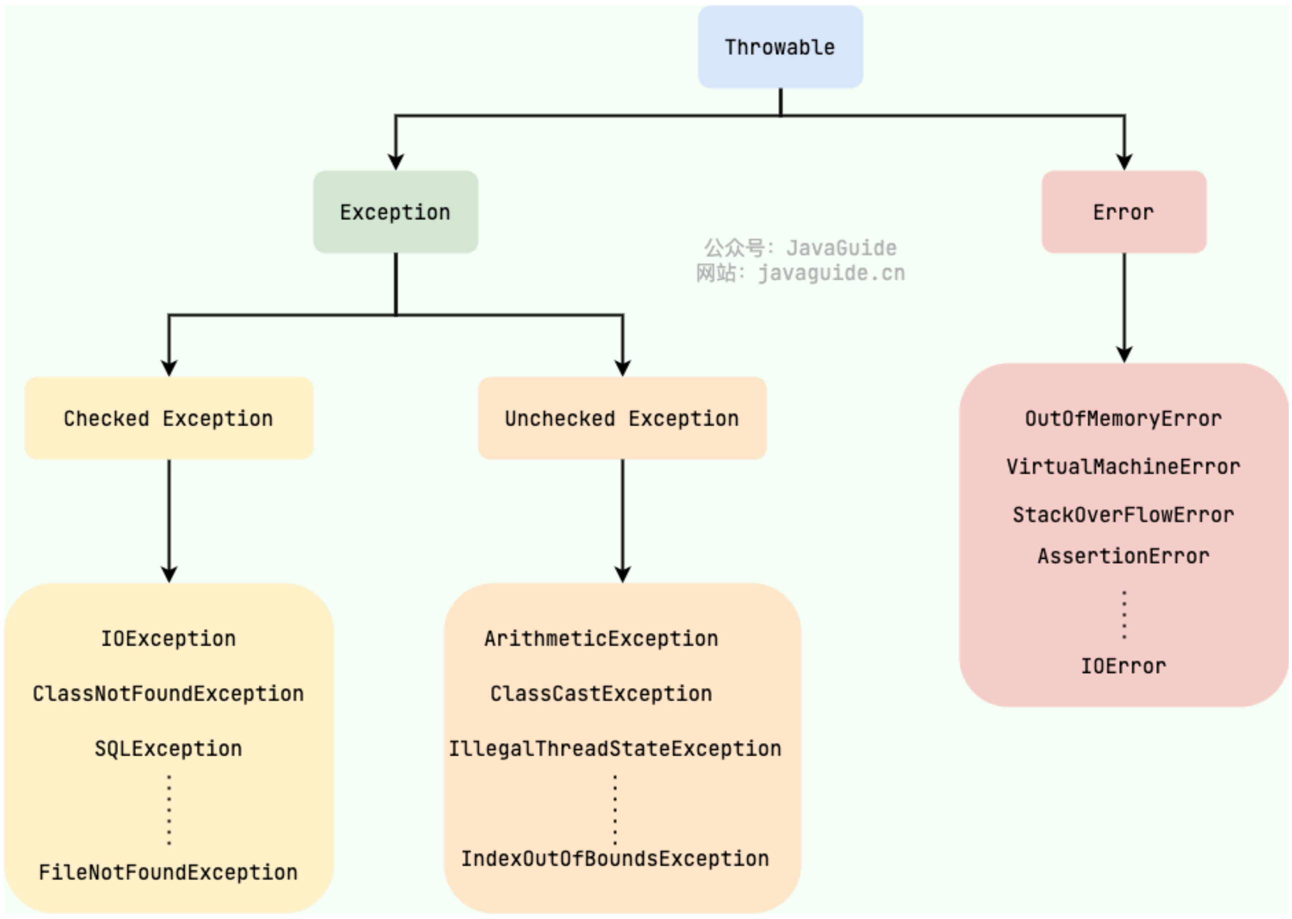
Summary of Common Java Basics Interview Questions (Part 2)

Guide Java Java Basics About 7019 words About 23 minutes

abnormal

Overview of Java exception class hierarchy diagram :



What is the difference between Exception and Error?

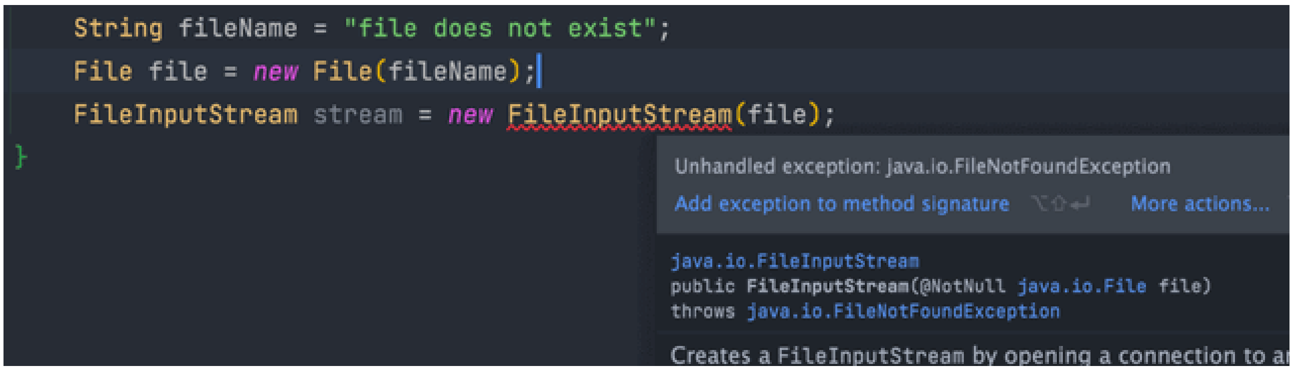
In Java, all exceptions have a common ancestor class `java.lang` in the package `Throwable` . `Throwable` The class has two important subclasses:

- **`Exception`** : Exceptions that the program itself can handle can be `catch` caught by . `Exception` They can be divided into Checked Exception (checked exceptions, must be handled) and Unchecked Exception (unchecked exceptions, can be left unhandled).
- **`Error`** `Error` Exceptions are errors that the program cannot handle. We ~~cannot~~ ~~`catch` catch them using Exception ,~~ and Exception catching is not recommended `catch` . Examples include Java Virtual Machine runtime errors (`Virtual MachineError`), insufficient virtual machine memory errors (`OutOfMemoryError`), and class definition errors (`NoClassDefFoundError`). When these exceptions occur, the Java Virtual Machine (JVM) typically terminates the thread.

⭐What is the difference between Checked Exception and Unchecked Exception?

Checked Exception is a checked exception. During the compilation process of Java code, if the checked exception is not handled by the `catch` or `throws` keyword, it will not pass the compilation.

For example, the following IO operation code:



```
    String fileName = "file does not exist";
    File file = new File(fileName);
    FileInputStream stream = new FileInputStream(file);
}
```

```
Unhandled exception: java.io.FileNotFoundException
Add exception to method signature ⌥⇧↵     More actions...
java.io.FileInputStream
public FileInputStream(@NotNull java.io.File file)
throws java.io.FileNotFoundException
Creates a FileInputStream by opening a connection to a
```

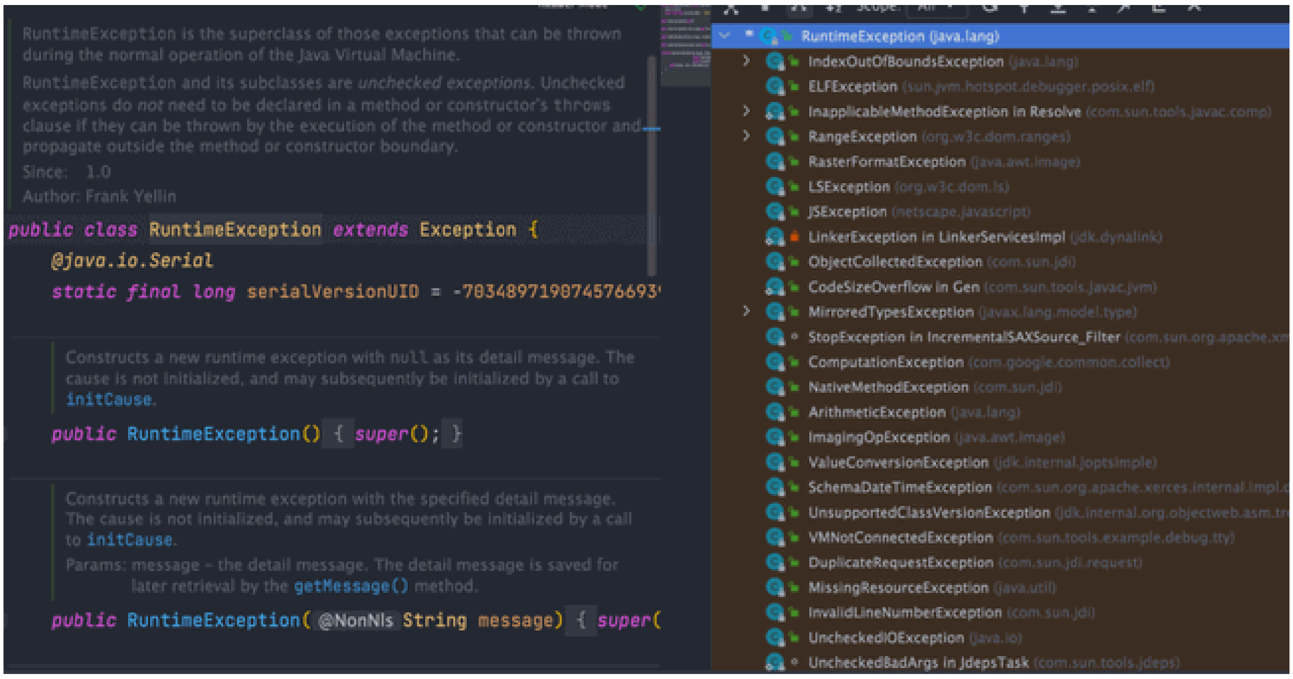
Except for `RuntimeException` Exception and its subclasses, all other `Exception` classes and their subclasses are checked exceptions. Common checked exceptions include: IO-related exceptions, Exception `ClassNotFoundException` , `SQLException` ...

Unchecked Exception means **unchecked exception** . During the compilation process of Java code, we can pass the compilation normally even if we do not handle unchecked exceptions.

`RuntimeException` And its subclasses are collectively referred to as unchecked exceptions. Common ones are (it is recommended to write them down, they will be often used in daily development):

- `NullPointerException` (Null pointer error)
- `IllegalArgumentException` (Parameter error, such as method input parameter type error)
- `NumberFormatException` (String to number conversion error, `IllegalArgumentException` subclass)
- `ArrayIndexOutOfBoundsException` (Array out of bounds error)
- `ClassCastException` (Type conversion error)
- `ArithmeticException` (Arithmetic error)
- `SecurityException` (Security errors such as insufficient permissions)

- `UnsupportedOperationException` (Unsupported operation errors such as creating the same user repeatedly)
- ...



```
RuntimeException is the superclass of those exceptions that can be thrown
during the normal operation of the Java Virtual Machine.
RuntimeException and its subclasses are unchecked exceptions. Unchecked
exceptions do not need to be declared in a method or constructor's throws
clause if they can be thrown by the execution of the method or constructor and
propagate outside the method or constructor boundary.
Since:   1.0
Author: Frank Yellin
public class RuntimeException extends Exception {
    @java.io.Serial
    static final long serialVersionUID = -7034897190745766939

        Constructs a new runtime exception with null as its detail message. The
        cause is not initialized, and may subsequently be initialized by a call to
        initCause.
    public RuntimeException() { super(); }

        Constructs a new runtime exception with the specified detail message.
        The cause is not initialized, and may subsequently be initialized by a call
        to initCause.
        Params: message - the detail message. The detail message is saved for
                later retrieval by the getMessage() method.
    public RuntimeException( @NonNls String message) { super(
```

```
RuntimeException (java.lang)
  IndexOutOfBoundsException (java.lang)
  ELFException (sun.jvm.hotspot.debugger.posix.elf)
  InapplicableMethodException in Resolve (com.sun.tools.javac.comp)
  RangeException (org.w3c.dom.ranges)
  RasterFormatException (java.awt.image)
  LSException (org.w3c.dom.ls)
  JSException (netscape.javascript)
  LinkerException in LinkerServicesImpl (jdk.dynalink)
  ObjectCollectedException (com.sun.jdi)
  CodeSizeOverflow in Gen (com.sun.tools.javac.jvm)
  MirroredTypesException (javax.lang.model.type)
  StopException in IncrementalSAXSource_Filter (com.sun.org.apache.xm
  ComputationException (com.google.common.collect)
  NativeMethodException (com.sun.jdi)
  ArithmeticException (java.lang)
  ImagingOpException (java.awt.image)
  ValueConversionException (jdk.internal.joptsimple)
  SchemaDateTimeException (com.sun.org.apache.xerces.internal.impl.d
  UnsupportedClassVersionException (jdk.internal.org.objectweb.asm.tr
  VMNotConnectedException (com.sun.tools.example.debug.tty)
  DuplicateRequestException (com.sun.jdi.request)
  MissingResourceException (java.util)
  InvalidLineNumberException (com.sun.jdi)
  UncheckedIOException (java.io)
  UncheckedBadArgs in JdepsTask (com.sun.tools.jdeps)
```

What are the common methods of Throwable class?

- `String getMessage()` : Returns detailed information when an exception occurs
- `String toString()` : Returns a brief description of the exception that occurred
- `String getLocalizedMessage()` : Returns the localized information of the exception object. Use `Throwable` a subclass that overrides this method to generate localized information. If the subclass does not override this method, the information returned by this method `getMessage()` is the same as the result returned
- `void printStackTrace()` `Throwable` : Print the exception information encapsulated by the object on the console

How to use try-catch-finally?

- `try` Block: Used to catch exceptions. It can be followed by zero or more `catch` blocks. If there is no `catch` block, it must be followed by a `finally` block.
- `catch` Block: used to handle exceptions caught by try.
- `finally` `finally` Block: The statements in the block are executed regardless of whether the exception is caught or handled . When a statement is encountered in block or `catch` a block `return` , `finally` the block is executed before the method returns.

Code example:

```java
try {
    System.out.println("Try to do something");
    throw new RuntimeException("RuntimeException");
} catch (Exception e) {
    System.out.println("Catch Exception -> " + e.getMessage());
} finally {
    System.out.println("Finally");
}
```

Output:

```plain
Try to do something
Catch Exception -> RuntimeException
Finally
```

Note: Do not use return in a finally block! When both a try statement and a finally block contain return statements, the return statement in the try block is ignored. This is because the return value in the try statement is first temporarily stored in a local variable. When the return in the finally block is executed, the value of the local variable becomes the return value in the finally block.

Code example:

```java
public static void main(String[] args) {
    System.out.println(f(2));
}

public static int f(int value) {
    try {
        return value * value;
    } finally {
        if (value == 2) {
            return 0;
        }
    }
}
```

Output:

```plain
0
```

Will the code in finally be executed?

Not necessarily! In some cases, the code in finally will not be executed.

For example, if the virtual machine is terminated before finally, the code in finally will not be executed.

```java
try {
    System.out.println("Try to do something");
    throw new RuntimeException("RuntimeException");
} catch (Exception e) {
    System.out.println("Catch Exception -> " + e.getMessage());
    //                Java
    System.exit(1);
} finally {
    System.out.println("Finally");
}
```

Output:

```plain
Try to do something
Catch Exception -> RuntimeException
```

`finally` In addition, the block code will not be executed in the following two special cases :

1. The thread where the program is located dies.
2. Turn off the CPU.

Related issue: https://github.com/Snailclimb/JavaGuide/issues/190 .

`try catch finally` Let's go a step further: analyze the implementation principle behind this syntactic sugar from the bytecode perspective .

How to use `try-with-resources` instead `try-catch-finally` ?

1. **Scope (resource definition):** Any object that implements `java.lang.AutoCloseable` or `java.io.Closeable`
2. **Closing resources and finally blocks are executed in the order that they are closed:** In `try-with-resources` a statement, any catch or finally block is executed after the declared resources are closed.

Effective Java clearly states:

> When faced with resources that must be closed, we should always prefer using `try-with-resources` instead of `try-finally` . The resulting code is shorter and clearer, and the exceptions it generates are more useful to us. `try-with-resources` The statement makes it easier to write code that must close resources, which would be `try-finally` almost impossible otherwise.

In Java, resources like `InputStream` , `OutputStream` , `Scanner` , `PrintWriter` etc. require us to call `close()` methods to manually close them. Generally, we use `try-catch-finally` statements to achieve this requirement, as follows:

```java
//
Scanner scanner = null;
try {
    scanner = new Scanner(new File("D://read.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

Use the statement after Java 7 `try-with-resources` to transform the above code:

```java
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

Of course, when multiple resources need to be closed, `try-with-resources` it is very simple to implement using . However, if you still use it, `try-catch-finally` it may cause many problems.

`try-with-resources` Multiple resources can be declared in a block by separating them with semicolons .

```java
try (BufferedInputStream bin = new BufferedInputStream(new FileInputStream(new File("test.txt")));
     BufferedOutputStream bout = new BufferedOutputStream(new FileOutputStream(new File("out.txt")))) {
    int b;
    while ((b = bin.read()) != -1) {
        bout.write(b);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

⭐What should I pay attention to when using exceptions?

- Do not define exceptions as static variables, because this will cause the exception stack information to be messed up. Every time an exception is thrown manually, we need to manually create a new exception object to throw.
- The exception information thrown must be meaningful.
- It is recommended to throw more specific exceptions, such as when a string is converted to a number with an incorrect format, it should be thrown `NumberFormatException` instead of its parent class `IllegalArgumentException` .
- Avoid duplicate logging: If sufficient information (including exception type, error message, and stack trace) has been logged where the exception is caught, the same error message should not be logged again when the exception is thrown again in the business

code. Duplicate logging will bloat the log file and may obscure the actual cause of the problem, making it more difficult to track and resolve.

- ...

Generics

What are generics? What are they used for?

Java Generics is a new feature introduced in JDK 5. Using generic parameters can enhance the readability and stability of code.

The compiler can detect generic parameters and specify the type of object passed in through generic parameters. For example, `ArrayList<Person> persons = new ArrayList<Person>()` this line of code specifies that `ArrayList` only `Person` objects of type _object can be passed in. If objects of other types are passed in, an error will be reported.

```java
ArrayList<E> extends AbstractList<E>
```

In addition, the native `List` return type `Object` requires manual type conversion before use, while the compiler automatically converts after using generics.

What are the ways to use generics?

There are generally three ways to use generics: **generic classes** , **generic interfaces** , and **generic methods** .

1. Generic class :

```java
//    T                           T  E  K  V
//                           T
public class Generic<T>{

    private T key;

    public Generic(T key) {
        this.key = key;
    }


    public T getKey(){
        return key;
    }
}
```

How to instantiate a generic class:

```java
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

2. Generic interface :

```java
public interface Generator<T> {
    public T method();
}
```

Implement a generic interface without specifying a type:

```java
class GeneratorImpl<T> implements Generator<T>{
    @Override
    public T method() {
        return null;
    }
}
```

Implement the generic interface and specify the type:

```java
class GeneratorImpl implements Generator<String> {
    @Override
    public String method() {
        return "hello";
    }
}
```

3. Generic methods :

```java
   public static < E > void printArray( E[] inputArray )
   {
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
```

```
        System.out.println();
    }
```

use:

```java
//                Integer, Double    Character
Integer[] intArray = { 1, 2, 3 };
String[] stringArray = { "Hello", "World" };
printArray( intArray  );
printArray( stringArray  );
```

> Note: `public static < E > void printArray( E[] inputArray )` Generally called static generic method; in Java, generic is just a placeholder and can only be used after the type is passed. The type parameter can only be passed when the class is instantiated. Since the static method is loaded before the class is instantiated, that is, the generic in the class has not yet passed the real type parameter, the static method has been loaded. Therefore, the static generic method cannot use the generic declared on the class. You can only use the generics declared by yourself. `<E>`

Where are generics used in the project?

- Custom interface general return result `CommonResult<T>` Through parameters, `T` the data type of the result can be dynamically specified according to the specific return type
- Define `Excel` a processing class `ExcelUtil<T>` to dynamically specify `Excel` the exported data type
- Build a collection tool class (see the method `Collections` in). `sort` `binarySearch`
- ...

⭐Reflection

For a detailed explanation of reflection, please see this article Detailed Explanation of Java Reflection Mechanism .

What is Reflection?

Simply put, Java reflection is **the ability to dynamically obtain class informati and operate classes or objects (methods, properties) while the program is running** .

Typically, the code we write has its type determined at compile time, making it clear which method to call and which field to access. However, reflection allows us to discover a class's methods, properties, and constructors at runtime **. We can even dynamically create objects, call methods, or modify properties, even if those methods or properties are private.**

It is this ability to "look back" and perform operations at runtime that makes reflection **the cornerstone of many common frameworks and libraries** . It makes the code more flexible and can handle types that are unknown at compile time.

What are the advantages and disadvantages of reflection?

advantage:

1. **Flexibility and dynamism** : Reflection allows programs to dynamically load classes, create objects, call methods, and access fields at runtime. This allows the program's behavior to be dynamically adapted and extended based on actual needs (such as configuration files, user input, and annotations), significantly improving the system's flexibility and adaptability.
2. **The foundation of framework development** : Many modern Java frameworks (such as Spring, Hibernate, and MyBatis) make extensive use of reflection to implement core features such as dependency injection (DI), aspect-oriented programming (AOP), object-relational mapping (ORM), and annotation processing. Reflection is an indispensable foundational tool for implementing these "magic" features.
3. **Decoupling and generality** : Reflection allows you to write more general, reusable, and highly decoupled code, reducing dependencies between modules. For example, reflection allows you to implement general object copying, serialization, and Bean tools.

shortcoming:

1. **Performance overhead** : Reflection operations are generally slower than direct code calls due to factors such as dynamic type resolution, method lookup, and limited JIT compiler optimizations. However, for most framework scenarios, this performance loss is generally acceptable, or the framework itself performs some caching optimizations.
2. **Security issues** : Reflection can bypass Java's access control mechanisms (such as access to `private` fields and methods), breaking encapsulation and potentially leading to data leakage or malicious program tampering. Furthermore, it can bypass generic type checks, posing type safety risks.
3. **Code readability and maintainability** : Excessive use of reflection can make c complex, difficult to understand, and difficult to debug. Errors are often exposed at runtime and are not as easy to find as compile-time errors.

Related reading: Java Reflection: Why is it so slow ?.

What are the application scenarios of reflection?

We may rarely interact directly with Java reflection when writing business code. But you might not realize it, but you're enjoying the benefits of reflection every day! **Many popular frameworks, such as Spring/Spring Boot and MyBatis, make extensive use of reflection at the underlying level** , which makes them so flexible and powerful.

Here are a few simple examples to help you understand.

1. Dependency Injection and Inversion of Control (IoC)

IoC frameworks represented by Spring/Spring Boot scan classes with specific annotations (such as `@Component` , `@Service` , `@Repository` , `@Controller`) at startup, use reflection to instantiate objects (Beans), and inject dependencies through reflection (such as `@Autowired` , constructor injection, etc.).

2. Annotation processing

An annotation itself is just a "marker"; someone needs to read it to understand what needs to be done. Reflection is that "reader." The framework uses reflection to check for specific annotations on classes, methods, and fields, and then executes logic based on the annotation information. For example, if it sees an annotation `@Value` , it uses reflection to read the annotation content, find the corresponding value in the configuration file, and then use reflection to set the value to the field.

3. Dynamic Proxy and AOP

Want to automatically add some functionality before and after a method call (such as logging, initiating a transaction, or performing permission checks)? Aspect-Oriented Programming (AOP) is exactly what this is, and dynamic proxies are a common way to implement AOP. The JDK's built-in dynamic proxies (Proxy and InvocationHandler) rely on reflection. When a proxy object calls methods on the real object, it `Method.invoke` does so through reflection.

```java
public class DebugInvocationHandler implements InvocationHandler {
    private final Object target; //

    public DebugInvocationHandler(Object target) { this.target = target; }

    // proxy:        , method:           , args:
    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
        System.out.println("                " + method.getName() +
"    ");
        //
        Object result = method.invoke(target, args);
        System.out.println("                " + method.getName() +
"    ");
        return result;
    }
}
```

4. Object-Relational Mapping (ORM)

Frameworks like MyBatis and Hibernate can automatically convert rows of database data into Java objects. How do they determine which Java properties correspond to database fields? Again, they rely on reflection. They use reflection to retrieve a list of Java class properties, then map the query results by name or configuration. They then use reflection to call setters or directly modify field values. Conversely, when saving objects to the database, reflection is also used to read property values and construct SQL.

annotation

What is annotation?

`Annotation` (Annotation) is a new feature introduced in Java 5. It can be regarded as a special comment, which is mainly used to modify classes, methods or variables and provide certain information for the program to use during compilation or runtime.

Annotation is essentially a `Annotation` special interface that inherits:

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}
```

```java

public interface Override extends Annotation{

}
```

JDK provides many built-in annotations (such as `@Override` , `@Deprecated`), and we can also customize annotations.

What are the methods for parsing annotations?

Annotations take effect only after being parsed. There are two common parsing methods:

- **Direct scanning during compilation** : The compiler scans and processes the corresponding annotations when compiling Java code. For example, if a method uses `@Override` annotations, the compiler will detect whether the current method overrides the corresponding method of the parent class during compilation.
- **Processing by reflection at runtime** : Annotations that come with the framework (such as the Spring framework `@Value`) `@Component` are all processed through reflection.

⭐️SPI

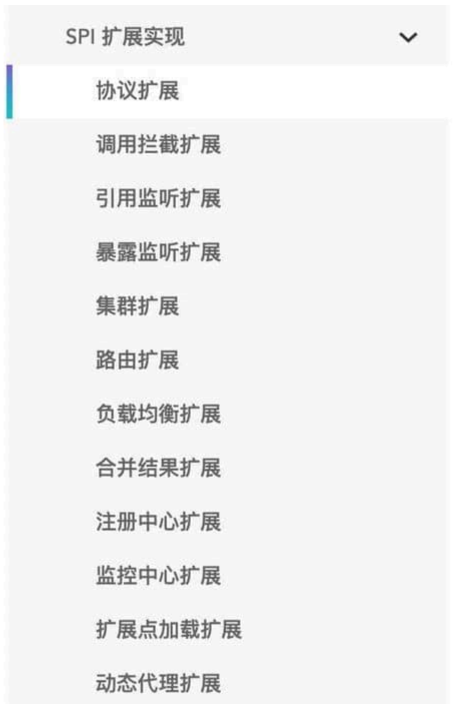
For a detailed explanation of SPI, please see this article Java SPI mechanism detailed explanation .

What is SPI?

SPI stands for Service Provider Interface, which literally means "service provider interface". My understanding is that it is an interface specifically provided to service providers or developers who extend framework functions.

SPI separates the service interface from the specific service implementation, decoupling the service caller from the service implementer, which can improve the scalability and maintainability of the program. Modifying or replacing the service implementation does not require modifying the caller.

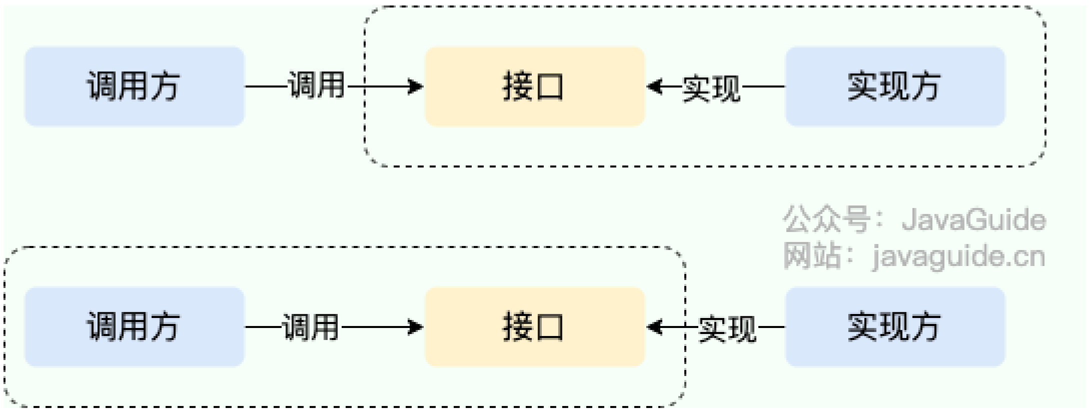
Many frameworks use Java's SPI mechanism, such as the Spring framework, databas loading driver, logging interface, and Dubbo's extended implementation.



What is the difference between SPI and API?

What is the difference between SPI and API?

When talking about SPI, we have to talk about API (Application Programming Interface). In a broad sense, they are both interfaces, and it is easy to confuse them. Here is a picture to illustrate:



Generally, modules communicate with each other through interfaces, so we introduce an "interface" between the service caller and the service implementer (also known as the service provider).

- When an implementer provides an interface and an implementation, we can access the capabilities provided by the implementer by calling the implementer's interface. T called **an API** . In this case, both the interface and the implementation are placed in the

implementer's package. The caller accesses the implementer's functionality through the interface without having to worry about the specific implementation details.
- When an interface exists on the caller side, this is called **an SPI** . The interface caller determines the interface rules, and then different manufacturers implement the interface according to these rules to provide services.

Let's take a simple example: Company H is a technology company that has designed a new chip and now needs to mass-produce it. There are several chip manufacturing companies on the market. At this time, as long as Company H specifies the production standards for this chip (defines the interface standards), then these cooperating chip companies (service providers) will deliver their own unique chips in accordance with the standards (providing different implementation solutions, but the results are the same).

What are the advantages and disadvantages of SPI?

The SPI mechanism can greatly improve the flexibility of interface design, but the SPI mechanism also has some disadvantages, such as:

- It is necessary to traverse and load all implementation classes, and it is not possible to load on demand, so the efficiency is relatively low.
- When multiple `ServiceLoader` are running at the same `load` time, there will be concurrency issues.

⭐Serialization and deserialization

For a detailed explanation of serialization and deserialization, please see this article Java Serialization Detailed Explanation , which covers more comprehensive knowledge points and interview questions.

What is serialization? What is deserialization?

If we need to persist Java objects, such as saving Java objects in files or transmitting Java objects over the network, serialization is required in these scenarios.

In simple terms:

- **Serialization** : Converting a data structure or object into a form that can be store transmitted, usually a binary byte stream, or a text format such as JSON or XML

- **Deserialization** : The process of converting the data generated during the serialization process into the original data structure or object

For object-oriented programming languages like Java, what we serialize are objects (Object), that is, instantiated classes (Class). However, in semi-object-oriented languages like C++, struct (structure) defines the data structure type, while class corresponds to the object type.

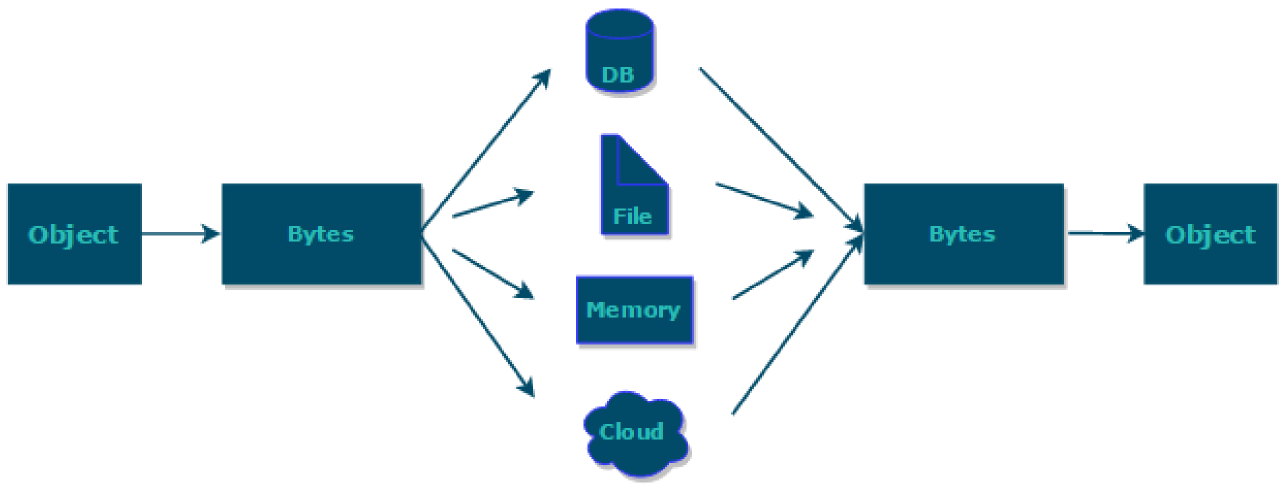
The following are common application scenarios for serialization and deserialization:

- Objects need to be serialized before being transmitted over the network (such as when calling a remote method RPC), and then deserialized after receiving the serialized object;
- Objects need to be serialized before being stored in a file, and objects need to be deserialized before being read from a file;
- Serialization is required before storing objects in a database (such as Redis), and deserialization is required to read objects from the cache database;
- Objects need to be serialized before being stored in memory and deserialized after being read from memory.

Wikipedia describes serialization as follows:

> **In computer science data processing, serialization** refers to the process of converting a data structure or object state into a usable format (such as storing it in a file, storing it in a buffer, or sending it over a network) so that the original state can be restored later in the same or another computer environment. When the bytes are retrieved according to the serialized format, it can be used to produce a copy with the same semantics as the original object. For many objects, such as complex objects that use a large number of references, this serialization and reconstruction process is not easy. Object serialization in object-oriented programming does not generalize the functions associated with the original object. This process is also called object marshalling. The reverse operation of extracting a data structure from a series of bytes is deserialization (also known as unmarshalling, deserialization, or unmarshalling).

To sum up: **the main purpose of serialization is to transmit objects over the network or to store objects in the file system, database, or memory.**

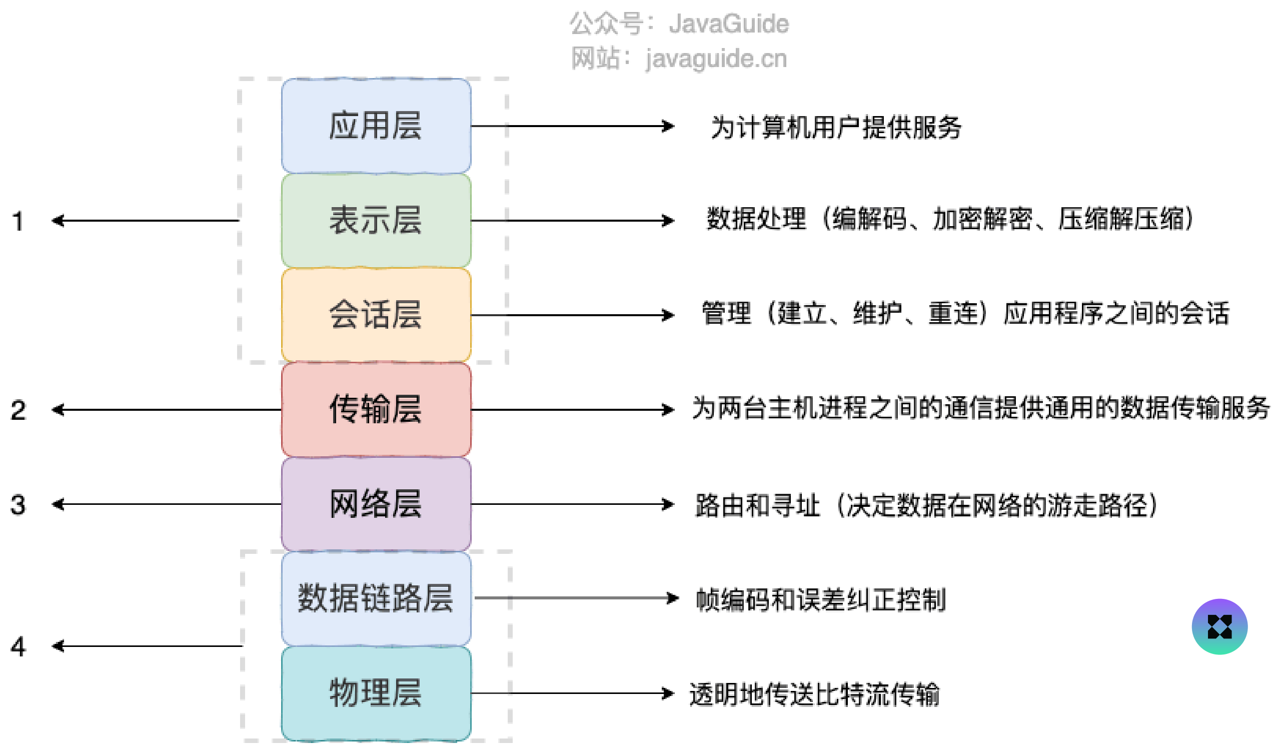


https://www.corejavaguru.com/java/serialization/interview-questions-1

To which layer of the TCP/IP 4-layer model does the serialization protocol correspond?

We know that both parties in network communication must use and adhere to the same protocol. The TCP/IP four-layer model looks like this. To which layer does the serialization protocol belong?

1. Application Layer
2. Transport layer
3. Network layer
4. Network Interface Layer


公众号：JavaGuide
网站：javaguide.cn
应用层
为计算机用户提供服务
1
表示层
数据处理（编解码、加密解密、压缩解压缩）
会话层
管理（建立、维护、重连）应用程序之间的会话
2
传输层
为两台主机进程之间的通信提供通用的数据传输服务
3
网络层
路由和寻址（决定数据在网络的游走路径）
4
数据链路层
帧编码和误差纠正控制
物理层
透明地传送比特流传输

As shown in the figure above, in the OSI seven-layer protocol model, the presentation layer primarily processes the application layer's user data and converts it into a binary stream. Conversely, it converts the binary stream into application layer user data. Doesn't this correspond to serialization and deserialization?

Because the application layer, presentation layer, and session layer in the OSI seven-layer protocol model all correspond to the application layer in the TCP/IP four-layer model, the serialization protocol is part of the TCP/IP protocol application layer.

What if some fields do not want to be serialized?

For variables that you do not want to serialize, use `transient` the keyword modification.

`transient` The function of the keyword is to prevent the serialization of variables in the instance that are modified with this keyword; when the object is deserialized, the `transient` modified variable values will not be persisted and restored.

`transient` A few more notes about :

- `transient` Only variables can be modified, not classes and methods.
- `transient` For modified variables, the variable value will be set to the default value of the type after deserialization. For example, if it is a modified `int` type, the result after deserialization will be `0` .
- `static` Since variables do not belong to any object, `transient` they will not be serialized regardless of whether they are modified by keywords or not.

What are the common serialization protocols?

The JDK's built-in serialization method is generally not used due to its low efficiency and security issues. Commonly used serialization protocols include Hessian, Kryo, Protobuf, and ProtoStuff, all of which are binary-based serialization protocols.

Text serialization methods like JSON and XML have better readability but poor performance and are generally not chosen.

Why is it not recommended to use JDK's own serialization?

We rarely or almost never use the serialization method that comes with JDK directly. The main reasons are as follows:

- **Cross-language calls are not supported** : If the service is developed in other languages, it is not supported.
- **Poor performance** : Compared with other serialization frameworks, the performance is lower. The main reason is that the byte array after serialization is larger, resulting in increased transmission costs.
- **Security issues exist** : Serialization and deserialization themselves aren't inherently problematic. However, if the input data for deserialization is user-controllable, an attacker can construct malicious input to cause deserialization to produce unexpected objects, potentially allowing arbitrary code to execute. Related reading: Application Security: The Tragedy of Java Deserialization Vulnerabilities .

I/O

For a detailed explanation of I/O, please read the following articles, which cover more comprehensive knowledge points and interview questions.

- Java IO basics summary
- Java IO Design Patterns Summary
- Detailed Explanation of the Java IO Model

Do you know Java IO streams?

IO stands for `Input/Output` input and output. Input is the process of inputting data into computer memory, while output is the process of outputting data to external storage (such as a database, file, or remote host). The data transfer process is similar to the flow of water, hence the name IO stream. In Java, IO streams are categorized as input streams and output streams, which are further categorized as byte streams and character streams depending on how the data is processed.

More than 40 Java IO stream classes are derived from the following four abstract base classes.

- `InputStream` / `Reader` : The base class of all input streams, the former is a byte input stream, and the latter is a character input stream.
- `OutputStream` / `Writer` : The base class of all output streams, the former is a byte output stream, and the latter is a character output stream.

Why are I/O streams divided into byte streams and character streams?

The essence of the question is: **whether it is file reading and writing or network sending and receiving, the smallest storage unit of information is byte, so why are I/O stream operations divided into byte stream operations and character stream operations?**

I think there are two main reasons:

- The character stream is obtained by converting bytes by the Java virtual machine, which is a relatively time-consuming process;
- If we don't know the encoding type, garbled characters may easily appear when using byte streams.

What are the design patterns in Java IO?

Reference answer: Java IO design pattern summary

What are the differences between BIO, NIO and AIO?

Reference answer: Detailed explanation of Java IO model

Syntactic sugar

What is syntactic sugar?

Syntactic sugar refers to a special syntax added to a programming language to make it easier for programmers to develop programs. This syntax does not affect the functionality of the programming language. To achieve the same functionality, code written using syntactic sugar is often simpler, more concise, and easier to read.

For example, in Java `for-each` is a commonly used syntactic sugar, and its principle i actually based on ordinary for loops and iterators.

```java
String[] strs = {"JavaGuide", "       JavaGuide", " https://javaguide.cn/"};
for (String s : strs) {
    System.out.println(s);
}
```

However, the JVM doesn't actually recognize syntactic sugar. For Java syntactic sugar to be correctly executed, it must first be desugared by the compiler. This means converting it into basic syntax recognized by the JVM during the program compilation phase. This also indirectly demonstrates that it's the Java compiler, not the JVM, that truly supports syntactic sugar in Java. If you look at `com.sun.tools.javac.main.JavaCompiler` the source code, you'll find `compile()` a step called "call" `desugar()` ; this method is responsible for desugaring.

What are the common syntactic sugars in Java?

The most commonly used syntactic sugars in Java include generics, automatic unpacking and boxing, variable-length parameters, enumerations, inner classes, enhanced for loops, try-with-resources syntax, lambda expressions, etc.

For a detailed explanation of these syntactic sugars, please see this article Java Syntactic Sugar Detailed Explanation .

JavaGuide官方公众号
（微信搜索JavaGuide）
1、公众号后台回复“PDF”获取原创PDF面试手册
2、公众号后台回复“学习路线”获取Java学习路线最新版
3、公众号后台回复“开源”获取优质Java开源项目合集
4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated2025/7/27 10:31

Contributors: guide , sam , CrazyKid , WT-AHA , sleepandsheep , ljgong007 , Guide , Mr.Hope , cxhello , qksuki , HaiBooLang