clause with an unchecked exception, the child class method can throw none or any number of unchecked exceptions, even though they are not related.
- Here's an example that honors the rule:

```
class Parent {
   void doSomething() throws
```

- Annotations are a form of an interface where the keyword interface is preceded by @, and whose body contains annotation type element declarations that look very similar to methods:

```
public @interface SimpleAnnotation {
   String value();

   int[] types();
}
```

- After the annotation is defined, yon can start using it in through your code:

```
@SimpleAnnotation(value = "an element", types = 1)
public class Element {
   @SimpleAnnotation(value = "an attribute", types = { 1, 2 })
   public Element nextElement;
}
```

- Note that, when providing multiple values for array elements, you must enclose them in brackets.
- Optionally, default values can be provided as long as they are constant expressions to the compiler:

```
public @interface SimpleAnnotation {
   String value() default "This is an element";

   int[] types() default { 1, 2, 3 };
}
```

- Now, you can use the annotation without those elements:

```
@SimpleAnnotation
public class Element {
   // ...
}
```

- Or only some of them:

```
@SimpleAnnotation(value = "an attribute")
public Element nextElement;
```

```
IllegalArgumentException {
      // ...
   }
}

class Child extends Parent {
   void doSomething()
    throws ArithmeticException,
BufferOverflowException {
      // ...
   }
}
```

What object types can be returned from an annotation method declaration?

- The return type must be a primitive, String, Class, Enum, or an array of one of the previous types. Otherwise, the compiler will throw an error.
- Here's an example code that successfully follows this principle:

```
enum Complexity {
   LOW, HIGH
}

public @interface ComplexAnnotation {
   Class<? extends Object> value();

   int[] types();

   Complexity complexity();
}
```

- The next example will fail to compile since Object is not a valid return type:

```
public @interface FailingAnnotation {
   Object complexity();
}
```

Which program elements can be annotated?

- Annotations can be applied in several places throughout the source code. They can be applied to declarations of classes, constructors, and fields:

```
@SimpleAnnotation
public class Apply {
   @SimpleAnnotation
   private String aField;
```

Is there a way to limit the elements in which an annotation can be applied?

- Yes, the @Target annotation can be used for this purpose. If we try to use an annotation in a context where it is not applicable, the compiler will issue an error.
- Here's an example to limit the usage of the @SimpleAnnotation annotation to field declarations only:

```
    @SimpleAnnotation
    public Apply() {
        // ...
    }
}
```

- Methods and their parameters:

```
@SimpleAnnotation
public void aMethod(@SimpleAnnotation String
param) {
    // ...
}
```

- Local variables, including a loop and resource variables:

```
@SimpleAnnotation
int i = 10;

for (@SimpleAnnotation int j = 0; j < i; j++) {
    // ...
}

try (@SimpleAnnotation FileWriter writer =
getWriter()) {
    // ...
} catch (Exception ex) {
    // ...
}
```

Other annotation types:

```
@SimpleAnnotation
public @interface ComplexAnnotation {
    // ...
}
```

- And even packages, through the package-info.java file:

```
@PackageAnnotation
package com.baeldung.interview.annotations;
```

- As of Java 8, they can also be applied to the use of types. For this to work, the annotation must specify an @Target annotation with a value of ElementType.USE:

```
@Target(ElementType.TYPE_USE)
public @interface SimpleAnnotation {
    // ...
}
```

- Now, the annotation can be applied to class instance creation:

```
new @SimpleAnnotation Apply();
```

Type casts:

```
aString = (@SimpleAnnotation String) something;
```

Implements clause:

```
@Target(ElementType.FIELD)
public @interface SimpleAnnotation {
    // ...
}
```

- We can pass multiple constants if we want to make it applicable in more contexts:

```
@Target({ ElementType.FIELD, ElementType.METHOD,
ElementType.PACKAGE })
```

- We can even make an annotation so it cannot be used to annotate anything. This may come in handy when the declared types are intended solely for use as a member type in complex annotations:

```
@Target({})
public @interface NoTargetAnnotation {
    // ...
}
```

What are meta-annotations?

- Are annotations that apply to other annotations.
- All annotations that aren't marked with @Target, or are marked with it but include ANNOTATION_TYPE constant are also meta-annotations:

```
@Target(ElementType.ANNOTATION_TYPE)
public @interface SimpleAnnotation {
    // ...
}
```

What are repeating annotations?

- These are annotations that can be applied more than once to the same element declaration.
- For compatibility reasons, since this feature was introduced in Java 8, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. For the compiler to do this, there are two steps to declared them.
- First, we need to declare a repeatable annotation:

```
@Repeatable(Schedules.class)
public @interface Schedule {
    String time() default "morning";
}
```

- Then, we define the containing annotation with a mandatory value element, and whose type must be an array of the repeatable annotation type:

```
public @interface Schedules {
    Schedule[] value();
}
```

```
public class SimpleList<T>
  implements @SimpleAnnotation
List<@SimpleAnnotation T> {
   // ...}
```

- And throws clause:

```
void aMethod() throws @SimpleAnnotation Exception
{
   // ...}
```

- Now, we can use @Schedule multiple times:

```
@Schedule
@Schedule(time = "afternoon")
@Schedule(time = "night")
void scheduledMethod() {
   // ...
}
```

How can you retrieve annotations? How does this relate to its retention policy?

- You can use the Reflection API or an annotation processor to retrieve annotations.
- The @Retention annotation and its RetentionPolicy parameter affect how you can retrieve them. There are three constants in RetentionPolicy enum:
- RetentionPolicy.SOURCE – makes the annotation to be discarded by the compiler but annotation processors can read them
- RetentionPolicy.CLASS – indicates that the annotation is added to the class file but not accessible through reflection
- RetentionPolicy.RUNTIME –Annotations are recorded in the class file by the compiler and retained by the JVM at runtime so that they can be read reflectively
- Here's an example code to create an annotation that can be read at runtime:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Description {
   String value();
}
```

- Now, annotations can be retrieved through reflection:

```
Description description
  =
AnnotatedClass.class.getAnnotation(Description.class)
;
System.out.println(description.value());
```

- An annotation processor can work with RetentionPolicy.SOURCE, this is described in the article Java Annotation Processing and Creating a Builder.
- RetentionPolicy.CLASS is usable when you're writing a Java bytecode parser.

Describe the different states of a Thread and when do the state transitions occur.

- The state of a Thread can be checked using the Thread.getState() method. Different states of a Thread are described in the Thread.State enum. They are:

Is it possible to extend annotations?

- No. Annotations always extend java.lang.annotation.Annotation, as stated in the Java Language Specification.

What is the difference between a process and a thread?

- Both processes and threads are units of concurrency, but they have a fundamental difference: processes do not share a common memory, while threads do.
- From the operating system's point of view, a process is an independent piece of software that runs in its own virtual memory space. Any multitasking operating system (which means almost any modern operating system) has to separate processes in memory so that one failing process wouldn't drag all other processes down by scrambling common memory.
- The processes are thus usually isolated, and they cooperate by the means of inter-process communication which is defined by the operating system as a kind of intermediate API.
- On the contrary, a thread is a part of an application that shares a common memory with other threads of the same application. Using common memory allows to shave off lots of overhead, design the threads to cooperate and exchange data between them much faster.

How can you create a Thread instance and run it?

- To create an instance of a thread, you have two options. First, pass a Runnable instance to its constructor and call start(). Runnable is a functional interface, so it can be passed as a lambda expression:

```
Thread thread1 = new Thread(() ->
  System.out.println("Hello World from Runnable!"));
thread1.start();
```

- Thread also implements Runnable, so another way of starting a thread is to create an anonymous subclass, override its run() method, and then call start():

```
Thread thread2 = new Thread() {
   @Override
```

- NEW — a new Thread instance that was not yet started via Thread.start()
- RUNNABLE — a running thread. It is called runnable because at any given time it could be either running or waiting for the next quantum of time from the thread scheduler. A NEW thread enters the RUNNABLE state when you call Thread.start() on it
- BLOCKED — a running thread becomes blocked if it needs to enter a synchronized section but cannot do that due to another thread holding the monitor of this section
- WAITING — a thread enters this state if it waits for another thread to perform a particular action. For instance, a thread enters this state upon calling the Object.wait() method on a monitor it holds, or the Thread.join() method on another thread
- TIMED_WAITING — same as the above, but a thread enters this state after calling timed versions of Thread.sleep(), Object.wait(), Thread.join() and some other methods
- TERMINATED — a thread has completed the execution of its Runnable.run() method and terminated

What is the Thread's interrupt flag? How can you set and check it? How does it relate to the InterruptedException?

- The interrupt flag, or interrupt status, is an internal Thread flag that is set when the thread is interrupted. To set it, simply call thread.interrupt() on the thread object.
- If a thread is currently inside one of the methods that throw InterruptedException (wait, join, sleep etc.), then this method immediately throws InterruptedException. The thread is free to process this exception according to its own logic.
- If a thread is not inside such method and thread.interrupt() is called, nothing special happens. It is thread's responsibility to periodically check the interrupt status using static Thread.interrupted() or instance isInterrupted() method. The difference between these methods is that the static Thread.interrupt() clears the interrupt flag, while isInterrupted() does not.

What are Executor and ExecutorService? What are the differences between these interfaces?

- Executor and ExecutorService are two related interfaces of java.util.concurrent framework. Executor is a very simple interface with a single execute method accepting Runnable instances for execution. In most cases, this is the interface that your task-executing code should depend on.
- ExecutorService extends the Executor interface with

```
    public void run() {
      System.out.println("Hello World from subclass!");
    }
  };
  thread2.start();
```

What is the difference between the Runnable and Callable interfaces? How are they used?

- The Runnable interface has a single run method. It represents a unit of computation that has to be run in a separate thread. The Runnable interface does not allow this method to return value or to throw unchecked exceptions.
- The Callable interface has a single call method and represents a task that has a value. That's why the call method returns a value. It can also throw exceptions. Callable is generally used in ExecutorService instances to start an asynchronous task and then call the returned Future instance to get its value.

What is a daemon thread, what are its use cases? How can you create a daemon thread?

- A daemon thread is a thread that does not prevent JVM from exiting. When all non-daemon threads are terminated, the JVM simply abandons all remaining daemon threads. Daemon threads are usually used to carry out some supportive or service tasks for other threads, but you should take into account that they may be abandoned at any time.
- To start a thread as a daemon, you should use the setDaemon() method before calling start():

```
Thread daemon = new Thread(()
  -> System.out.println("Hello from daemon!"));
daemon.setDaemon(true);
daemon.start();
```

- Curiously, if you run this as a part of the main() method, the message might not get printed. This could happen if the main() thread would terminate before the daemon would get to the point of printing the message. You generally should not do any I/O in daemon threads, as they won't even be able to execute their finally blocks and close the resources if abandoned.

What are the available implementations of ExecutorService in the standard library?

- The ExecutorService interface has three standard implementations:
  - ThreadPoolExecutor — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.

multiple methods for handling and checking the lifecycle of a concurrent task execution service (termination of tasks in case of shutdown) and methods for more complex asynchronous task handling including Futures.

What is Java Memory Model (JMM)? Describe its purpose and basic ideas.

- It specifies how multiple threads access common memory in a concurrent Java application, and how data changes by one thread are made visible to other threads. While being quite short and concise, JMM may be hard to grasp without strong mathematical background.
- The need for memory model arises from the fact that the way your Java code is accessing data is not how it actually happens on the lower levels. Memory writes and reads may be reordered or optimized by the Java compiler, JIT compiler, and even CPU, as long as the observable result of these reads and writes is the same.
- This can lead to counter-intuitive results when your application is scaled to multiple threads because most of these optimizations take into account a single thread of execution (the cross-thread optimizers are still extremely hard to implement). Another huge problem is that the memory in modern systems is multilayered: multiple cores of a processor may keep some non-flushed data in their caches or read/write buffers, which also affects the state of the memory observed from other cores.\To make things worse, the existence of different memory access architectures would break the Java's promise of "write once, run everywhere". Happily for the programmers, the JMM specifies some guarantees that you may rely upon when designing multithreaded applications. Sticking to these guarantees helps a programmer to write multithreaded code that is stable and portable between various architectures.

The main notions of JMM are:

- Actions, these are inter-thread actions that can be executed by one thread and detected by another thread, like reading or writing variables, locking/unlocking monitors and so on
- Synchronization actions, a certain subset of actions, like reading/writing a volatile variable, or locking/unlocking a monitor
- Program Order (PO), the observable total order of actions inside a single thread
- Synchronization Order (SO), the total order between all synchronization actions — it has to be consistent with Program Order, that is, if two synchronization actions come one before another in PO, they occur in the same order in SO
    - ScheduledThreadPoolExecutor allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
- ForkJoinPool is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

What is a volatile field and what guarantees does the JMM hold for such field?

- A volatile field has special properties according to the Java Memory Model (see Q9). The reads and writes of a volatile variable are synchronization actions, meaning that they have a total ordering (all threads will observe a consistent order of these actions). A read of a volatile variable is guaranteed to observe the last write to this variable, according to this order.
- If you have a field that is accessed from multiple threads, with at least one thread writing to it, then you should consider making it volatile, or else there is a little guarantee to what a certain thread would read from this field.
- Another guarantee for volatile is atomicity of writing and reading 64-bit values (long and double). Without a volatile modifier, a read of such field could observe a value partly written by another thread.

Which of the following operations are atomic?

- writing to a non-volatile int;
- writing to a volatile int;
- writing to a non-volatile long;
- writing to a volatile long;

incrementing a volatile long?

- A write to an int (32-bit) variable is guaranteed to be atomic, whether it is volatile or not. A long (64-bit) variable could be written in two separate steps, for example, on 32-bit architectures, so by default, there is no atomicity guarantee. However, if you specify the volatile modifier, a long variable is guaranteed to be accessed atomically.
- The increment operation is usually done in multiple steps (retrieving a value, changing it and writing back), so it is never guaranteed to be atomic, wether the variable is volatile or not. If you need to implement atomic increment of a value, you should use classes AtomicInteger, AtomicLong etc.

What special guarantees does the JMM hold for final fields of a class?

- synchronizes-with (SW) relation between certain synchronization actions, like unlocking of monitor and locking of the same monitor (in another or the same thread)
- Happens-before Order — combines PO with SW (this is called transitive closure in set theory) to create a partial ordering of all actions between threads. If one action happens-before another, then the results of the first action are observable by the second action (for instance, write of a variable in one thread and read in another)
- Happens-before consistency — a set of actions is HB-consistent if every read observes either the last write to that location in the happens-before order, or some other write via data race
- Execution — a certain set of ordered actions and consistency rules between them

- For a given program, we can observe multiple different executions with various outcomes.
- But if a program is correctly synchronized, then all of its executions appear to be sequentially consistent, meaning you can reason about the multithreaded program as a set of actions occurring in some sequential order.
- This saves you the trouble of thinking about under-the-hood reorderings, optimizations or data caching.

If two threads call a synchronized method on different object instances simultaneously, could one of these threads block? What if the method is static?

- If the method is an instance method, then the instance acts as a monitor for the method. Two threads calling the method on different instances acquire different monitors, so none of them gets blocked.
- If the method is static, then the monitor is the Class object. For both threads, the monitor is the same, so one of them will probably block and wait for another to exit the synchronized method.

Describe the conditions of deadlock, livelock, and starvation. Describe the possible causes of these conditions.

- Deadlock is a condition within a group of threads that cannot make progress because every thread in the group has to acquire some resource that is already acquired by another thread in the group. The most simple case is when two threads need to lock both of two resources to progress, the first resource is already locked by one thread, and the second by another. These threads will never acquire a lock to both resources and thus will never progress.
- Livelock is a case of multiple threads reacting to conditions, or events, generated by themselves. An

- JVM basically guarantees that final fields of a class will be initialized before any thread gets hold of the object. Without this guarantee, a reference to an object may be published, i.e. become visible, to another thread before all the fields of this object are initialized, due to reorderings or other optimizations. This could cause racy access to these fields.
- This is why, when creating an immutable object, you should always make all its fields final, even if they are not accessible via getter methods.

What is the meaning of a synchronized keyword in the definition of a method? Of a static method? Before a block?

- The synchronized keyword before a block means that any thread entering this block has to acquire the monitor (the object in brackets). If the monitor is already acquired by another thread, the former thread will enter the BLOCKED state and wait until the monitor is released.

```
synchronized(object) {
  // ...
}
```

- A synchronized instance method has the same semantics, but the instance itself acts as a monitor.

```
synchronized void instanceMethod() {
  // ...
}
```

- For a static synchronized method, the monitor is the Class object representing the declaring class.

```
static synchronized void staticMethod() {
  // ...
}
```

What is the purpose of the wait, notify and notifyAll methods of the Object class?

- A thread that owns the object's monitor (for instance, a thread that has entered a synchronized section guarded by the object) may call object.wait() to temporarily release the monitor and give other threads a chance to acquire the monitor. This may be done, for instance, to wait for a certain condition.
- When another thread that acquired the monitor fulfills the condition, it may call object.notify() or object.notifyAll() and release the monitor. The notify method awakes a single thread in the waiting state, and the notifyAll method awakes all threads that wait for this monitor, and they all compete for re-acquiring the lock.
- The following BlockingQueue implementation shows how multiple threads work together via the

event occurs in one thread and has to be processed by another thread. During this processing, a new event occurs which has to be processed in the first thread, and so on. Such threads are alive and not blocked, but still, do not make any progress because they overwhelm each other with useless work.
- Starvation is a case of a thread unable to acquire resource because other thread (or threads) occupy it for too long or have higher priority. A thread cannot make progress and thus is unable to fulfill useful work.

Describe the purpose and use-cases of the fork/join framework.

- The fork/join framework allows parallelizing recursive algorithms. The main problem with parallelizing recursion using something like ThreadPoolExecutor is that you may quickly run out of threads because each recursive step would require its own thread, while the threads up the stack would be idle and waiting.
- The fork/join framework entry point is the ForkJoinPool class which is an implementation of ExecutorService. It implements the work-stealing algorithm, where idle threads try to "steal" work from busy threads. This allows to spread the calculations between different threads and make progress while using fewer threads than it would require with a usual thread pool.

What new features were added in Java 8?

Java 8 ships with several new features but the most significant are the following:

- Lambda Expressions − a new language feature allowing treating actions as objects
- Method References − enable defining Lambda Expressions by referring to methods directly using their names
- Optional − special wrapper class used for expressing optionality
- Functional Interface – an interface with maximum one abstract method, implementation can be provided using a Lambda Expression
- Default methods − give us the ability to add full implementations in interfaces besides abstract methods
- Nashorn, JavaScript Engine − Java-based engine for executing and evaluating JavaScript code
- Stream API − a special iterator class that allows processing collections of objects in a functional manner
- Date API − an improved, immutable JodaTime-inspired Date API
- Along with these new features, lots of feature enhancements are done under-the-hood, at both

wait-notify pattern. If we put an element into an empty queue, all threads that were waiting in the take method wake up and try to receive the value. If we put an element into a full queue, the put method waits for the call to the get method. The get method removes an element and notifies the threads waiting in the put method that the queue has an empty place for a new item.

```
public class BlockingQueue<T> {
   private List<T> queue = new LinkedList<T>();
   private int limit = 10;
   public synchronized void put(T item) {
      while (queue.size() == limit) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      if (queue.isEmpty()) {
         notifyAll();
      }
      queue.add(item);
   }

   public synchronized T take() throws InterruptedException {
      while (queue.isEmpty()) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      if (queue.size() == limit) {
         notifyAll();
      }
      return queue.remove(0);
   }
}
```

What is a method reference?

A method reference is a Java 8 construct that can be used for referencing a method without invoking it. It is used for treating methods as Lambda Expressions. They only work as syntactic sugar to reduce the verbosity of some lambdas.

What is the meaning of String::valueOf expression?

- It is a static method reference to the valueOf method of the String class.

What is Optional? How can it be used?

- Optional is a new class in Java 8 that encapsulates an optional value i.e. a value that is either there or not. It is a wrapper around an object, and you can think of it as a container of zero or one element.
- Optional has a special Optional.empty() value instead of wrapped null. Thus it can be used instead of a

compiler and JVM level.

Describe some of the functional interfaces in the standard library.

There are a lot of functional interfaces in the java.util.function package, the more common ones include but not limited to:

- Function – it takes one argument and returns a result
- Consumer – it takes one argument and returns no result (represents a side effect)
- Supplier – it takes not argument and returns a result
- Predicate – it takes one argument and returns a boolean
- BiFunction – it takes two arguments and returns a result
- BinaryOperator – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types
- UnaryOperator – it is similar to a Function, taking a single argument and returning a result of the same type

What is a functional interface? What are the rules of defining a functional interface?

- A functional interface is an interface with no more, no less but one single abstract method (default methods do not count).
- Where an instance of such interface is required, a Lambda Expression can be used instead. More formally put: Functional interfaces provide target types for lambda expressions and method references.
- The arguments and return type of such expression directly match those of the single abstract method.

What is a Lambda Expression and what is it used for

- In very simple terms, a lambda expression is a function that can be referenced and passed around as an object.
- Lambda expressions introduce functional style processing in Java and facilitate the writing of compact and easy-to-read code.
- Because of this, lambda expressions are a natural replacement for anonymous classes as method arguments. One of their main uses is to define inline implementations of functional interfaces.

Explain the syntax and characteristics of a Lambda Expression

- A lambda expression consists of two parts: the parameter part and the expressions part separated by a forward arrow as below:
- params -> expressions

Any lambda expression has the following characteristics:

- Optional type declaration – when declaring the parameters on the left-hand side of the lambda, we don't need to declare their types as the compiler can

nullable value to get rid of NullPointerException in many cases.

- The main purpose of Optional, as designed by its creators, was to be a return type of methods that previously would return null.
- Such methods would require you to write boilerplate code to check the return value and sometimes could forget to do a defensive check.
- In Java 8, an Optional return type explicitly requires you to handle null or non-null wrapped values differently.
- For instance, the Stream.min() method calculates the minimum value in a stream of values. But what if the stream is empty? If it was not for Optional, the method would return null or throw an exception.
- But it returns an Optional value which may be Optional.empty() (the second case).

What is a default method and when do we use it?

- A default method is a method with an implementation – which can be found in an interface.
- We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface.
- Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method. In Java 8, this problem has been solved by the use of default method.
- For example, Collection interface does not have forEach method declaration. Thus, adding such method would simply break the whole collections API.
- Java 8 introduces default method so that Collection interface can have a default implementation of forEach method without requiring the classes implementing this interface to implement the same.

What is Nashorn in Java8?

- Nashorn is the new Javascript processing engine for the Java platform that shipped with Java 8. Until JDK 7, the Java platform used Mozilla Rhino for the same purpose. as a Javascript processing engine.
- Nashorn provides better compliance with the ECMA normalized JavaScript specification and better runtime performance than its predecessor.

What is jjs?

infer them from their values. So int param -> … and param ->… are all valid
- Optional parentheses – when only a single parameter is declared, we don't need to place it in parentheses. This means param -> … and (param) -> … are all valid. But when more than one parameter is declared, parentheses are required
- Optional curly braces – when the expressions part only has a single statement, there is no need for curly braces. This means that param – > statement and param – > {statement;} are all valid. But curly braces are required when there is more than one statement
- Optional return statement – when the expression returns a value and it is wrapped inside curly braces, then we don't need a return statement. That means (a, b) – > {return a+b;} and (a, b) – > {a+b;} are both valid.

What is the difference between intermediate and terminal operations?

- Stream operations are combined into pipelines to process streams. All operations are either intermediate or terminal.
- Intermediate operations are those operations that return Stream itself allowing for further operations on a stream.
- These operations are always lazy, i.e. they do not process the stream at the call site, an intermediate operation can only process data when there is a terminal operation. Some of the intermediate operations are filter, map and flatMap.
- Terminal operations terminate the pipeline and initiate stream processing. The stream is passed through all intermediate operations during terminal operation call. Terminal operations include forEach, reduce, Collect and sum.
- The intermediate operations are only triggered when a terminal operation exists.

What is stream pipelining in Java 8?

- Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories: intermediate operations and terminal operations.
- Each intermediate operation returns an instance of Stream itself when it runs, an arbitrary number of intermediate operations can, therefore, be set up to process data forming a processing pipeline.
- There must then be a terminal operation which returns a final value and terminates the pipeline.

Tell us about the new Date and Time API in Java 8

- A long-standing problem for Java developers has been the inadequate support for the date and time
- In Java 8, jjs is the new executable or command line tool used to execute Javascript code at the console.

What is a stream? How does it differ from a collection?

- In simple terms, a stream is an iterator whose role is to accept a set of actions to apply on each of the elements it contains.
- The stream represents a sequence of objects from a source such as a collection, which supports aggregate operations.
- They were designed to make collection processing simple and concise.
- Contrary to the collections, the logic of iteration is implemented inside the stream, so we can use methods like map and flatMap for performing a declarative processing.
- Another difference is that the Stream API is fluent and allows pipelining

```
int sum = Arrays.stream(new int[]{1, 2, 3})
  .filter(i -> i >= 2)
  .map(i -> i * 3)
  .sum();
```

- And yet another important distinction from collections is that streams are inherently lazily loaded and processed.

What is the difference between map and flatMap stream operation?

- There is a difference in signature between map and flatMap. Generally speaking, a map operation wraps its return value inside its ordinal type while flatMap does not.
- For example, in Optional, a map operation would return Optional<String> type while flatMap would return String type.
- So after mapping, one needs to unwrap (read "flatten") the object to retrieve the value whereas, after flat mapping, there is no such need as the object is already flattened. The same concept is applied to mapping and flat mapping in Stream.
- Both map and flatMap are intermediate stream operations that receive a function and apply this function to all elements of a stream.
- The difference is that for the map, this function returns a value, but for flatMap, this function returns a stream. The flatMap operation "flattens" the streams into one.
- Here's an example where we take a map of users' names and lists of phones and "flatten" it down to a list of phones of all the users using flatMap:

```
Map<String, List<String>> people = new HashMap<>();
people.put("John", Arrays.asList("555-1123",
"555-3389"));
```

manipulations required by ordinary developers.

- The existing classes such as java.util.Date and SimpleDateFormatter aren't thread-safe, leading to potential concurrency issues for users.
- Poor API design is also a reality in the old Java Data API. Here's just a quick example – years in java.util.Date start at 1900, months start at 1, and days start at 0 which is not very intuitive.
- These issues and several others have led to the popularity of third-party date and time libraries, such as Joda-Time.
- In order to address these problems and provide better support in JDK, a new date and time API, which is free of these problems, has been designed for Java SE 8 under the package java.time.

```
people.put("Mary", Arrays.asList("555-2243",
"555-5264"));
people.put("Steve", Arrays.asList("555-6654",
"555-3242"));

List<String> phones = people.values().stream()
 .flatMap(Collection::stream)
  .collect(Collectors.toList());
```

Note: Information gathered in this document has been collected from various sources on Internet.
Sources: www.baeldung.com, other sources (TBA)