21 Essential Java Interview Questions *

Describe and compare fail-fast and fail-safe iterators. Give examples.

View the answer →Hide answer

The main distinction between **fail-fast** and **fail-safe** iterators is whether or not the collection can be modified *while* it is being iterated. Fail-safe iterators allow this; fail-fast iterators do not.

- **Fail-fast** iterators operate directly on the collection itself. During iteration, fail-fast iterators fail as soon as they realize that the collection has been modified (i.e. upon realizing that a member has been added, modified, or removed) and will throw a `ConcurrentModificationException`. Some examples include `ArrayList`, `HashSet`, and `HashMap` (most JDK1.4 collections are implemented to be fail-fast).

- **Fail-safe** iterates operate on a *cloned copy* of the collection and therefore do *not* throw an exception if the collection is modified during iteration. Examples would include iterators returned by `ConcurrentHashMap` or `CopyOnWriteArrayList`.

`ArrayList`, `LinkedList`, and `Vector` are all implementations of the `List` interface. Which of them is most efficient for adding and removing elements from the list? Explain your answer, including any other alternatives you may be aware of.

View the answer →Hide answer

Of the three, `LinkedList` is generally going to give you the best performance. Here's why:

`ArrayList` and `Vector` each use an array to store the elements of the list. As a result, when an element is inserted into (or removed from) the middle of the list, the elements that follow must all be shifted accordingly. `Vector` is synchronized, so if a thread-safe implementation is *not* needed, it is recommended to use `ArrayList` rather than Vector.

`LinkedList`, on the other hand, is implemented using a doubly linked list. As a result, an inserting or removing an element only requires updating the links that immediately precede and follow the element being inserted or removed.

However, it is worth noting that if performance is that critical, it's better to just use an array and manage it yourself, or use one of the high performance 3rd party packages such as Trove or HPPC.

Why would it be more secure to store sensitive data (such as a password, social security number, etc.) in a character array rather than in a String?

View the answer →Hide answer

In Java, Strings are immutable and are stored in the String pool. What this means is that, once a String is created, it stays in the pool in memory until being garbage collected. Therefore, even after you're done processing the string value (e.g., the password), it remains available in memory for an indeterminate period of time thereafter (again, until being garbage collected) which you have no real control over. Therefore, anyone having access to a memory dump can potentially extract the sensitive data and exploit it.

In contrast, if you use a mutable object like a character array, for example, to store the value, you can set it to blank once you are done with it with confidence that it will no longer be retained in memory.

What is the `ThreadLocal` class? How and why would you use it?

View the answer →Hide answer

A single `ThreadLocal` instance can store different values for each thread independently. Each thread that accesses the `get()` or `set()` method of a `ThreadLocal` instance is accessing its own, independently initialized copy of the variable. `ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or transaction ID). The example below, from the `ThreadLocal` Javadoc, generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes `ThreadId.get()` and remains unchanged on subsequent calls.

```
public class ThreadId {
    // Next thread ID to be assigned
    private static final AtomicInteger nextId = new AtomicInteger(0);

    // Thread local variable containing each thread's ID
    private static final ThreadLocal<Integer> threadId =
        new ThreadLocal<Integer>() {
            @Override protected Integer initialValue() {
                return nextId.getAndIncrement();
        }
    };

    // Returns the current thread's unique ID, assigning it if necessary
    public static int get() {
        return threadId.get();
    }
}
```

Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

What is the `volatile` keyword? How and why would you use it?

View the answer →Hide answer

In Java, each thread has its own stack, including its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables into its own stack. The `volatile` keyword basically says to the JVM "Warning, this variable may be modified in another Thread".

In all versions of Java, the `volatile` keyword guarantees global ordering on reads and writes to a variable. This implies that every thread accessing a volatile field will read the variable's current value instead of (potentially) using a cached value.

In Java 5 or later, `volatile` reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.

Using `volatile` may be faster than a lock, but it will not work in some situations. The range of situations in which volatile is effective was expanded in Java 5; in particular, double-checked locking now works correctly.

The volatile keyword is also useful for 64-bit types like long and double since they are written in two operations. Without the volatile keyword you risk stale or invalid values.

One common example for using `volatile` is for a flag to terminate a thread. If you've started a thread, and you want to be able to safely interrupt it from a different thread, you can have the thread periodically check a flag (i.e., to stop it, set the flag to `true`). By making the flag volatile, you can ensure that the thread that is checking its value will see that it has been set to `true` without even having to use a synchronized block. For example:

```
public class Foo extends Thread {
    private volatile boolean close = false;
    public void run() {
        while(!close) {
            // do work
        }
    }
    public void close() {
        close = true;
        // interrupt here if needed
    }
}
```

Compare the `sleep()` and `wait()` methods in Java, including when and why you would use one vs. the other.

View the answer →Hide answer

`sleep()` is a blocking operation that keeps a hold on the monitor / lock of the shared object for the specified number of milliseconds.

`wait()`, on the other hand, simply *pauses* the thread until *either* (a) the specified number of milliseconds have elapsed *or* (b) it receives a desired notification from another thread (whichever is first), *without* keeping a hold on the monitor/lock of the shared object.

`sleep()` is most commonly used for polling, or to check for certain results, at a regular interval. `wait()` is generally used in multithreaded applications, in conjunction with `notify()` / `notifyAll()`, to achieve synchronization and avoid race conditions.

Tail recursion is functionally equivalent to iteration. Since Java does not yet support tail call optimization, describe how to transform a simple tail recursive function into a loop and why one is typically preferred over the other.

View the answer →Hide answer

Here is an example of a typical recursive function, computing the arithmetic series 1, 2, 3…N. Notice how the addition is performed after the function call. For each recursive step, we add another frame to the stack.

```
public int sumFromOneToN(int n) {
  if (n < 1) {
    return 0;
  }

  return n + sumFromOneToN(n - 1);
}
```

Tail recursion occurs when the recursive call is in the tail position within its enclosing context - after the function calls itself, it performs no additional work. That is, once the base case is complete, the solution is apparent. For example:

```
public int sumFromOneToN(int n, int a) {
  if (n < 1) {
    return a;
  }

  return sumFromOneToN(n - 1, a + n);
}
```

Here you can see that `a` plays the role of the accumulator - instead of computing the sum on the way down the stack, we compute it on the way up, effectively making the return trip unnecessary, since it stores no additional state and performs no further computation. Once we hit the base case, the work is done - below is that same function, “unrolled”.

```
public int sumFromOneToN(int n) {
  int a = 0;

  while(n > 0) {
    a += n--;
  }

  return a;
}
```

Many functional languages natively support tail call optimization, however the JVM does not. In order to implement recursive functions in Java, we need to be aware of this limitation to avoid `StackOverflowErrors`. In Java, iteration is almost universally preferred to recursion.

How can you catch an exception thrown by another thread in Java?

View the answer →Hide answer

This can be done using Thread.UncaughtExceptionHandler.

Here's a simple example:

```
// create our uncaught exception handler
Thread.UncaughtExceptionHandler handler = new Thread.UncaughtExceptionHandler() {
    public void uncaughtException(Thread th, Throwable ex) {
        System.out.println("Uncaught exception: " + ex);
    }
};

// create another thread
Thread otherThread = new Thread() {
    public void run() {
        System.out.println("Sleeping ...");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted.");
        }
        System.out.println("Throwing exception ...");
        throw new RuntimeException();
    }
};

// set our uncaught exception handler as the one to be used when the new thread
// throws an uncaught exception
otherThread.setUncaughtExceptionHandler(handler);

// start the other thread - our uncaught exception handler will be invoked when
// the other thread throws an uncaught exception
otherThread.start();
```

What is the Java Classloader? List and explain the purpose of the three types of class loaders.

View the answer →Hide answer

The Java Classloader is the part of the Java runtime environment that loads classes on demand (lazy loading) into the JVM (Java Virtual Machine). Classes may be loaded from the local file system, a remote file system, or even the web.

When the JVM is started, three class loaders are used: 1. **Bootstrap Classloader:** Loads core java API file rt.jar from folder. 2. **Extension Classloader:** Loads jar files from folder. 3. **System/Application Classloader:** Loads jar files from path specified in the CLASSPATH environment variable.

Is a `finally` block executed when an exception is thrown from a `try` block that does not have a `catch` block, and if so, when?

View the answer →Hide answer

A `finally` block is executed even if an exception is thrown or propagated to the calling code block.

Example:

```
public class FinallyExecution {
        public static void main(String[] args) {
                try{
                        FinallyExecution.divide(100, 0);}
                finally{
                        System.out.println("finally in main");
                }
        }
        public static void divide(int n, int div){
                try{
                        int ans = n/div; }
                finally{
                        System.out.println("finally of divide");
                }
        }
}
```

Output can vary, being either:

```
finally of divide
finally in main
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptions.FinallyExecution.divide(FinallyExecution.java:20)
        at exceptions.FinallyExecution.main(FinallyExecution.java:9)
```

…or…

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptions.FinallyExecution.divide(FinallyExecution.java:20)
        at exceptions.FinallyExecution.main(FinallyExecution.java:9)
finally of divide
finally in main
```

When designing an abstract class, why should you avoid calling abstract methods inside its constructor?

View the answer →Hide answer

This is a problem of initialization order. The subclass constructor will not have had a chance to run yet and there is no way to force it to run it before the parent class. Consider the following example class:

```
        public abstract class Widget {
                private final int cachedWidth;
                private final int cachedHeight;

                public Widget() {
                    this.cachedWidth = width();
                    this.cachedHeight = height();
                }
```

```
        protected abstract int width();
        protected abstract int height();
    }
```

This seems like a good start for an abstract Widget: it allows subclasses to fill in `width` and `height`, and caches their initial values. However, look when you spec out a typical subclass implementation like so:

```
public class SquareWidget extends Widget {
        private final int size;

        public SquareWidget(int size) {
            this.size = size;
        }

        @Override
        protected int width() {
            return size;
        }

        @Override
        protected int height() {
            return size;
        }
    }
```

Now we've introduced a subtle bug: `Widget.cachedWidth` and `Widget.cachedHeight` will always be zero for `SquareWidget` instances! This is because the `this.size = size` assignment occurs *after* the `Widget` constructor runs.

Avoid calling abstract methods in your abstract classes' constructors, as it restricts how those abstract methods can be implemented.

What variance is imposed on generic type parameters? How much control does Java give you over this?

View the answer →Hide answer

Java's generic type parameters are *invariant*. This means for any distinct types `A` and `B`, `G<A>` is not a subtype or supertype of `G<B>`. As a real world example, `List<String>` is not a supertype or subtype of `List<Object>`. So even though `String` extends (i.e. is a subtype of) `Object`, both of the following assignments will fail to compile:

```
List<String> strings = Arrays.<Object>asList("hi there");
List<Object> objects = Arrays.<String>asList("hi there");
```

Java does give you some control over this in the form of *use-site variance*. On individual methods, we can use `? extends Type` to create a *covariant* parameter. Here's an example:

```
public double sum(List<? extends Number> numbers) {
    double sum = 0;
    for (Number number : numbers) {
        sum += number.doubleValue();
    }
    return sum;
}


List<Long> longs = Arrays.asList(42L, 128L, -10L);
double sumOfLongs = sum(longs);
```

Even though `longs` is a `List<Long>` and not `List<Number>`, it can be passed to `sum`.

Similarly, `? super Type` lets a method parameter be *contravariant*. Consider a function with a callback parameter:

```
public void forEachNumber(Callback<? super Number> callback) {
    callback.call(50.0f);
    callback.call(123123);
    callback.call((short) 99);
}
```

`forEachNumber` allows `Callback<Object>` to be a subtype of `Callback <Number>`, which means any callback that handles a supertype of `Number` will do:

```
forEachNumber(new Callback<Object>() {
    @Override public void call(Object value) {
        System.out.println(value);
```

```
    }
});
```

Note, however, that attempting to provide a callback that handles only `Long` (a subtype of `Number`) will rightly fail:

```
// fails to compile!
forEachNumber(new Callback<Long>() { ... });
```

Liberal application of use-site variance can prevent many of the unsafe casts that often appear in Java code and is crucial when designing interfaces used by multiple developers.

If one needs a `Set`, how do you choose between `HashSet` vs. `TreeSet`?

View the answer →Hide answer

At first glance, `HashSet` is superior in almost every way: O(1) `add`, `remove` and `contains`, vs. O(log(N)) for `TreeSet`.

However, `TreeSet` is indispensable when you wish to maintain order over the inserted elements or query for a range of elements within the set.

Consider a `Set` of timestamped `Event` objects. They could be stored in a `HashSet`, with `equals` and `hashCode` based on that timestamp. This is efficient storage and permits looking up events by a specific timestamp, but how would you get all events that happened on any given day? That would require a O(n) traversal of the `HashSet`, but it's only a O(log(n)) operation with `TreeSet` using the `tailSet` method:

```
public class Event implements Comparable<Event> {
    private final long timestamp;

    public Event(long timestamp) {
        this.timestamp = timestamp;
    }

    @Override public int compareTo(Event that) {
        return Long.compare(this.timestamp, that.timestamp);
    }
}

...

SortedSet<Event> events = new TreeSet<>();
events.addAll(...); // events come in

// all events that happened today
long midnightToday = ...;
events.tailSet(new Event(midnightToday));
```

If `Event` happens to be a class that we cannot extend or that doesn't implement `Comparable`, `TreeSet` allows us to pass in our own `Comparator`:

```
SortedSet<Event> events = new TreeSet<>(
        (left, right) -> Long.compare(left.timestamp, right.timestamp));
```

Generally speaking, `TreeSet` is a good choice when order matters and when reads are balanced against the increased cost of writes.

What are method references, and how are they useful?

View the answer →Hide answer

Method references were introduced in Java 8 and allow constructors and methods (static or otherwise) to be used as lambdas. They allow one to discard the boilerplate of a lambda when the method reference matches an expected signature.

For example, suppose we have a service that must be stopped by a shutdown hook. Before Java 8, we would have code like this:

```
final SomeBusyService service = new SomeBusyService();
service.start();

onShutdown(new Runnable() {
    @Override
    public void run() {
        service.stop();
    }
});
```

With lambdas, this can be cut down considerably:

```
onShutdown(() -> service.stop());
```

However, `stop` matches the signature of `Runnable.run` (`void` return type, no parameters), and so we can introduce a method reference to the `stop` method of that specific `SomeBusyService` instance:

```
onShutdown(service::stop);
```

This is terse (as opposed to verbose code) and clearly communicates what is going on.

Method references don't need to be tied to a specific instance, either; one can also use a method reference to an arbitrary object, which is useful in `Stream` operations. For example, suppose we have a `Person` class and want just the lowercase names of a collection of people:

```
List<Person> people = ...

List<String> names = people.stream()
        .map(Person::getName)
        .map(String::toLowerCase)
        .collect(toList());
```

A complex lambda can also be pushed into a static or instance method and then used via a method reference instead. This makes the code more reusable and testable than if it were "trapped" in the lambda.

So we can see that method references are mainly used to improve code organization, clarity and terseness.

How are Java enums more powerful than integer constants? How can this capability be used?

View the answer →Hide answer

Enums are essentially final classes with a fixed number of instances. They can implement interfaces but cannot extend another class.

This flexibility is useful in implementing the strategy pattern, for example, when the number of strategies is fixed. Consider an address book that records multiple methods of contact. We can represent these methods as an enum and attach fields, like the filename of the icon to display in the UI, and any corresponding behaviour, like how to initiate contact via that method:

```
public enum ContactMethod {
    PHONE("telephone.png") {
        @Override public void initiate(User user) {
            Telephone.dial(user.getPhoneNumber());
        }
    },
    EMAIL("envelope.png") {
        @Override public void initiate(User user) {
            EmailClient.sendTo(user.getEmailAddress());
        }
    },
    SKYPE("skype.png") {
        ...
    };

    ContactMethod(String icon) {
        this.icon = icon;
    }

    private final String icon;

    public abstract void initiate(User user);
```

```
        public String getIcon() {
            return icon;
        }
    }
```

We can dispense with `switch` statements entirely by simply using instances of `ContactMethod`:

```
    ContactMethod method = user.getPrimaryContactMethod();
    displayIcon(method.getIcon());
    method.initiate(user);
```

This is just the beginning of what can be done with enums. Generally, the safety and flexibility of enums means they should be used in place of integer constants, and switch statements can be eliminated with liberal use of abstract methods.

What does it mean for a collection to be “backed by” another? Give an example of when this property is useful.

View the answer →Hide answer

If a collection backs another, it means that changes in one are reflected in the other and vice-versa.

For example, suppose we wanted to create a `whitelist` function that removes invalid keys from a `Map`. This is made far easier with `Map.keySet`, which returns a set of keys that is backed by the original map. When we remove keys from the key set, they are also removed from the backing map:

```
    public static <K, V> Map<K, V> whitelist(Map<K, V> map, K... allowedKeys) {
        Map<K, V> copy = new HashMap<>(map);
        copy.keySet().retainAll(asList(allowedKeys));
        return copy;
    }
```

`retainAll` writes through to the backing map, and allows us to easily implement something that would otherwise require iterating over the entries in the input map, comparing them against `allowedKey`, etcetera.

Note, it is important to consult the documentation of the backing collection to see which modifications will successfully write through. In the example above, `map.keySet().add(value)` would fail, because we cannot add a key to the backing map without a value.

What is reflection? Give an example of functionality that can only be implemented using reflection.

View the answer →Hide answer

Reflection allows programmatic access to information about a Java program’s types. Commonly used information includes: methods and fields available on a class, interfaces implemented by a class, and the runtime-retained annotations on classes, fields and methods.

Examples given are likely to include:

- Annotation-based serialization libraries often map class fields to JSON keys or XML elements (using annotations). These libraries need reflection to inspect those fields and their annotations and also to access the values during serialization.
- Model-View-Controller frameworks call controller methods based on routing rules. These frameworks must use reflection to find a method corresponding to an action name, check that its signature conforms to what the framework expects (e.g. takes a `Request` object, returns a `Response`), and finally, invoke the method.
- Dependency injection frameworks lean heavily on reflection. They use it to instantiate arbitrary beans for injection, check fields for annotations such as `@In` discover if they require injection of a bean, and also to set those values.
- Object-relational mappers such as Hibernate use reflection to map database columns to fields or getter/setter pairs of a class, and can go as far as to infer tab column names by reading class and getter names, respectively.

A concrete code example could be something simple, like copying an object's fields into a map:

```
Person person = new Person("Doug", "Sparling", 31);

Map<String, Object> values = new HashMap<>();
for (Field field : person.getClass().getDeclaredFields()) {
    values.put(field.getName(), field.get(person));
}

// prints {firstName=Doug, lastName=Sparling, age=31}
System.out.println(values);
```

Such tricks can be useful for debugging, or for utility methods such as a `toString` method that works on any class.

Aside from implementing generic libraries, direct use of reflection is rare but it is still a handy tool to have. Knowledge of reflection is also useful for when these mechanisms fail.

However, it is often prudent to avoid reflection unless it is strictly necessary, as it can turn straightforward compiler errors into runtime errors.

What are static initializers and when would you use them?

View the answer →Hide answer

A static initializer gives you the opportunity to run code during the initial loading of a class and it guarantees that this code will only run once and will finish running before your class can be accessed in any way.

They are useful for performing initialization of complex static objects or to register a type with a static registry, as JDBC drivers do.

Suppose you want to create a static, immutable `Map` containing some feature flags. Java doesn't have a good one-liner for initializing maps, so you can use static initializers instead:

```
public static final Map<String, Boolean> FEATURE_FLAGS;
static {
    Map<String, Boolean> flags = new HashMap<>();
    flags.put("frustrate-users", false);
    flags.put("reticulate-splines", true);
    flags.put(...);
    FEATURE_FLAGS = Collections.unmodifiableMap(flags);
}
```

Within the same class, you can repeat this pattern of declaring a static field and immediately initializing it, since multiple static initializers are allowed.

Nested classes can be static or non-static (also called an inner class). How do you decide which to use? Does it matter?

View the answer →Hide answer

The key difference between is that inner classes have full access to the fields and methods of the enclosing class. This can be convenient for event handlers, but comes at a cost: every instance of an inner class retains and requires a reference to its enclosing class.

With this cost in mind, there are many situations where we should prefer static nested classes. When instances of the nested class will outlive instances of the enclosing class, the nested class should be static to prevent memory leaks. Consider this implementation of the factory pattern:

```
public interface WidgetParser {
    Widget parse(String str);
```

```
}

public class WidgetParserFactory {
    public WidgetParserFactory(ParseConfig config) {
        ...
    }

    public WidgetParser create() {
        new WidgetParserImpl(...);
    }

    private class WidgetParserImpl implements WidgetParser {
        ...

        @Override public Widget parse(String str) {
            ...
        }
    }
}
```

At a glance, this design looks good: the `WidgetParserFactory` hides the implementation details of the parser with the nested class `WidgetParserImpl`. However, `WidgetParserImpl` is not static, and so if `WidgetParserFactory` is discarded immediately after the `WidgetParser` is created, the factory will leak, along with all the references it holds.

`WidgetParserImpl` should be made static, and if it needs access to any of `WidgetParserFactory`'s internals, they should be passed into `WidgetParserImpl`'s constructor instead. This also makes it easier to extract `WidgetParserImpl` into a separate class should it outgrow its enclosing class.

Inner classes are also harder to construct via reflection due to their "hidden" reference to the enclosing class, and this reference can get sucked in during reflection-based serialization, which is probably not intended.

So we can see that the decision of whether to make a nested class static is important, and that one should aim to make nested classes static in cases where instances will "escape" the enclosing class or if reflection on those nested classes is involved.

What is the difference between `String s = "Test"` and `String s = new String("Test")`? Which is better and why?

View the answer →Hide answer

In general, `String s = "Test"` is more efficient to use than `String s = new String("Test")`.

In the case of `String s = "Test"`, a String with the value "Test" will be created in the String pool. If another String with the same value is then created (e.g., `String s2 = "Test"`), it will reference this same object in the String pool.

However, if you use `String s = new String("Test")`, in addition to creating a String with the value "Test" in the String pool, that String object will then be passed to the constructor of the String Object (i.e., `new String("Test")`) and will create another String object (not in the String pool) with that value. Each such call will therefore create an additional String object (e.g., `String s2 = new String("Test")` would create an addition String object, rather than just reusing the same String object from the String pool).

How can you swap the values of two numeric variables **without** using any other variables?

View the answer →Hide answer

You can swap two values `a` and `b` without using any other variables as follows:

```
a = a + b;
b = a - b;
a = a - b;
```

* There is more to interviewing than tricky technical questions, so these are intended merely as a guide. Not every “A” candidate worth hiring will be able to answer them all, nor does answering them all guarantee an “A” candidate. At the end of the day, hiring remains an art, a science — and a lot of work.

Submit an interview question

Submitted questions and answers are subject to review and editing, and may or may not be selected for posting, at the sole discretion of Toptal, LLC.

Name

Email

Enter your question here

Enter your answer here

All fields are required

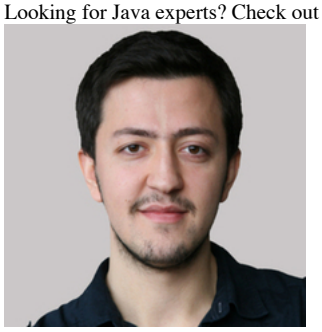
I agree with the Terms and Conditions of Toptal, LLC's Privacy Policy

Submit a Question

Thanks for submitting your question.

Our editorial staff will review it shortly. Please note that submitted questions and answers are subject to review and editing, and may or may not be selected for posting, at the sole discretion of Toptal, LLC.

Looking for Java experts? Check out Toptal’s Java developers.



View full profile »

Furkan Varol

Germany

Furkan is a software engineer with a focus on back-end programming. With over five years of professional development experience, he is known for producing clean code very quickly, has a passion for learning, and enjoys solving algorithmic problems. He is dedicated and tenacious with his work and will be a great addition to any team.

JavaApache SparkSpring

Hire Furkan

View full profile »

Christian Diego De Martino

Argentina

Christian is a software engineer with over 14 years of experience developing applications with Java as well as extremely high insight on the iOS languages such us Objective-C and Swift since the early betas. Over the last few years, he's being coding in JavaScript and widely used frameworks and libraries like React and Node. Christian has a strong command of English and is able to communicate extremely well.

JavaObjective-CUI KitGitGitHub+6 more

Hire Christian

View full profile »

Christopher Smith

United States

Christopher is an expert JVM developer with solid experience building with Spring, Groovy, and the JVM ecosystem in general. He has managed the full softwar lifecycle from requirements to deployment, along with systems administration and networking (CCDP/CCNP). He is a proactive guy who's looking for projects th involve back-end engineering, infrastructure questions, networking, orchestration, and harder functional programming.

JavaSpringSpring BootLinux
Hire Christopher
Toptal connects the top 3% of freelance talent all over the world.

Join the Toptal community.

Hire a developer
or
Apply as a developer

Highest In-Demand Talent

- iOS Developers
- Front-End Developers
- UX Designers
- UI Designers
- Financial Modeling Consultants
- Interim CFOs
- Digital Project Managers

About

- Top 3%
- Clients
- Freelance Developers
- Freelance Designers
- Freelance Finance Experts
- Freelance Project Managers
- Freelance Product Managers
- About Us

Contact

- Contact Us
- Press Center
- Careers
- FAQ

Social

- Facebook
- Twitter
- Instagram
- LinkedIn

Hire the top 3% of freelance talent™

- Privacy Policy
- Website Terms