# Exam 1Z0-817: Upgrade OCP Java 6, 7 & 8 to Java SE 11 Developer Quiz


## Mikalai Zaikin

Belarus
Minsk
<NZaikin[at]iba.by>




October 2020

| **Revision History** | | |
|---|---|---|
| Revision commit fdc5bf26aadbd081f741fe08fb7eecffed07a3e3 | Date: Mon Oct 19 01:12:13 2020 +0300 | Author: mzaikin <mzaikin@172.16.114.135> |
| . | | |
| Revision commit 9598fef486886d5b9ff5158335a6bf22fd9c9d3a | Date: Mon Oct 19 01:00:43 2020 +0300 | Author: mzaikin <mzaikin@172.16.114.135> |
| Improved 030103 question wording | | |
| Revision commit 79ae69504168398270b09b56dc0d70e994c126f0 | Date: Thu Feb 6 21:15:38 2020 +0300 | Author: Mikalai Zaikin <mzaikin@us.ibm.com> |
| Fixed typo (number of correct options) | | |


**Abstract**

The purpose of this document is to help in preparation for Upgrade OCP Java 6, 7 & 8 to Java SE 11 Developer Test (Exam 1Z0-817).

This document should not be used as the only study material for Upgrade OCP Java 6, 7 & 8 to Java SE 11 Developer Test (Exam 1Z0-817). It might cover not all objective topics, and it might be not enough. I tried to make this document as much accurate as possible, but if you find any error, please let me know.


---

### Preface

If you believe you have found an error in the "Upgrade OCP Java 6, 7 & 8 to Java SE 11 Developer Quiz" or have a suggestion to improve it, please send an e-mail to me. Please, indicate the topic and page URL.

# Part I. Exam Objectives

## Chapter 1. Understanding Modules

### 1.1. Describe the Modular JDK

#### Question 010101

Given the code fragment:

```
try (
    Connection conn = DriverManager.getConnection(...);
    Statement stmt = conn.createStatement();) {
    // ...
}
```

Which two actions when done simultaneously will allow the code compile successfully?

*Options (choose 2):*

1. Add to the class definition: `import java.sql.*;`

2. Add to the class definition: `requires java.sql.*;`
3. Add to the module definition: `requires java.base;`
4. Add to the module definition: `requires java.se;`
5. Add to the module definition: `open java.sql;`

*Answer:*

The correct options are 1 and 4.

As of Java SE 9 all standard Java API packages are placed into modules -- JDK became modular.

Standard Java SE modules start with `java.` prefix, ideally your application should "require" only `java.*` modules.

The non-standard JDK modules start with `jdk.` prefix, they include debugging and serviceability tools and APIs, development tools, and various service providers, which are made available to other modules via the existing `java.util.ServiceLoader` mechanism.

Option 1 is correct, even if you "require" a correct Java SE module, you still need to import the package (or individual classes) in order to use short form of class name (`Connection` v.s. `java.sql.Connection`). NOTE: two packages are always implicitly imported in Java applications: current package where the class is located, and `java.lang` package (which contains such common classes as `String`, `Object`, `System`, etc..).

Option 2 is wrong, you need to use "`import`" keyword in order to import package/class.

Option 3 is wrong. The `java.base` standard Java SE module indeed contains some common packages (e.g. `java.io`, `java.lang`, `java.nio`, `java.util`, `java.time`, and many more), but the JDBC classes are placed into own `java.sql` module.

Option 4 is correct. Java SE has a special **aggregator module** named `java.se`, which does not contain any own package, rather "`requires transitive`" all standard `java.*` modules. The "`transitive`" modifier means that what "`required`" in `java.se` (i.e. all standard Java SE modules) will be required to client code as soon as client module definition has "`requires java.se;`"

Another approach to compile the code would be to require specifically the SQL module: `requires java.sql;`

Option 5 is wrong: A module is allowed to open only own packages. It does not make sense and violates encapsulation to open a package from some third-party module. And the `open` directive will not provide visibility of JDBC classes to the application.

*Source:*

Understanding Java 9 Modules [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

**Question 010102**

You are writing a modular application for Java 11 platform which uses SAX XML parser.

From JavaDoc you know that the `javax.xml.parsers.SAXParser` parser class belongs to `java.xml` module.

Which step you must do in order to use this class in your application?

*Options (choose 1):*

1. Add the option to `java` and `javac`: `-p java.xml`
2. Add the option to `java` and `javac`: `-cp java.xml.mod`
3. Add directive to the module definition: `requires java.xml;`
4. Add directive to the module definition: `requires java.base;`

*Answer:*

The correct option is 3.

Option 1 is wrong. It does not cause error, but it is useless and does not make sense. The `-p` is a short form of `--module-path` option, which defines location where modules are looked up when creating a module graph during compile time or runtime.

Standard Java SE modules start with `java.` prefix (e.g. `java.xml`) and they are avalable by design, without need to add them to module path.

Option 2 is wrong. It does not cause error, but it is useless. The `-cp` is a short form of `--class-path` (or `-classpath`) option, which defines location where classes are looked up when classloader needs to load a class to JVM.

There is a problem with this option for several reasons:

- Standard Java modules are available by design without adding to classpath or module path.
- Modules are added to modulepath.
- To classpath you can add `.jar` files or directories
- When something added and loaded later via classpath it becomes part of unnamed module in Java 9 and higher, the unnamed module cannot be accessed from a named module (the opposite direction access is possible).

Option 3 is correct. When we define some module as "required" in module definition, it will be added to modules graph at compile time or runtime, and classes/interfaces from this module can be seen by JVM classloader.

Option 4 is wrong. The `java.base` module always implicitly added in modular applications, even if you do not "require" it. But `java.base` contains only some common packages (e.g. `java.lang`). The XML parsing classes/interfaces are stored in a separate module (`java.xml`) which must be direcly or indirectly "required".

A possible option which would work can be requiring Java SE aggregator module in module definition: `requires java.se;`

*Source:*

Understanding Java 9 Modules [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

**Question 010103**

While compiling your modular application you got the error from `javac`:

```
import java.sql.Connection;
           ^
  (package java.sql is declared in module java.sql, but module modS does not read it)
1 error
```

Which two changes when done independently will resolve this error?

*Options (choose 2):*

1. Run `javac` with option: `--module-path java.sql`
2. Run `javac` with option: `-modulepath java.sql`
3. Run `javac` with option: `-mp java.sql`
4. Add directive to the module definition: `requires java.sql;`
5. Add directive to the module definition: `requires transitive java.sql;`

*Answer:*

The correct options are 4 and 5.

Option 1 is wrong. It does not cause error, but it is useless and does not make sense. The `--module-path` option defines location where modules are looked up when creating a module graph during compile time or runtime.

Standard Java SE modules start with `java.` prefix (e.g. `java.sql`) and they are avalable by design, without need to add them to module path.

Options 2 and 3 are wrong. These are invalid options (flags) for `javac` compiler. You need to remember common options for `java` and `javac`.

Option 4 is correct. When we define some module as "required" in moodule definition, it will be added to modules graph at compile time or runtime, and classes/interfaces from this module can be seen by JVM classloader.

The `requires` directive specifies the name of a module (i.e. `java.sql`) on which the current module has a dependence.

Option 5 is correct. It is similar to option 4 above, but with some "enhancement".

The `requires` keyword may be followed by the modifier `transitive`. This causes any module which requires the current module to have an implicitly declared dependence on the module specified by the `requires transitive` directive. In other words, when some other module "requires" our application module, it will implicitly "require" `java.sql` module and will not

need to require it explicitly via separate `requires java.sql;` directive.

*Source:*

Understanding Java 9 Modules [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

JLS 11. Section 7.7.1. Dependences [https://docs.oracle.com/javase/specs/jls/se11/html/jls-7.html#jls-7.7.1]

**Question 010104**

Which two statements are correct about Java modules?

*Options (choose 2):*

1. Automatic module name is defined in `module-info.java` which is compiled later to `module-info.class`
2. The `java.base` standard module exports Java core packages
3. A `module-info.class` automatically found by JVM inside any sub-folder on module path
4. One module automatically can use another module as long as both are listed on module path
5. A `module-info.java` with zero directives is a valid module definition

*Answer:*

The correct options are 2 and 5.

Option 1 is wrong. Application modules may be classified as:

1. Named. They have a `module-info.class` definition with a module name and [optionally] some directives.
2. Automatic. A JAR file which does not have `module-info.class`, and the module name is derived either from JAR file name, or from `Automatic-Module-Name` header (if any) in `META-INF/MANIFEST.MF`
3. Unnamed. In most cases there will be one per application (but saying more accurately -- one per classloader), a catch-all module for all classes loader via class path.

So, we can see that **automatic module** may not have `module-info.class`, otherwise it would be **named module**.

Option 2 is correct. The `java.base` module is a standard Java module which contains core Java platform packages and exports them to other modules. It does not depend on any Java module, but other modules depend on it. This module is always implicitly "required" by JVM, so you don't need to do it explicitly in your module definitions. You can check what modules exported by running the command:

```
jmod describe java.base.jmod

java.base@11.0.2
...
exports java.io
exports java.lang
exports java.math
exports java.net
exports java.nio
exports java.security
exports java.text
exports java.time
exports java.util
exports javax.crypto
exports javax.net
...
```

Option 3 is wrong. You have 2 places where you can put `module-info.class` to have JVM recognize it:

1. In case of JAR file, the `module-info.class` must be in the root (the top most) directory of the archive, e.g.

```
C:\1Z0-817>jar -tvf modA.jar
     0 Sun Apr 21 20:38:34 AST 2019 META-INF/
    66 Sun Apr 21 20:38:34 AST 2019 META-INF/MANIFEST.MF
   211 Sun Apr 21 20:38:34 AST 2019 module-info.class
     0 Sun Apr 21 20:24:42 AST 2019 pkgA/
```

```
   451 Sun Apr 21 20:24:42 AST 2019 pkgA/ClassA.class
```

2. In case of expanded code (bytecode stored in the various subfolders on the filesystem), the `module-info.class` must be exactly inside the folder which is listed on module path, e.g.:

```
java --module-path ./mod1 --module
by.boot.java.mod.logger/by.boot.java.pkg.logger.MyLogger
```

then

```
mod1
•
••• module-info.class
•
••••by
    ••••boot
        ••••java
            ••••pkg
                ••••logger
                    •••• MyLogger.class
```

With this said, JVM will NOT find module definition in any arbitrary sub-folder on module path.

Option 4 is wrong. Placing two modules on the module path is not sufficient for one module classes can use another module classes. Placing on module path is a pre-requisite for one module may use another, but in order to make classes accessible, one module must "require" another module, and another module must "export" a package with classes. NOTE: automatic modules export all own packages and require all modules on the module path. But named modules need to do it explicily.

Option 5 is correct. The most minimalistic valid `module-info.java` definition will look as follows:

```
module module_name {
}
```

As we see, the only mandatory attribute is module name, but all directives are optional (empty curly baces).

*Source:*

Understanding Java 9 Modules [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

JLS 11. Section 7.7.1. Dependences [https://docs.oracle.com/javase/specs/jls/se11/html/jls-7.html#jls-7.7.1]

## 1.2. Declare modules and enable access between modules

**Question 010201**

Given the application consisting of 2 modules:

- `modA` module contains `pkgA` package
- `modB` module contains `pkgB` package
- classes in `pkgB` package depend on classes from `pkgA` package

Which two actions when done simultaneously will allow the code to compile successfully?

*Options (choose 2):*

1. `modA` module must open `pkgA` package
2. `modA` module must export `pkgA` package
3. `modB` module must require `pkgA` package
4. `modB` module must require `modA` module

*Answer:*

The correct options are 2 and 4.

Option 1 is wrong: the `opens` directive grants only **runtime visibility** for `public` and `protected` classes/members from the module (as well Reflection API access to all classes/members). The question specifically says about compilations.

Option 2 is correct. The `exports` directive grants compile time and runtime access to `public` and `protected` memebers from the exported package(s).

Option 3 is wrong. The `requires` directive define a required module, not a package. Based on this information the module dependencies graph is created.

Option 4 is correct. Since `modB` module classes require some classes from `modA` module at compile time, `modB` must "require" `modA` in own module definition.

*Source:*

JLS 11. Section 7.7. Module Declarations [https://docs.oracle.com/javase/specs/jls/se11/html/jls-7.html#jls-7.7]

**Question 010202**

Given the application consisting of 2 modules:

- `modA` module contains `pkgA` package
- `modB` module contains `pkgB` package
- classes in `pkgB` package depend on classes from `pkgA` package

Which action among other must a developer take in order to run the application successfully?

*Options (choose 1):*

1. `modA` module must open `pkgA` package
2. `modA` module must require `modB` module
3. `modB` module must import `modA` module
4. `modB` module must open `modA` module

*Answer:*

The correct option is 1.

Option 1 is correct: the `opens` directive grants **runtime visibility** for `public` and `protected` classes/members from the module (as well Reflection API access to all classes/members, including `private`). The question specifically asks what is needed to run the application successfully.


The `opens` directive specifies the name of a package to be opened by the current module. For code in other modules, this grants access at run time, but not compile time, to the `public` and `protected` types in the package, and the `public` and `protected` members of those types. It also grants reflective access to all types in the package, and all their members, for code in other modules.

Option 2 is wrong. The depedency direction of modules is pointing from `modB` to `modA`. So, the opposite would be true: `modB` module must require `modA` module.

Option 3 is wrong. When a module depends on another module, it must use `requires` directive. The `import` is a statement inside class definition for using short form of class/interface names later in the class body, or for using short form of `static` members (methods/fields).

Option 4 is wrong. First off, a module may open a package, not a module. Second, a module may open only **own** package(s), it may not open anything from third-party module.

*Source:*

JLS 11. Section 7.7. Module Declarations [https://docs.oracle.com/javase/specs/jls/se11/html/jls-7.html#jls-7.7]

**Question 010203**

Which statement is true about Java modules?

*Options (choose 1):*

1. Java module may contain at most 1 package.
2. Java package may be splitted between several modules.
3. Java module must be named as the top level package which the module contains.
4. Java module must contain at least one package.
5. In Java module definition all directives are optional.

*Answer:*

The correct option is 5.

Option 1 is wrong. For sure a module may contain more than one package. The most common example is `java.base` module which contains about 51 package in Java SE 11.

Option 2 is wrong. The definition of the module is that it must fully contain a package. A Java package may not be splitted between several modules.

Option 3 is wrong. It is recommended to name a Java module the same as the name of the root Java package contained in the module -- if that is possible (some modules might contain multiple root packages). But this is not mandatory requiement, only recommendation. For example, `java.sql` module contains `java.sql` and `javax.sql` packages.

Option 4 is wrong. There are so called "aggregate modules" which only contain "`requires transitive ...;`" directives and do not contain own packages. The most common example is `java.se` module which contains inside only `module-info.class` and is only 2208 bytes length in Java 11. But when user imports it -- he gets access to all Java SE modules.

Option 5 is correct. This is the module declaration grammar:

```
ModuleDeclaration:
  {Annotation} [open] module Identifier {. Identifier} { {ModuleDirective} }
```

```
ModuleDirective:
  requires {RequiresModifier} ModuleName ;
  exports PackageName [to ModuleName {, ModuleName}] ;
  opens PackageName [to ModuleName {, ModuleName}] ;
  uses TypeName ;
  provides TypeName with TypeName {, TypeName} ;

RequiresModifier:
  (one of)
  transitive static
```

As you can see only `module Identifier { }` part is mandatory for module definition and empty curly braces is a valid syntax.

*Source:*

JLS 11. Section 7.7. Module Declarations [https://docs.oracle.com/javase/specs/jls/se11/html/jls-7.html#jls-7.7]

## 1.3. Describe how a modular project is compiled and run

**Question 010301**

Given:

- An application module `greeter` packaged in the `greeter.jar` file.
- The `p2.Client` class depends on some classes from the `greeter` module.
- Layout of the modular application is as follows:

```
•
••••mod
•       greeter.jar
•
••••src
```

```
    •   module-info.java
    •
    ••••p2
            Client.java
```

Which two commands when run independently will compile the `.java` source code?

*Options (choose 2):*

1. `javac --module-path mod src/module-info.java src/p2/Client.java`
2. `javac --module-path mod/greeter.jar src/module-info.java src/p2/Client.java`
3. `javac -classpath mod src/module-info.java src/p2/Client.java`
4. `javac -cp mod/greeter.jar src/module-info.java src/p2/Client.java`
5. `javac -mp mod/greeter.jar src/module-info.java src/p2/Client.java`

*Answer:*

The correct options are 1 and 2.

When you compile code of application module (i.e. not "automatic" module which is a plain old JAR), then you need to provide location of all dependent modules via "module path".

We can assume that `src/module-info.java` might contain something like this:

```
module client {
    requires greeter;
}
```

Therefore, for successful compilation we need to provide `--module-path` option.

Option 1 is correct. The `--module-path mod` option will tell compiler to look in the `mod` directory for the required modules. The "module path" option is similar to pre-Java 9 "class path" option. The `--module-path` option uses "path separator" character if needed, i.e. semicolon on Windows OS, or colon on Linux OS/Mac OS.

Option 2 is correct. Like with "class path" you can provide a direct JAR file location(s) and compiler will resolve module dependecies by looking at the JAR(s).

Option 3 is wrong. The dependecies loaded via "class path" will be added to "unnamed module". The unnamed module classes may access named module classes, but not the opposite. Compilation will fail with error:

```
src\module-info.java:2: error: module not found: greeter
    requires greeter;
             ^
1 error
```

Option 4 is wrong. Same as option 3. The `-cp` is synonym for `-classpath` (along with `--class-path`), but the problem is that all class path loaded classes are part of unnamed module which is not visible to named module.

Option 5 is wrong. The `-mp` is a WRONG synonym for "module path". The correct synonym is `-p`. Compilation fails with the error:

```
error: invalid flag: -mp
Usage: javac <options> <source files>
use --help for a list of possible options
```

This command would compile sucessfully the `.java` source code: `javac -p mod/greeter.jar src/module-info.java src/p2/Client.java`

*Source:*

javac Tool Reference [https://docs.oracle.com/en/java/javase/11/tools/javac.html]

**Question 010302**

Given:

- An application module `greeter` is packaged in the `greeter.jar` file.
- An application module `client` is packaged in the `client.jar` file.
- The `client` module depends on classes from the `greeter` module.
- The `client` JAR was created by this command:

```
jar --create --file=client.jar --main-class=pkg.Main -C . .
```

- Both JARs are located in the current directory.

Which three commands when run independently will run the `pkg.Main` class?

*Options (choose 3):*

1. `java -p . -m client`
2. `java --module-path . --module client/pkg.Main`
3. `java -cp greeter.jar;client.jar -m client/pkg.Main`
4. `java -cp greeter.jar;client.jar pkg.Main`
5. `java -cp greeter.jar;client.jar -jar client.jar`

*Answer:*

The correct options are 1, 2 and 4.

Option 1 is correct. This option demonstrates syntaxt to run a runnable module (the one which has entry class)

The `-p` option is synonym of `--module-path`, and the `.` path means current directory (where JARs are located), so modules when searched will be looked up in the current directory.

The `-m` option is synonym of `--module` option which defines the module of the class to run.

Option 2 is correct. It demonstrates running a class from a module. It's not necessary to provide class name in this case, as the `client` module already defines entry point, but it's not a problem for Java interpreter.

Option 3 is wrong. The `-m` option assumes `client` module contains the class to run, but the module never get resolved, because there is no `--module-path` option, and classes loaded via `-cp` option are part of unnamd module, and `client` module still not visible.

The result will be as follows:

```
java -cp greeter.jar;client.jar -m client/pkg.Main
Error occurred during initialization of boot layer
java.lang.module.FindException: Module client not found
```

Option 4 is correct. This is pre-Java 9 approach, which still working (in most of cases, like if you do not use internal APIs or deprecated classes) for backward compatibility.

Option 5 is wrong. You can use either `-jar` or `-cp`, you can not combine the two. If you want to put additional JARs on the classpath then you should either put them in the `client.jar`'s MANIFEST.MF and then use java `-jar` or you put the full classpath (including the main JAR and its dependencies) in `-cp` and name the main class explicitly on the command line (this is what option 4 does). With this said, the `greeter.jar` is not visible to class loader and the command fails with the `NoClassDefFoundError` error:

```
Exception in thread "main" java.lang.NoClassDefFoundError: p1/Greeter
        at pkg.Main.main(Main.java:7)
Caused by: java.lang.ClassNotFoundException: p1.Greeter
        at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:583)
        at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:178)
```

```
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
        ... 1 more
```

*Source:*

java Tool Reference [https://docs.oracle.com/en/java/javase/11/tools/java.html]

**Question 010303**

Given:

- An application module `modA` is packaged in the `modA.jar` file.
- An application module `modB` is packaged in the `modB.jar` file.
- An application module `modC` is packaged in the `modC.jar` file.
- All the JAR files are located in the same directory.
- The `modC` module depends on classes from the `modB` module.
- The `modB` module depends on classes from the `modA` module.

Which option can be used to run a class from `modC` module?

*Options (choose 1):*

1. `--module-path modA.jar;modB.jar;modC.jar` on Windows platform
2. `--module-path modA.jar,modB.jar,modC.jar` on all platforms
3. `--module-path modA:modB:modC` on Linux platform
4. `--module-path modA,modB,modC` on all platforms

*Answer:*

The correct option is 1.

All three modules must me added to module path, as all three needed for the code to run properly.

The `--module-path` option may list directories, each directory is a directory of modules.

Also `--module-path` option may list modular JAR files.

In any case, the list is separated with the path separator (a `:` on Unix/Linux/Mac, and `;` on Windows).

Option 2 is wrong. The comma will not be recognized as a path separator and the whole text after the option (`modA.jar,modB.jar,modC.jar`) will be treated as a huge name of a single module, the result -- `modC` will not be resolved and loaded.

Option 3 is wrong. The separator for the list is correct, we use colon on Linux, Unix, Mac OS X. But the list must contain either directory names or JAR file names, not module names. The correct module path option for Linux would be like that:

```
java --module-path modA.jar:modB.jar:modC.jar ...
```

Option 4 is wrong. Same as option 2. Note, some `java` options, like `--add-modules`, use a comma to separate list of root modules beyond the initial module. But the the `--module-path` option uses path separator (platform dependent), not comma.

*Source:*

java Tool Reference [https://docs.oracle.com/en/java/javase/11/tools/java.html]

## Chapter 2. Services in a Modular Application

### 2.1. Describe the components of Services including directives

**Question 020101**

Given the modular Service application consisting of three modules:

- The `modI` module contains Service Interface
- The `modP` module contains Service Provider
- The `modC` module contains Service Consumer

What is true about the modular application?

*Options (choose 1):*

1. The `modI` module definition must have `requires` directive with the service provider module name
2. The `modC` module definition must have `requires` directive with the service provider module name
3. The service implementation class from `modP` module must implement service interface from `modI` module
4. The `modP` module definition must have `exports` directive with the package name of the service implementation class
5. The `modC` module definition must have `requires` directive with the service interface module name

*Answer:*

The correct option is 5.

Option 1 is wrong. It does not make sense for service interface module to know which module implements the interface. Moreover, iterface module may not know and must know nothing about implementation modules, which may be switched easily my modifying module path for startup command.

The minimalistic mandatory module definition of service interface module is as follows:

```
module modI {
    exports [service interface package name];
}
```

Option 2 is wrong. Also, it does not make sence. The idea of services is -- switch implementations easily and make the code resolve implementation at runtime by looking at module path.

The minimalistic mandatory module definition of consumer module is as follows:

```
module modC {
    requires modI;
    uses [fuly qualified service interface name];
}
```

Option 3 is wrong. Implementing a service inerafce is not a mandatory for service provider.

A service provider is a single type, usually a concrete class. An interface or abstract class is permitted because it may declare a `static provider()` method. The type must be `public` and must not be an inner class.

There are two ways to create provider:

- If the service provider declares a `provider()` method, then the service loader invokes that method to obtain an instance of the service provider. A provider method is a `public static` method named "`provider`" with no formal parameters and a return type that is assignable to the service's interface or class.

  In this case, the service provider itself need not be assignable to the service's interface or class.
- If the service provider does not declare a `provider()` method, then the service provider is instantiated directly, via its provider constructor. A provider constructor is a `public` constructor with no formal parameters.

  In this case, the service provider must be assignable to the service's interface or class

Option 4 is wrong. Module definition uses "`provides`" directive instead of "`exports`" to provide information about implementation class and grant access to implementation class.

The minimalistic mandatory module definition of provider module is as follows:

```
module modP {
    requires modI;
    provides [fully qualified service interface name] with [fully qualified service
implementation class name];
}
```

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

**Question 020102**

Which two statements are true about modular Service application?

*Options (choose 2):*

1. Service Provider class must have `public` no-args constructor.
2. Service Provider must be a `public` type.
3. Service Provider must have a `public static provider()` method which returns an instance of the type which is assignable to the Service's type.
4. At runtime Service Locator returns the first provider found on the module path.
5. Services mechanism can be used to break cyclic dependencies.

*Answer:*

The correct options are 2 and 5.

Option 1 is wrong. Service Provider may be a class (concrete or abstract) or an interface. In case of interface no constructors can be provided. In general, Service Providers can be designed in two ways (and instantiated in two ways too):

- If the Service Provider declares a `public static provider()` method, then the service loader invokes that method to obtain an instance of the Service Provider. A provider method is a `public static` method named "`provider`" with no formal parameters and a return type that is assignable to the service's interface or class.

  In this case, the Service Provider itself need not be assignable to the service's interface or class.

  For this case Service Provider can be a concrete class, an abstract class, or an interface (NOTE: as of Java 8 interfaces can have static methods.)
- If the Service Provider does not declare a `provider()` method, then the Service Provider is instantiated directly, via its provider constructor. A provider constructor is a public constructor with no formal parameters.

  In this case, the service provider must be assignable to the service's interface or class

  For this case Service Provider can only be a concrete class.

Option 2 is correct. Service Provider type must be `public`.

Service Provider type may not be `enum`. Service Provider type must be `public` and must not be an inner class.

Option 3 is wrong. As was described above, `public static provider()` is only one of the options to desigh Service Provider, another option is to use concrete class which implements service interface and has a no-args constructor.

Option 4 is wrong. The `ServiceLoader` class has several methods to locate servide, only one of them is `Optional<S> findFirst•()` which loads the first available Service Provider of this loader's Service.

The other two methods to access Service Providers are:

- `Iterator<S> iterator()` - lets you iterate over the instantiated Service Providers.
- `public Stream<ServiceLoader.Provider<S>> stream•()` - lets you stream over Service Providers, which are wrapped into a `Provider` instance.

Option 5 is correct. Cyclic dependencies on the modules are not allowed.

For example, `modA` module definition has `reqires modB;` and `modB` module definition has `reqires modA;`.

In that case you can refactor `Service` interface from `modA` to `modS`, replace in `modB` module definition `reqires modA;` with `reqires modS;` and `uses Service;`.

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

**Question 020103**

Which two statements are true about modular Service type?

*Options (choose 2):*

1. Service type can be an interface.
2. Service type may NOT be an abstract class.
3. Service type can be a concrete class.
4. Service type may NOT be a final class.
5. Service type can be an enum.

*Answer:*

The correct options are 1 and 3.

A Service can be a concrete class, a concrete final class, an abstract class, or an interface.

Using a concrete class (especially a `final` one) as a Service is a bad practice. The idea of Service is to provide different Service Providers (implementations) at runtime. For this reason, a Service should always be an abstract class or an interface.

A Service CAN NOT be an enum. Compiler will fail to compile service consumer module definition which has the `uses` directive as follows:

```
uses service.ServiceType;
```

Where `ServiceType` is an enum:

```
package service;

public enum ServiceType {
    ONE
}
```

```
module-info.java:3: error: the service definition is an enum: ServiceType
    uses service.ServiceType;
                 ^
1 error
```

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

## 2.2. Design a service type, load the services using `ServiceLoader`, check for dependencies of the services including consumer module and provider module

**Question 020201**

You are developing a modular application which uses Services. There could be hundreds of service providers from different vendors. For performance improving and memory saving reasons you need to analyze vendor's Service Provider class name before creating an instance of the provider. Which service loader method allows you to implement this requirement?

*Options (choose 1):*

1. `ServiceLoader.findFirst()`
2. `ServiceLoader.iterator()`
3. `ServiceLoader.stream()`
4. `ServiceLoader.load•(...)`

*Answer:*

The correct option is 3.

Let's start with the last option. There are several overloaded `public static load(...)` methods in `ServiceLoader` class.

All of the `load(...)` methods accept Service type class as one of the parameters, and create an instance of the `ServiceLoader` specifically for the particular Service. Therefore, they are factory methods for `ServiceLoader` and only after `load(...)` invoked we have a proper instance of Service Loader to get instance(s) of Service Provider. This makes option 4 wrong.

Option 1 is wrong for several reasons. First, it loads the first available Service Provider of this loader's Service. Second, it returns an instance and we cannot control class name before instance created, this makes application requirement not implemented.

Option 2 is wrong. While it returns an iterator to lazily load and instantiate the available Providers of this loader's Service, it still does not allow us to analyze Service Provider class name before creating an instance. When we call `next()` method on the returned iterator it locates and instantiates (if not cached) the next available Provider.

Option 3 is correct. This method returns a stream to lazily load available Providers of this loader's Service. The stream elements are of type `ServiceLoader.Provider`, the `Provider`'s `get()` method must be invoked to get or instantiate the provider. But before we do this, in order to analyze class name of the Provider, there is another method in `Provider` class: `type()` which returns `Class` of the Provider type without creating of instance of it.

This example demonstrates how can we implement requirement for the application:

```
import java.util.ServiceLoader;
import java.util.ServiceLoader.Provider;
...
String vendor = "Oracle";
CodecFactory cf = ServiceLoader.load(CodecFactory.class)
                               .stream()
                               .filter(p -> p.type().getName().contains(vendor)) //
analyze Provider type name
                               .map(Provider::get) // create an instance
                               .findAny()
                               .orElse(new DefaultCodecFactory());
```

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

**Question 020202**

You have developed a `mod.service` Services module with a `pkg.service.MonitoringService` interface.

Now you started working on `mod.provider` Service Provider module with a `pkg.provider.MonitoringServiceImpl` class. Current module definition looks as follows:

```
module mod.provider {
    requires mod.service;
    provides pkg.service.MonitoringService with pkg.provider.MonitoringServiceImpl;
}
```

Which statement is true about `mod.provider` module definition?

*Options (choose 1):*

1. You need to add `uses pkg.service.MonitoringService;` directive.
2. You need to remove `requires mod.service;` directive.
3. You need to add `exports pkg.provider;` directive.
4. The current module definition is correct and does not require any directive.

*Answer:*

The correct option is 4.

Option 1 is wrong. The `uses` directive should be used in Consumer module.

Option 2 is wrong. In order to compile successfully the `mod.provider` must require the `mod.service` module. Otherwise it simply can not refer the Servie type.

Option 3 is wrong. The `exports` directive with Provider class package name is NOT needed. The `provides` directive is sufficient to have JRE access to the class and provide instance to Consumer.

Based on the said above, we can conclude that module definition for the Service Provider module is syntactically and logically correct and does not require any changes.

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

**Question 020203**

Assuming Service, Provider, and Consumer modules are separate. Which two directives are mandatory in the Consumer module definition?

*Options (choose 2):*

1. `uses <provider module name>;`
2. `uses <service module name>;`
3. `uses <service type name>;`
4. `requires <provider module name>;`
5. `requires <service module name>;`

*Answer:*

The correct options are 3 and 5.

Option 1 and 2 are wrong. The `uses` directive may be followed only by Java type, not by module name. And the Java type must be a Service fully qualified (i.e. with the package name prepended) type name (class or interface).

Option 4 is wrong. It may be present, but it would be a bad practice as Provider module never used directly by Consumer:

- Provider instance created and returned by JVM to Consumer, not by Consumer itself
- The idea of services is to decouple real implementation from service. Implemenmtation (provider) is resolved at runtime, and is not coded in module definition explicitly. Moreover, Provider module may not be created at the time when Consumer module written.

Option 5 is correct. Consumer must require Service module at compile- and run-time, as it must refer and use Service type in it's code.

*Source:*

ServiceLoader JavaDoc [https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html]

**Question 020204**

You are migrating Service application created for Java 8 version to Java 11. You need to integrate old Java 8 based Consumer and Provier JARs in the new modular application. Which two statements are correct about the migration to new Java version?

*Options (choose 2):*

1. Place the unchanged Consumer JAR file on the module path.
2. You have to add to the Consumer JAR file new module definition with the `uses` directive and put the JAR on the module path.
3. Place the unchanged Provider JAR file on the module path.
4. You have to add to the Provider JAR file new module definition with the `provides` directive and put the JAR on the module path.

*Answer:*

The correct options are 1 and 3.

Option 1 is correct. By placing a Java 8 created (i.e. non-modular) JAR on the module path you make it an automatic module. An automatic module is considered to use every available service. No explicit module definition with `uses` directive

is required.

Option 2 is wrong. Indeed you CAN add new module definition, but it's NOT mandatory (note: the question says "have to"). At the minimum, adding the Consumer JAR to the module path will be sufficient (see Option 1).

Option 3 is correct. The Java 8 and below Service Provider JAR file was created as follows: to register your Service Provider, you create a provider configuration file, which is stored in the `META-INF/services` directory of the Service Provider's JAR file. The name of the configuration file is the fully qualified type name of the Service (e.g. `java.sql.Driver`, and the full file name inside JAR would be: `/META-INF/services/java.sql.Driver`).

The provider configuration file contains the fully qualified class names of your Service Providers, one name per line.

For example, to register two Service Providers `com.mysql.jdbc.Driver` and `com.mysql.fabric.jdbc.FabricMySQLDriver` the file will contain two lines:

```
com.mysql.jdbc.Driver
com.mysql.fabric.jdbc.FabricMySQLDriver
```

When you place a Java 8 created (i.e. non-modular) JAR on the module path you make it an automatic module. For migration purposes, if a JAR file that defines an automatic module contains `META-INF/services` resource entries then each such entry is treated as if it were a corresponding `provides` clause in a hypothetical declaration of that module. With this said, JVM will register implementation without explicit module definition.

Option 4 is wrong. Indeed you CAN add new module definition, but it's NOT mandatory (note: the question says "have to"). At the minimum, adding the Provider JAR to the module path will be sufficient (see Option 3).

*Source:*

The State of the Module System [http://openjdk.java.net/projects/jigsaw/spec/sotms/#services]

## Chapter 3. Java Interfaces

### 3.1. Create and use methods in interfaces

#### Question 030101

Given the `C1` class:

```
class C1 implements I1, I2 { }
```

Which two `I1` and `I2` pair will compile successfully along with `C1` class?

*Options (choose 2):*

1.

```
interface I1 {
    public void doIt() { }
}
interface I2 {
    private void doIt() { }
}
```

2.

```
interface I1 {
    default public void doIt() { }
}
interface I2 {
    public static void doIt() { }
}
```

3.

```
interface I1 {
    public static void doIt() { }
}
interface I2 {
    public static void doIt() { }
}
```

4.

```
interface I1 {
    default public void doIt() { }
}
interface I2 {
    default public void doIt() { }
}
```

5.

```
interface I1 {
    private static void doIt() { }
}
interface I2 {
    private abstract void doIt() { }
}
```

*Answer:*

The correct options are 2 and 3.

Option 1 is wrong. Interface `I2` is OK, it may declare `private` non-abstract methods as of Java 9:

```
interface I2 {
    private void doIt() { } // Allowed 'private' since Java 9
}
```

But interface `I1` will not compile.

```
interface I1 {
    public void doIt() { } // Fail without 'default' keyword
}
```

If an interfcace declares `public` non-abstract method, it must provide `default` keyword before the method declaration:

```
interface I1 {
    default public void doIt() { } // Allowed since Java 8
}
```

Since `I1` fails to compile, the `C1` class will not compile too.

Option 2 is correct.

Interface `I1` correctly declares non-abstract `public` method:

```
```

```
interface I1 {
    default public void doIt() { } // Correct declaration of public non-abstract method
}
```

And interface `I2` correctly declares `public static` non-abstract method, allowed since Java 8:

```
interface I2 {
    public static void doIt() { }
}
```

Now, the interfaces are good. What about `C1` class? Will it compile as the interfaces declare method with the same name? The class will compile OK, because "`default`" (i.e. "instance") method from `I1` and `static` method from `I2` will not conflict. The scope of the `static` method is interface only, and class may not call the method from own methods without prepending the method with interface name.

Consider this valid code when a class extends another class with `public static` method:

```
class S1 {
    public static void method1() {}
}
class S2 extends S1 {
    public static void method2() {
        method1();
    }
}
```

The same situation is not possible in case of interface implementation:

```
interface S1 {
    public static void method1() {}
}
class S2 implements S1 {
    public static void method2() {
        method1(); // Compilation fails ! Method not in scope of S2
    }
}
```

In order to call `method1()` from inside `S2` class, the method name MUST be prepended with interface name:

```
interface S1 {
    public static void method1() {}
}
class S2 implements S1 {
    public static void method2() {
        S1.method1(); // OK
    }
}
```

With this said, you can see that both `doIt()` methods valid and do not conflict and the code successfully compiles.

Option 3 is correct. Similar to option 2, interfaces declare two `public static` methods with the same name. But the scope of the each method is own declaring interface, they do not conflict when `C1` class implements both interfaces. The interfaces and the class compile OK.

Option 4 is wrong. The interfaces are OK, but the class may not implement two interfaces with the same `default` method without resolving conflict, in other words without declaring and implementing own copy of the same method.

This `C1` class would make the option 4 correct:

```
class C1 implements I1, I2 {
```

```
    @Override
    public void doIt() { }
}
```

Option 5 is wrong. Two problems here:

- The `private` modifier may not be used together with `abstract` keyword.
- The `abstract` method may not have body with curly braces.

This makes `I2` interface invalid and option 5 wrong.

*Source:*

JLS 9. Chapter 9. Interfaces [https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html]

**Question 030102**

Given the `I1` interface:

```
interface I1 {
    public static void doIt() {
        // ...
    }
}
```

Which statement is correct about `doIt()` method?

*Options (choose 1):*

1. The method can be overriden in the class which implements `I1` interface.
2. The method can be shadowed in the class which implements `I1` interface.
3. The method can only be invoked from `public static` or `private static` methods of `I1` interface.
4. The method can invoke `public static` or `private static` methods of `I1` interface.

*Answer:*

The correct option is 4.

Option 1 is wrong. In Java classes `static` methods in general are not inherited and thus may not be overriden. And for interfaces it is even more strict: the `static` method stays in the scope of the interface and not visible in scope of the implementing class.

Option 2 is wrong. When you implement an interface that contains a `static` method, the `static` method is still part of the interface and not part of the implementing class. You cannot shadow in class something which is not part of the class.

Assume `class C1 implements I1 {}`, you cannot prefix the `doIt()` method with the `C1` class name. Instead, you must always prefix the `doIt()` method with the `I1` interface name (unless you call the `doIt()` from inside the `I1` interface).

Option 3 is wrong. The `static doIt()` method of the interface can be invoked from any non-abstract method of the `I1` interaface (or from other Java class/interface by using syntax: `I1.doIt();`).

Option 4 is correct. The `static doIt()` method can invoke `static` methods of `I1` interface, both `public` and `private`. Moreover, if you need to invoke some `private` non-static method of an interface, the only way to reach it from outside code is to invoke via `default public` method in the same interface.

*Source:*

JLS 9. Chapter 9. Interfaces [https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html]

**Question 030103**

Which four method declarations are allowed in Java 11 interfaces?

*Options (choose 4):*

1. `public abstract void doIt();`
2. `private static void doIt() {}`
3. `private abstract void doIt();`
4. `public void doIt();`
5. `protected void doIt() {}`
6. `private void doIt() {}`

*Answer:*

The correct options are 1, 2, 4, and 6.

This is the summary of methods allowed in Java 11 interfaces:

- `public abstract`, allowed since Java 1.0. The `public` and `abstract` keywords can be omited and will be added by compiler.
- `default public`, allowed since Java 8. It's a non-abstract method. The `public` modifier can be omited and will be added by compiler.
- `public static`, allowed since Java 8. It's a non-abstract method. The `public` modifier can be omited and will be added by compiler.
- `private`, allowed since Java 9. It's a non-abstract method. Visible only within the interface.
- `private static`, allowed since Java 9. It's a non-abstract method. Visible only within the interface.

Option 1 is correct. The `public abstract` methods were the only declarations allowed in Java 7 and earlier:

```
interface I1 {
    public abstract void doIt();
}
```

The same definition:

```
interface I1 {
    void doIt(); // implicitly public and abstract
}
```

Option 2 is correct. This type of non-abstract methods was added in Java 9:

```
interface I1 {
    private static void doIt() { }
}
```

Option 3 is wrong. The `private` modifier MAY NOT be used in combination with `abstract` keyword. The former assumes that memeber is not accessible outside, and the latter assumes that memeber is implemented in other class. These are mutually exclusive.

Option 4 is correct. This is the interface demonstrating the metod declaration:

```
interface I1 {
    public void doIt();
}
```

The full form of the same interface will be:

```
interface I1 {
    public abstract void doIt();
}
```

Option 5 is wrong. Interfaces may not declare `protected` methods (both abstract or non-abstract).

Option 6 is correct. Interfaces may declare `private` non-abstract methods as of Java 9:

---

```
interface I1 {
    private void doIt() { }
}
```

*Source:*

JLS 9. Chapter 9. Interfaces [https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html]

**Question 030104**

Given the code:

```
class Calc implements Adding {
    public static void main(String[] args) {
        System.out.print(new Calc().sumOfTwo(1, 1));
    }
}
interface Adding {
    public Integer sumOfTwo(Integer a, Integer b) {
        return sum(a, b);
    }

    private Integer sum(Integer... ints) {
        return Arrays.asList(ints).stream().mapToInt(a->a).sum();
    }
}
```

What is the result?

*Options (choose 1):*

1. `2`
2. Compilation fails due to `Calc.main(...)` method
3. Compilation fails due to `Adding.sumOfTwo(...)` method
4. Compilation fails due to `Adding.sum(...)` method

*Answer:*

The correct option is 3.

The `public Integer sumOfTwo(Integer a, Integer b) { ... }` declaration is invalid for interfaces.

If we want to declare non-abstract method (as in example), then we must use `default` keyword. It means that we have a method with implementation and if implementing class does not provide own implementation, the "default" implementation from interface will be used and we do not need to declare class `abstract`. This approach allows adding new methods to interfaces without impacting classes which already implement this interface. Default methods were added in Java 8.

In order to make the code compile and run we need to change method signature as follows: **`default`** `public Integer sumOfTwo(Integer a, Integer b)`. In this case option 1 would be correct.

If we talk about original method signature, it would be correct for `abstract` method. In that case interface `Adding` would compile, but `Calc` class would fail, because `abstract` method not implemented.

The following signatures are syntactically identical and do not allow curly braces after method declaration:

```
public Integer sumOfTwo(Integer a, Integer b);

// The same:
public abstract Integer sumOfTwo(Integer a, Integer b);
abstract Integer sumOfTwo(Integer a, Integer b);
Integer sumOfTwo(Integer a, Integer b);
```

*Source:*

JLS 9. Chapter 9. Interfaces [https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html]

**Question 030105**

Given the code:

```
interface Greeter {
    default public String say() {
        return "Hello";
    }
}
interface Welcomer {
    default public String say() {
        return "Welcome";
    }
}
class Speaker implements Greeter, Welcomer {
    public static void main(String[] args) {
        System.out.println(new Speaker().say());
    }
}
```

What is the result?

*Options (choose 1):*

1. Code successfully compiles and prints `Hello`.
2. Compilation fails, in order to print `Hello` new method must be added to the `Speaker` class:

```
public String say() {
    return Greeter.say();
}
```

3. Compilation fails, in order to print `Hello` new method must be added to the `Speaker` class:

```
public String say() {
    return Greeter.this.say();
}
```

4. Compilation fails, in order to print `Hello` new method must be added to the `Speaker` class:

```
public String say() {
    return Greeter.super.say();
}
```

*Answer:*

The correct option is 4.

If a class implements two interfaces, both of which have a `default` method with the same name and parameter types, then you must resolve the conflict. This makes option 1 wrong.

The `Speaker` class inherits two `say()` methods provided by the `Greeter` and `Welcomer` interfaces. There is no way for the Java compiler to choose one over the other. The compiler reports an error and leaves it up to you to resolve the ambiguity. You must provide a `say()` method in the `Speaker` class and either implement your own logic, or delegate to one of the conflicting methods, like this:

```
```

```
public String say() {
    return Greeter.super.say();
}
```

The `super` keyword lets you call a supertype method. In this case, we need to specify which supertype we want.

If one interface provides an implementation and another interface declares same method as `abstract`, the compiler reports a conflict too, and it is up to the programmer to resolve the ambiguity.

If neither interface provides a `default` for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented and declare the class as `abstract`.

If a class extends a superclass and implements an interface, inheriting the same method from both, in that case, only the superclass method matters, and any `default` method from the interface is simply ignored.

This makes option 4 correct.

Option 2 has an error: invoking an instance (`default`) method in a static style (directly on the type).

Option 3 uses wrong syntax. We need to specify `super` keyword, as we delegate to superclass' implementation.

*Source:*

JLS 9. Chapter 9. Interfaces [https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html]

## 3.2. Define and write functional interfaces

**Question 030201**

Given the two interfaces, which statement is correct:

```
@FunctionalInterface
public interface ParentIntf {
    void doIt();
}
```

```
@FunctionalInterface
public interface ChildIntf extends ParentIntf {

}
```

*Options (select 1):*

1. Compilation of `ChildIntf` fails. To make the code compile sucessfully, remove `@FunctionalInterface` from `ChildIntf` declaration.
2. Compilation of `ChildIntf` fails. To make the code compile sucessfully, add in `ChildIntf` interface:

   ```
   @FunctionalInterface
   public interface ChildIntf extends ParentIntf {
       void doIt();
   }
   ```

3. Compilation of `ChildIntf` fails. To make the code compile sucessfully, add in `ChildIntf` interface:

   ```
   @FunctionalInterface
   public interface ChildIntf extends ParentIntf {
       void doItNow();
   }
   ```

4. The code compiles successfully.

*Answer:*

The correct option is 4.

The `ChildIntf` is still valid functional interface. It inherits the parent's abstract `doIt();` method (SAM).

A functional interface is an interface that has just one abstract method (aside from the methods of `Object`), and thus represents a single function contract. This "single" method may take the form of multiple abstract methods with override-equivalent signatures inherited from superinterfaces; in this case, the inherited methods logically represent a single method.

Option 2 is wrong. The code compiles successfully. If you add new method with signature `void doIt();` the code remains valid.

*Source:*

JLS 11, Section 9.8. Functional Interfaces [https://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8]

**Question 030202**

Given the two interfaces, which three statements are correct:

```
@FunctionalInterface
public interface ParentIntf {
        void doIt();
}
```

```
@FunctionalInterface
public interface ChildIntf extends ParentIntf {
        void doItNow();
}
```

*Options (select 3):*

1. Compilation fails. To make the code compile sucessfully remove `@FunctionalInterface` annotation from the `ChildIntf` interface:

   ```
   public interface ChildIntf extends ParentIntf {
       void doItNow();
   }
   ```

2. Compilation fails. To make the code compile sucessfully, modify the `ChildIntf` interface method signature:

   ```
   @FunctionalInterface
   public interface ChildIntf extends ParentIntf {
       void doIt();
   }
   ```

3. Compilation fails. To make the code compile sucessfully remove `@FunctionalInterface` annotation from the `ParentIntf` interface:

   ```
   public interface ParentIntf {
       void doIt();
   }
   ```

4. Compilation fails. To make the code compile sucessfully, modify the `ChildIntf` declaration:

   ```
   @FunctionalInterface
   public interface ChildIntf {
   ```

```
    void doItNow();
}
```

*Answer:*

The correct options are 1, 2, and 4.

Option 1 correct. The `@FunctionalInterface` annotation is analyzed by Java compiler. The `ChildIntf` is invalid functional interface, as has 2 abstract methods: one own and one inherited. After annotation is removed, compiler does not complain (but the `ChildIntf` remains invalid functional interface.)

Option 2 correct. The `ChildIntf` will have a single abstract method (no matter that it's already declared in parent interface.) This code also would be valid:

```
@FunctionalInterface
public interface ParentIntf {
    void doIt();
}
```

```
@FunctionalInterface
public interface ChildIntf extends ParentIntf {
        @Override
        void doIt();
}
```

Option 3 is wrong, as child interface still has 2 abstract methods and can not comply it's annotation.

Option 4 is correct, as `ChildIntf` interface does not inherit any methods, and has only own single abstract method (SAM).

A functional interface is an interface that has just one abstract method (aside from the methods of `Object`), and thus represents a single function contract. This "single" method may take the form of multiple abstract methods with override-equivalent signatures inherited from superinterfaces; in this case, the inherited methods logically represent a single method.

*Source:*

JLS 11. Section 9.8. Functional Interfaces [http://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8]

**Question 030203**

Which two statements are correct about Java functional interfaces?

*Options (select 2):*

1. A functional interface must be annotated with `@FunctionalInterface`.
2. A functional interface may contain one or more `abstract` methods.
3. A functional interface may contain one or more `default` methods.
4. A functional interface may contain one or more `static` methods.

*Answer:*

The correct options are 3 and 4.

Option 1 is wrong. The annotation type `@FunctionalInterface` is used to indicate that an interface is meant to be a functional interface. It facilitates early detection of inappropriate method declarations appearing in or inherited by an interface that is meant to be functional.

It is a compile-time error if an interface declaration is annotated with `@FunctionalInterface` but is not, in fact, a functional interface.

Because some interfaces are functional incidentally, it is NOT necessary or desirable that all declarations of functional interfaces be annotated with `@FunctionalInterface`.

Option 2 is wrong. A functional interface is any interface that contains **only one abstract method (a.k.a. Single Abstract**

**Method - SAM)**.

A functional interface may contain one or more `default` methods or `static` methods.

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

JLS 11. Section 9.8. Functional Interfaces [http://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8]

**Question 030204**

Given the code:

```
interface CPU1 {
    int doAdd(); // line 1
    default int doSub() { return 1 - 1; }
}

interface CPU2 {
    default int doAdd() {return 1 + 1;} // line 2
    String toString();
}

public class Calculator {

    CPU1 c1 = () -> { return 1 + 2; }; // line 3
    CPU2 c2 = () -> { return 1 + 3; }; // line 4

    void calculate() {
        System.out.print(c1.doAdd());
        System.out.print(c2.doAdd());
    }

    public static void main(String[] args) {
        new Calculator().calculate();
    }
}
```

What is the result?

*Options (select 1):*

1. Compilation fails at `line 1`.
2. Compilation fails at `line 2`.
3. Compilation fails at `line 3`.
4. Compilation fails at `line 4`.
5. The code prints `32`.
6. The code prints `34`.

*Answer:*

The correct option is 4. The code will fail at the line:

```
CPU2 c2 = () -> { return 1 + 3; }; // line 4
```

with error "no abstract method found in interface CPU2". You may notice that it's `doAdd` method is `default` and has implementation (i.e. not `abstract`). And the `toString` method is not considered as SAM, as it is already a `public` member of `Object` class.


The definition of functional interface excludes methods in an interface that are also `public` methods in `Object`.

Option 1 is wrong. The `CPU1` is a valid interface.

Option 2 is wrong. The `CPU2` is a valid interface. In case `@FunctionalInterface` annotation presented, the compilation would fail.

Option 3 is wrong. It's perfectly valid example of lambda usage, which prints "`3`" to screen.

Option 5 is wrong, the code won't compile, as the `CPU2` can't be used in lambda expression.

Option 6 is wrong. In case `CPU2` was a valid functional interface, the code would print `34`.

*Source:*

JLS 11. Section 9.8. Functional Interfaces [http://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8]

**Question 030205**

Given the `Event` interface:

```
interface Event {
    String toString();
}
```

and the code:

```
Event e = () -> { return "Mouse"; };
System.out.println(e);
System.out.println(e.toString());
```

What is the result?

*Options (select 1):*

1. A hashcode of `e` object, followed by `Mouse`.
2. A blank line, followed by `Mouse`.
3. 

```
Mouse
Mouse
```

4. Compilation fails.

*Answer:*

The correct option is 4. The code will fail to compile, because `Event` is not a functional interface (i.e. does not have a Single Abstract Method, a.k.a. SAM). Compilation will fail at this line:

```
Event e = () -> { return "Mouse"; };
```


The `public` methods from `java.lang.Object` class are NOT counted toward the interface's `abstract` method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere.

Option 1 might be correct if we rename `toString()` to `toString1()`.

```
interface Event {
    String toString1();
```

```
}
...
System.out.println(e);
System.out.println(e.toString1());
```

In that case output will be as follows:

```
Xxxxxyyyyyzzzz$$Lambda$1/989110044@2f2c9b19
Mouse
```

*Source:*

JLS 11. Section 9.8. Functional Interfaces [http://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8]

## Chapter 4. Lambda Operations on Streams

### 4.1. Extract stream data using `map`, `peek`, and `flatMap` methods

#### Question 040101

What two statements are true about stream pipelines in Java?

*Options (choose 2):*

1. A pipeline may have one or more sources.
2. A pipeline may consist of a source followed by a terminal operation.
3. A pipeline must have one or more intermediate operations.
4. A pipeline must have a terminal operation.

*Answer:*

The correct options are 2 and 4.

A pipeline contains the following components:

- A source. This could be a `Collection`, an array, a generator function, or an I/O channel.
- **Zero or more** intermediate operations. An intermediate operation, such as `filter(...)`, always produces a new `Stream`.

  A stream is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline.

  For example, the `filter(...)` operation returns a new `Stream` that contains elements that match its `Predicate` (this operation's parameter).
- A terminal operation. A terminal operation, such as `forEach(...)`, produces a non-stream result, such as a primitive value (like a `double` value), a `Collection`, or in the case of `forEach`, no value at all (`void`).

Option 1 is wrong: pipeline may (and must) have exactly 1 source (`java.util.List`, array, etc..)

Option 3 is wrong: pipeline may have zero intermediate operations, like shown below:

```
List<String> list = Arrays.asList("I'm", "pipeline", "too");
list.stream().forEach((String s) -> System.out.println(s)); // zero intermediate
operations
```

*Source:*

Lesson: Aggregate Operations [https://docs.oracle.com/javase/tutorial/collections/streams/]

Interface Stream JavaDoc [http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html]

**Question 040102**

Given the code:

```
List<String> list = List.of("one", "two", "three");
long l = list.stream().forEach(s -> System.out.println(s)).filter(s ->
s.startsWith("t")).count();
System.out.println(l);
```

What is the result?

*Options (choose 1):*

1.
```
two
three
2
```

2.
```
one
two
three
2
```

3.
```
one
two
three
```

   followed by Runtime Exception.
4. Compilation fails.

*Answer:*

The correct option is 4.

Some points:

- While you don't have to memorize exact return type of `forEach(...)` or `count()`, you must know that they both are terminal operations.
- A terminal operation does not return a `Stream` instance, so, it may not be followed by another intermediate operation (like `filter(...)`) which always is invoked on `Stream`, and returns a `Stream` (i.e. allows chaining).
- If you forgot the previous bulletpoint rule, another hint will be pipelines rule: it may contain only ONE terminal operation. And to answer this you must know which operations from `Stream` interface are terminal, and which are intermediate.

*Source:*

Lesson: Aggregate Operations [https://docs.oracle.com/javase/tutorial/collections/streams/]

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

**Question 040103**

Given the code:

```
List<Integer> nums = List.of(1, 2, 3);
Stream<Integer> stream = nums.stream();
stream.map(i -> i*2).forEach(i -> System.out.printf("%d ", i));
stream.forEach(i -> System.out.printf("%d ", i));
```

What is the result?

*Options (choose 1):*

1.

```
2 4 6 2 4 6
```

2.

```
2 4 6 1 2 3
```

3.

```
2 4 6
```

followed by Runtime Exception.

4. Compilation fails.

*Answer:*

The correct option is 3.

Java streams cannot be reused.


As soon as you call a terminal operation -- the stream is closed.

Calling `forEach` after `forEach` on the same stream results in the following exception:

```
2 4 6

Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
        at
java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:274)
        at java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:580)
        at ...
```

Like the existing "streams" (e.g. `InputStream`), you can consume streams only once.

A `Stream` is consumed by a terminal operation only once. These operations are terminal: `forEach`, `reduce`, `collect`, `sum`, `max`, `count`, `matchAny`, `findFirst`, `allMatch`, `noneMatch`, `findAny`, they perform an operation on the `Stream` that is their input, potentially producing an `Object` that is not a `Stream` (could be a `Boolean`, a `Map`, an `array` or a `Collection` or just a `void`.

*Source:*

Lesson: Aggregate Operations [https://docs.oracle.com/javase/tutorial/collections/streams/]

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

**Question 040104**

Given the code fragment:

```
List<String> words = List.of("One", "Two", "Three");
Stream<String> s = words.stream().map(w -> w.toLowerCase());
List<String> list = s.collect(Collectors.toList());
System.out.print(list);
```

What is the output?

*Options (choose 1):*

1. `[one]`
2. `[One]`
3. `[one, two, three]`
4. `[One, Two, Three]`
5. Blank output.

*Answer:*

The correct option is 3.

The `Stream.map(...)` is an intermediate operation which returns a stream consisting of the results of applying the given function to the elements of this stream.

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

The original stream and the resulting stream may be of different parameterized types, i.e. original stream defined with `String` parameterized type, the `Function` passed into `map(...)` may return `Integer`.

In the question's code fragment the `Function` accepts `String` and returns `String` too, but it's not a mandatory to be the same type.

The `Stream.collect(...)` is a terminal operation which performs a mutable reduction operation on all elements of this stream using a `Collector`. In our case it collects `Stream` content into new collection - `List` using `Collectors` class with multiple `static` methods which return predefined `Collector` implementations.

Oprion 1 is wrong, as `Stream.collect()` is not a short circuit terminal operation, and requires all elements processing.

Options 2 and 4 are wrong, as `map()` processes input values and returns a stream from the values it's function produces (transform to lowercase in our code).

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

`java.util.stream` package JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040105**

Given the `Employee` class:

```
public class Employee {
    private String name;
```

```
    private int age;
    public Employee(String n, int i) {
        name = n;
        age = i;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

and the code fragment:

```
Stream<Employee> emps = Stream.of(new Employee("John", 53), new Employee("Jane", 46), new
Employee("Deb", 45));
... // line n1
System.out.print(averageAge);
```

Which line inserted at `line n1` position will print average age of all employees?

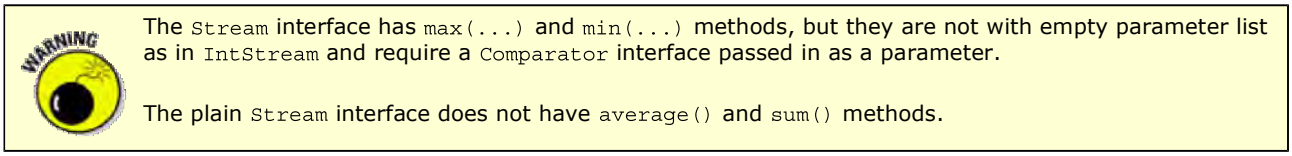
*Options (choose 1):*

1. `double averageAge = emps.map(e -> e.getAge()).average().getAsDouble();`
2. `double averageAge = emps.mapToInt(e -> e.getAge()).average().getAsDouble();`
3. `double averageAge = emps.map(e -> e).average().getAsDouble();`
4. `double averageAge = emps.mapToInt(e -> e).average().getAsDouble();`

*Answer:*

The correct option is 2.

The `average()` method (as well as `min()`, `max()`, `sum()`) exists in specific primitives-oriented Java streams: `IntStream`, `DoubleStream`, and `LongStream`.




The `map(...)` operation always returns `Stream`, which makes options 1 and 3 wrong.

The `IntStream` can be created from a `Stream` by using `Stream.mapToInt(...)` operation which applies `ToIntFunction` to all elements of the `Stream`:

```
IntStream mapToInt(ToIntFunction<? super T> mapper)
```

The `ToIntFunction` is a functional interface which accepts an `Object` (in our case - `Employee`) and returns an `int` primitive (in our case - `age`), the sample code shown in option 2:

```
e -> e.getAge()
```

The function `(e -> e)` returns the same `Employee` object which passed in as parameter, it may not be used with `mapToInt(...)` operation, which makes option 4 wrong.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

`java.util.stream` package JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040106**

Given the code fragment and assuming that `salaries` is a properly populated list of `Double` values:

```
List<Double> salaries = ... // properly populated list
... // line n1
System.out.print(minSalary);
```

Which two code fragments when inserted independently at `line n1` position will print the lowest salary?

*Options (choose 2):*

1. `double minSalary = salaries.stream().mapToDouble(Double::doubleValue).min().getAsDouble();`
2. `double minSalary = salaries.stream().mapToDouble((double d) -> d).min().getAsDouble();`
3. `double minSalary = salaries.stream().mapToDouble(d -> d).min().getAsDouble();`
4. `double minSalary = salaries.stream().map(d -> (double)d).min().getAsDouble();`
5. `double minSalary = salaries.stream().map(d -> d).min().getAsDouble();`

*Answer:*

The correct options are 1 and 3.

The `min()` function is designed to work with primitives and it exist in primivies-oriented streams (the `DoubleStream` in our example):

```
OptionalDouble min()
```

it returns an `OptionalDouble` describing the minimum element of this stream, or an empty `OptionalDouble` if this stream is empty.


The `Stream` interface has `max(...)` and `min(...)` methods, but they are not with empty parameter list as in `DoubleStream` and require a `Comparator` interface passed in as a parameter.

The `map(...)` always returns a `Stream` (which contains some non-primitive values), so options 4 and 5 are wrong. These options will not compile.

The `Stream.mapToDouble(...)` returns `DoubleStream` which contains double primitives. The parameter of `mapToDouble` is `ToDoubleFunction`:

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

The `ToDoubleFunction` accepts any object (`Double` wrapper in our case) and returns `double` primitive:

```
@FunctionalInterface
public interface ToDoubleFunction<T> {

    double applyAsDouble(T value);
}
```

Option 1 is correct as it shows a valid `ToDoubleFunction`: the method reference invokes `Double.doubleValue()` which returns a `double` primitive.

Option 2 is wrong: it tries to define explicitly type for input parameter as `double`, but input parameter must be `Object` (`Double`), so it's invalid `ToDoubleFunction` implementation, and code will not compile.

Option 3 is correct: `mapToDouble` receives a `Double` and returns the same `Double`, but Java implicitly does auto-unboxing and function returns a `double` primitive, similar code is shown below:

```
// auto unboxing example
public double getAsPrimitive(Double d) {
    return d;
}
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface DoubleStream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/DoubleStream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040107**

Given the `Employee` class:

```
public class Employee {
    public static final int MANAGER=100;
    public int type;
    public String name;
    public double salary;

    public Employee(int t, String n, double s) {
        type = t;
        name = n;
        salary = s;
    }
}
```

Assuming that `emps` variable refers to properly declared and initialized `List` of `Employee`s, which line of code will help to find manager's highest salary?

*Options (choose 1):*

1. `double maxSalary = emps.filter(a -> a.type == Employee.MANAGER).max().getAsDouble();`
2. `double maxSalary = emps.filter(a -> a.type == Employee.MANAGER).map(a -> a.salary).max().getAsDouble();`
3. `double maxSalary = emps.filter(a -> a.type == Employee.MANAGER).mapToDouble(a -> a.salary).max().getAsDouble();`
4. `double maxSalary = emps.filter(a -> a.type == Employee.MANAGER).mapToDouble(a -> a).max().getAsDouble();`

*Answer:*

The correct option is 3.

The `filter(a -> a.type == Employee.MANAGER)` first filters only managers, and then returns new `Stream` of `Employee`s.

As the `Stream` does not have `max()` method, option 1 is wrong, it just won't compile.

The `Stream` interface has `max(...)` and `min(...)` methods, but they are not with empty parameter list as in `DoubleStream` and require a `Comparator` interface passed in as a parameter.

Option 2 is wrong: the `map(...)` operation will map `Stream` of `Employee`s to `Stream` of `Double`s, and since `Stream` does not have `max()` method, this line will fail to compile too.

Option 4 is wrong, as it has invalid `ToDoubleFunction` implementation passed into `mapToDouble`.

The `Stream.mapToDouble(...)` returns `DoubleStream` which contains `double` primitives. The parameter of `mapToDouble` is `ToDoubleFunction`:

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

The `ToDoubleFunction` accepts any `Object` (`Employee` instance in our case) and returns `double` primitive:

```
@FunctionalInterface
public interface ToDoubleFunction<T> {

    double applyAsDouble(T value);
}
```

Option 4 implementation - `(a -> a)` - accepts `Employee` and returns the same `Employee`, which will cause the code compilation fails.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface DoubleStream JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/DoubleStream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040108**

Given the code fragment:

```
List<String> names1 = List.of("Dzmitry", "John");
List<String> names2 = List.of("Mikalai");
List<String> names3 = List.of("David", "Laura");
Stream<List<String>> s = Stream.of(names1, names2, names3);
s.flatMap(names -> names.stream()).forEach(System.out::println);
```

What is the output?

*Options (choose 1):*

1.

```
Dzmitry
John
```

2.

```
David
Laura
```

3.

```
Dzmitry
John
Mikalai
David
Laura
```

4.

```
David
Dzmitry
John
Laura
Mikalai
```

*Answer:*

The correct option is 3. The output is:

```
Dzmitry
John
Mikalai
David
Laura
```

The `flatMap` transforms each element of the `s` stream into a stream of `String` objects. So each `List<String>` object will be transformed into one or two `String` objects backed by streams. The contents of those streams will then be placed into the returned stream of the `flatMap` operation. And `forEach` on the returned stream will print all `String` values as they placed in backed lists.

Option 4 is wrong. The option might be correct if `sorted()` operation was applied to the resulting stream:

```
...
s.flatMap(names -> names.stream()).sorted().forEach(System.out::println);
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

### 4.2. Search stream data using search `findFirst`, `findAny`, `anyMatch`, `allMatch` and `noneMatch` methods

**Question 040201**

Given the code fragment:

```
Stream<String> txt = Stream.of("Once", "One", "Other");
System.out.print(txt.filter(w -> w.startsWith("O")).findAny().get());
```

What is the output?

*Options (choose 1):*

1. `Once`
2. `OnceOneOther`
3. Either `Once`, or `One`, or `Other`, chosen randomly.
4. All three: `Once`, and `One`, and `Other` in no particular order.

*Answer:*

The correct option is 3.

The `findAny` is a terminal operation, as soon as it is invoked, the pipeline starts processing.

Stream can be "with defined encounter order" (e.g. based on `java.util.List` collection) or "with no encounter order" (e.g. based on `java.util.Set` collection).

For these both types of stream the `findAny` may return any element (i.e. `Once`, or `One`, or `Other`).

NOTE: although `findAny` in this particular code will most likely return `Once`, don't be confused by this behavior -- any element of the stream may be returned.

Sidenote: the `Stream.of(...)` factory method returns a stream with defined encounter order:

```
Stream<String> s = Stream.of("1", "2", "3");
Spliterator<String> sp = s.spliterator();
boolean isOrdered = sp.hasCharacteristics(Spliterator.ORDERED);
System.out.println(isOrdered); // prints 'true'
```

Option 2 is wrong: the `findAny` is a short-circuit operation, it breaks pipeline execution as soon as any element from stream is obtained.

Option 1 is wrong, this option might be true if we used `findFirst()` terminal operation. It works as follows:

- For defined encounter order streams returns the first element in encounter order in stream (i.e. `Once`)
- For no encounter order stream it returns any element.

Option 4 is wrong: the `findAny` is a short-circuit terminal operation.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

`java.util.stream` package JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040202**

Given the code fragment:

```
Stream<String> txt = Stream.of("Code", "And", "Code");
txt.allMatch(w -> w.equals("Code")).forEach(System.out::print); // line n1
```

What is the result?

*Options (choose 1):*

1. `CodeAndCode`
2. `CodeCode`

3. `Code`
4. Compilation fails at `line n1`.

*Answer:*

The correct option is 4.

The `allMatch(...)` is a terminal operation, it may not be followed by another terminal operation (`forEach(...)`).

The `anyMatch(...)`, `allMatch(...)`, `noneMatch(...)`, `findFirst()`, and `findAny()` all are **terminal operations**.

Option 2 is wrong. This option would be correct for the following code:

```
Stream<String> txt = Stream.of("Code", "And", "Code");
txt.filter(w -> w.equals("Code")).forEach(System.out::print);
```

*Source:*

Interface Stream JavaDoc [http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040203**

Given the `Employee` class:

```
public class Employee {
    public static final int MANAGER=100;
    public int type;
    public String name;

    public Employee(int t, String n) {
        type = t;
        name = n;
    }
}
```

and the following code fragment:

```
List<Employee> emps = List.of(new Employee(100, "Mike"), new Employee(99, "John"), new
Employee(100, "Deb"));
boolean b1 = emps.stream().map(e -> e.type).allMatch(t -> t == Employee.MANAGER);
boolean b2 = emps.stream().filter(e -> !(e.type == Employee.MANAGER)).anyMatch(m -> m.type
== Employee.MANAGER); // line n1
System.out.print(b1 + " " + b2);
```

What is the result?

*Options (choose 1):*

1. `true true`
2. `true false`

3. `false true`
4. `false false`
5. Compilation fails at `line n1`.

*Answer:*

The correct option is 4.

The output will be `false false`.

The `boolean b1 = ...` line will first map a `Stream` of `Employee`s to a `Stream` of `Integer`s with content: `[100, 99, 100]`. NOTE: Autoboxing will handle implicit `int` transforming to `Integer`, so there is no code compilation error. Then terminal operation `allMatch` will check if all elements of the new `Stream` are equal to `100` (by using unboxing), and it will give `false` result.


The `anyMatch(...)`, `allMatch(...)`, `noneMatch(...)` all are terminal operations which accept `Predicate` and return `boolean`.

The `boolean b2 = ...` line will first filter `Stream` of `Employee`s to remove those, whose `type` is equals to `100`. New `Stream` will contain only 1 `Employee` with `type=99`. Next, terminal operation `anyMatch` will check if any `Employee` from the new `Stream` has `type` equal to `100`, and it will give `false` result.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040204**

Given the following code fragment:

```
List<String> list = List.of("A", "B", "C", "A", "A", "B");
... // line n1
System.out.print(v);
```

What line inserted at `line n1` position will print `true`?

*Options (choose 1):*

1. `boolean v = list.stream().allMatch(a -> "A".equals(a));`
2. `boolean v = list.stream().noneMatch(a -> "A".equals(a));`
3. `boolean v = list.stream().map(a -> "A").noneMatch(a -> "A".equals(a));`
4. `boolean v = list.stream().filter(a -> "A".equals(a)).allMatch(a -> "A".equals(a));`
5. `boolean v = list.stream().filter(a -> "A".equals(a)).noneMatch(a -> "A".equals(a));`

*Answer:*

The correct option is 4.

Option 1 is wrong: it tests if all `String`s from original `Stream` equal to "`A`". Since there are other elements in `Stream`, like "`B`" and "`C`", it gives the `false` result.

Option 2 is wrong: it tests if none `String`s from original `Stream` equal to "A". Since there are elements in `Stream` with "`A`" value, it gives the `false` result.

Option 3 is wrong: it simply maps (replaces) all values from the original `Stream` "`A`" value. Then it tests if none `String`s from original `Stream` equal to "`A`", it gives the `false` result, as all elements are equal to "`A`".

Option 4 is correct: first it filters the stream and leaves only "`A`" values, then checks if all elements are equal to "`A`", which

gives obviously `true`.

Option 5 is wrong: first it filters the stream and leaves only "`A`" values, then checks if none of elements are equal to "`A`", which gives `false` because all elements are equal to "`A`".

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040205**

Given the following code fragment:

```
List<Integer> ints = List.of(10, 20, 30);
Optional<Integer> o = ints.stream().anyMatch(i -> i > 15); // line n1
System.out.print(o.get());
```

What is the result?

*Options (choose 1):*

1. `20`
2. `30`
3. Either `20`, or `30`, randomly chosen.
4. Compilation fails at `line n1`.

*Answer:*

The correct option is 4.

The `anyMatch(i -> i > 15)` returns `boolean`, not `Optional`. The code just will not compile.


- The `anyMatch(...)`, `allMatch(...)`, `noneMatch(...)`, `findFirst()`, and `findAny()` methods are **short-circuit terminal operations**.
- All `findXxx()` methods have no arguments and return `Optional`.
- All `xxxMatch(...)` methods accept a `Predicate` and return a `boolean` primitive.

Option 1 is wrong, it could be correct with the following code:

```
List<Integer> ints = Arrays.asList(new Integer(10), new Integer(20), new Integer(30));
Optional<Integer> o = ints.stream().filter(i -> i > 15).findFirst();
System.out.println(o.get());
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

java.util.stream package JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

**Question 040206**

Given the code fragment:

```
```

```
Integer i1 = Stream.of(1,2).findAny().get();
Integer i2 = Stream.of(1,2).parallel().findAny().get();
```

What statement is correct about the assignments?

*Options (choose 1):*

1. `i1` is always `1`, and `i2` is always `1`
2. `i1` is always `1`, and `i2` can be `1` or `2`
3. `i1` is always `1`, and `i2` is always `2`
4. Both `i1` and `i2` can be `1` or `2`

*Answer:*

The correct option is 4.


`Optional<T> findAny()` returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty.

This is a short-circuiting terminal operation.

The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result.

Because the `Stream.findAny()` behavior is nondeterministic it may choose arbitrary element of the `Stream` no matter if it is sequential and parallel. This makes option 1, 2, and 3 wrong, and option 4 correct.

NOTE: when running the code you may repeatedly get the same result, like `i1` is `1` and `i2` is `2`, but you should know what documentation says and answer on exam based on the JavaDoc documentation. You must not rely on what the code produces on your particular JVM.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

`java.util.stream` package JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html]

## 4.3. Use the `Optional` class

**Question 040301**

Given the following code fragment:

```
1 Integer i = null;
2 Optional<Integer> o = Optional.of(i);
3 System.out.println(o.isPresent());
4 Integer ii = o.get();
5 System.out.println(ii.intValue());
```

What is the result?

*Options (choose 1):*

1. Runtime exception at line 2.
2. Runtime exception at line 4.
3. Runtime exception at line 5.
4. Compilation fails.

*Answer:*

The correct option is 1.

The `Optional.of(T value)` method returns an `Optional` with the specified present non-null `value`. It throws `NullPointerException` if `value` is `null`:

```
Exception in thread "main" java.lang.NullPointerException
        at java.util.Objects.requireNonNull(Objects.java:203)
        at java.util.Optional.<init>(Optional.java:96)
        at java.util.Optional.of(Optional.java:108)
        ...
```

Option 2 is wrong. It might be correct if we used `Optional.ofNullable(T value)` instead of `Optional.of(T value)`:

```
  1 Integer i = null;
  2 Optional<Integer> o = Optional.ofNullable(i);
  3 System.out.println(o.isPresent());
  4 Integer ii = o.get();
  5 System.out.println(ii.intValue());
```

Result:

```
Exception in thread "main" java.util.NoSuchElementException: No value present
        at java.util.Optional.get(Optional.java:135)
        ...
```

*Source:*

Class Optional JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html]

**Question 040302**

Given the following code fragment:

```
String s1 = "ABC";
String s2 = null;
Optional<String> o1 = Optional.ofNullable(s1);
Optional<String> o2 = Optional.ofNullable(s2);
System.out.println(o1.isPresent());
System.out.println(o2.isPresent());
System.out.println(o1.get());
System.out.println(o2.get());
```

What is the result?

*Options (choose 1):*

1.

```
true
true
ABC
null
```

2.

```
true
false
ABC
```

```
null
```

3.

```
true
true
ABC
```

Followed by runtime exception.

4.

```
true
false
ABC
```

Followed by runtime exception.

5.

```
true
```

Followed by runtime exception.

*Answer:*

The correct option is 4. Result of this code is:

```
true
false
ABC
Exception in thread "main" java.util.NoSuchElementException: No value present
        at java.util.Optional.get(Optional.java:135)
        at ...
```

The `Optional.ofNullable(T value)` can accept both object and `null` values. The `ofNullable` method is intended as a bridge from the use of `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null`, and `Optional.empty()` otherwise.

So, after initialization, `o1` refers to `Optional` with value, and `o2` refers to empty `Optional`.

Next, the `isPresent` method invoked. It returns `true` if there is a value present; otherwise `false`. The output will be `true` for `o1`, and `false` for `o2`.

And finally, code attempts to get and to print values from `Optional`s. For `o1` it prints `ABC`, and for empty `o2` it throws `NoSuchElementException`. Here is the source and JavaDoc of the `get()` method:

```
/**
 * If a value is present in this {@code Optional}, returns the value,
 * otherwise throws {@code NoSuchElementException}.
 *
 * @return the non-null value held by this {@code Optional}
 * @throws NoSuchElementException if there is no value present
 *
 * @see Optional#isPresent()
 */
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
```

```
}
```

Options 1 and 3 are wrong: the `isPresent()` for empty optional returns `false`.

Option 5 is wrong. It might be correct if we used `Optional.of(...)` instead of `Optional.ofNullable(...)`.

*Source:*

Class Optional JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html]

**Question 040303**

Given the code:

```
var o = Optional.ofNullable(1);
o = o.or(() -> Optional.ofNullable(2));
var i = o.orElseGet(() -> 3);
System.out.print(i);
```

What is the output?

*Options (choose 1):*

1. `1`
2. `2`
3. `3`
4. Blank output.

*Answer:*

The correct option is 1. Result of this code is: `1`.

The `Optional.ofNullable(T value)` can accept both object and `null` values. The `ofNullable` method is intended as a bridge from the use of `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null`, and `Optional.empty()` otherwise.

So, after the line runs:

```
var o = Optional.ofNullable(1);
```

the `o` refers to `Optional<Integer>` which wraps `Integer(1)`.

Next, the `Optional.or(...)` method invoked. This method was added in Java 9 to give ability supply another `Optional<T>` if the original `Optional` is empty. Since `o` is not empty, the same value is returned and the supplier lambda not invoked.

And finally, `Optional.orElseGet(...)` method invoked. This method returns wrapped value for non-empty `Optional` or invoke `Supplier<T>` returning new parameterized value. Since `o` is not empty, the supplier part is not invoked and the method returns value wrapped by the optional optional -- `Integer(1)`. So, the code prints `1`.

Options 2 is wrong: the oputput `2` could be possible with this code:

```
var o = Optional.ofNullable(null);
o = o.or(() -> Optional.ofNullable(2));
...
```

Option 3 is wrong. It might be correct with the following code:

```
var o = Optional.ofNullable(null);
o = o.or(() -> Optional.ofNullable(null));
var i = o.orElseGet(()-> 3);
...
```

Option 4 is wrong. The `i` variable is always initialized by some `Integer` and can not be blank.

*Source:*

Class Optional JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html]

### 4.4. Perform calculations using `count`, `max`, `min`, `average` and `sum` stream operations

**Question 040401**

Given the following code fragment:

```
List<Integer> ints = List.of(2, 4, 6, 5, 8);
Stream<Integer> stream = ints.stream();
... // line n1
```

Which code inserted at `line n1` position will calculate average value of numbers from the list?

*Options (choose 1):*

1. `Optional<Double> average = stream.average().get();`
2. `Double average = stream.average().get();`
3. `double average = stream.map(d -> d).average().getAsDouble();`
4. `double average = stream.mapToInt(d -> d).average().getAsDouble();`

*Answer:*

The correct option is 4.

The `average()` method exists only in primitives-oriented streams: `IntStream`, `LongStream`, and `DoubleStream` which store primitive values directly, without using wrappers.

The `stream` variable refers to `Stream` of objects, so options 1 and 2 are wrong.

The `Stream.map(...)` operation returns `Stream` of objects, so option 3 is wrong.

There is additional hint for you: primitives-oriented stream can be created from stream of objects using `mapToXxxx` operations. When you have a stream of objects (`Integer`s in our case), you can transform it to a primitive type stream with the `Stream.mapToInt`, `Stream.mapToLong`, or `Stream.mapToDouble` methods.

NOTE: the `average()` method returns `OptionalDouble`, and to retrieve `double` you need to invoke `OptionalDouble.getAsDouble()` method.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface IntStream JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html]

Class OptionalDouble JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalDouble.html]

**Question 040402**

Given the following code fragment:

```
List<Integer> ints = List.of(2, 4, 6, -1, 5, 8);
Stream<Integer> stream = ints.stream();
... // line n1
```

Which two code fragments inserted independently at `line n1` position will print the minimum value contained by the list?

*Options (choose 2):*

1. `System.out.print(stream.min(Integer::min).get());`
2. `System.out.print(stream.min(Integer::max).get());`
3. `System.out.print(stream.min(Integer::compare).get());`
4. `System.out.print(stream.map(i -> i).min().get());`
5. `System.out.print(stream.mapToInt(i -> i).min().getAsInt());`

*Answer:*

The correct options are 3 and 5.

Option 1 is wrong. The code will actually compile successfully, because `Stream.min(...)` accepts an instance of `Comparator` so that items in the stream can be compared against each other to find the minimum. And `Integer::min` is accepted because any interface with one abstract method can be automatically implemented by any lambda -- or method reference -- whose method signature is a match for the one method on the interface (`Comparator` in our case)). So examining the `Comparator` interface:

```
public Comparator<T> {
    int compare(T v1, T v2);
}
```

If a `Stream.min(...)` method is looking for a `Comparator<Integer>`, then it's essentially looking for this signature:

```
int xxx(Integer o1, Integer o2);
```

NOTE: method name is NOT used for matching purposes.

Therefore, `Integer.min(int a, int b)` is close enough that autoboxing will allow this to appear as a `Comparator<Integer>` in the `min(...)` method context.

Option 1 provides `Integer.min(int, int)` as an implementation of `Comparator.compare(Integer, Integer)`. That method does not match the requirements of `Comparator.compare`. Instead of returning a value indicating which parameter is bigger, it returns the minumum value from two (and almost always will be positive the list of ints in the code, so first parameter almost always will be considered bigger).

Option 2 is wrong for the same reason as option 1. The code will compile and run, but produce a wrong result.


The mechanism that allows such code compile is called **target typing**.

The idea is that the type the compiler assigns to a lambda expressions or a method references does not depend only on the expression itself, but also on where it is used.

The target of an expression is the variable to which its result is assigned or the parameter to which its result is passed.

Lambda expressions and method references are assigned a type which matches the type of their target, if such a type can be found.

Option 3 is correct. It is a valid example of finding minimum value from the stream of objects. For this case a `Comparator` is required, and we use built-in `Comparator` from `Integer` class.

Option 4 is wrong. Method `min()` without parameters exists only in primitive-oriented streams (e.g. `IntStream`). The code will not compile.

Option 5 is correct. We first map stream of objects to stream of integers (`IntStream`), and then get the minimum value.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface IntStream JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html]

Class OptionalInt JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalInt.html]

**Question 040403**

Given the `Employee` class:

```
public class Employee {
    public String name;
    public int salary;
    public Employee(String n, int s) {
        name = n;
        salary = s;
    }
}
```

and following code fragment:

```
Stream<Employee> stream = ... // assume stream is properly initialized by valid Employee
objects
... // line n1
```

Which code inserted at `line n1` position will print the minimum salary of the employees?

*Options (select 1):*

1. `System.out.print(stream.min((a, b) -> a.salary - b.salary).get().salary);`
2. `System.out.print(stream.min((a, b) -> a.salary - b.salary).get().getAsInt());`
3. `System.out.print(stream.min(a -> a.salary).get().getAsInt());`
4. `System.out.print(stream.min(Integer::compare).get());`

*Answer:*

The correct option is 1.

When we invoke `min(...)` method on stream of objects, we must provide a `Comparator` interface implementation. The `Comparator` is a Functional Interface, so we can implement it as lambda expression `(a, b) -> a.salary - b.salary` which compares salaries. The `min(...)` method returns `Optional`, so we need to invoke `get()` next to access `Employee` object.

Option 2 is wrong. The `getAsInt` method exists in `OptionalInt` which is a wrapper for `int` primitive and returned by methods of primitives-oriented `IntStream` stream. The code will not compile.

Option 3 is wrong. The `min(...)` accepts a `Comparator`, and it's single abstract method (SAM) looks like:

```
int compare(T v1, T v2);
```

So, there must be 2 parameters in lambda expression. Another hint: `getAsInt()` method does not exist in `Optional` class. The code will not compile.

Option 4 is wrong. We can not use comparator with signature `int xxxx(Integer v1, Integer v2);` where we need comparator with signature `int xxxx(Employee v1, Employee v2);`. The code will not compile.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface IntStream JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html]

Class OptionalInt JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalInt.html]

### 4.5. Sort a collection using lambda expressions

**Question 040501**

Managers of your company want to make a bonus payment to an employee who (1) has the longest record of service and (2) has the lowest base salary.

Given the `Employee` class:

```
public class Employee {
    public String name;
    public int baseSalary;      // in USD
    public int recordOfService; // in years
    public Employee(String n, int s, int r) {
        name = n;
        baseSalary = s;
        recordOfService = r;
    }
}
```

and the code fragment:

```
Stream<Employee> emps = // assume stream is properly initialized by valid Employee objects

Comparator<Employee> byRoS = (e1, e2) -> e1.recordOfService - e2.recordOfService;
Comparator<Employee> bySal = (e1, e2) -> e1.baseSalary - e2.baseSalary;

... // line n1
```

Which code inserted at `line n1` position will help you to find the required employee?

*Options (choose 1):*

1. `Employee e = emps.sorted(byRoS).reversed().sorted(bySal).findFirst().get();`
2. `Employee e = emps.sorted(byRoS).reversed().thenComparing(bySal).findFirst().get();`
3. `Employee e = emps.sorted(byRoS.reversed().thenComparing(bySal)).findFirst().get();`
4. `Employee e = emps.sorted(byRoS.reversed()).thenComparing(bySal).findFirst().get();`

*Answer:*

The correct option is 3. This is the code which returns the required employee:

```
Employee e = emps.sorted(byRoS.reversed().thenComparing(bySal)).findFirst().get();
```

Here are the steps required for composite sorting condition:

1. Use the `byRoS` comparator to sort the original stream in ascending order. Those employee who work longest time will be in the end of stream.
2. Reverse the condition. Now, employees with longes record of service will be in the beginning of stream.

3. Add extra condition to sort by salary in ascending order using `bySal` comparator. Those employees who have the same record of service values will be ordered by base salary ascending order.

Now, the first element in the resulting stream will be the employee (1) with the longest record of service and (2) with the lowest base salary.

You are performing just a single sort using `sorted(...)` method, passing in the composite complex condition.

Option 1 and 2 are wrong. There is no `reversed()` method on the `Stream` interface. The code will not compile.

Options 2 and 4 are wrong. There is no `thenComparing` method on the `Stream` interface. The code will not compile.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface Comparator JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html]

**Question 040502**

Given the `Employee` class:

```
public class Employee {
    public String name;
    public Employee(String n) {
        name = n;
    }
    @Override
    public boolean equals(Object o) {
        return name.equals(o);
    }
    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

and the code fragment:

```
Stream<Employee> emps = Stream.of(new Employee("Steve"), new Employee("Nick"), new
Employee("Jack"));
emps.sorted().forEach(a -> System.out.print(a.name + " ")); // line n1
```

What is the result?

*Options (choose 1):*

1. `Steve Nick Jack`
2. `Jack Nick Steve`
3. Compilation fails at `line n1`.
4. Runtime exception.

*Answer:*

The correct option is 4.

The result of this code will be:

```
Exception in thread "main" java.lang.ClassCastException: _4_5_2.Employee cannot be cast to
java.lang.Comparable
        at java.util.Comparators$NaturalOrderComparator.compare(Comparators.java:47)
        at java.util.TimSort.countRunAndMakeAscending(TimSort.java:351)
        at java.util.TimSort.sort(TimSort.java:216)
        at java.util.Arrays.sort(Arrays.java:1507)
        at java.util.stream.SortedOps$SizedRefSortingSink.end(SortedOps.java:302)
        at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:513)
        at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:502)
        at java.util.stream.ForEachOps$ForEachOp.evaluateSequential(ForEachOps.java:150)
        at
java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateSequential(ForEachOps.java:173)
        at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
        at java.util.stream.ReferencePipeline.forEach(ReferencePipeline.java:418)
        at ...
```

The `Stream.sorted()` returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not `Comparable`, a `java.lang.ClassCastException` may be thrown when the terminal operation is executed.

Option 2 is wrong, it might be correct with the following `BetterEmployee`:

```
public class BetterEmployee implements Comparable {
    public String name;
    public BetterEmployee(String n) {
        name = n;
    }
    @Override
    public boolean equals(Object o) {
        return name.equals(o);
    }
    @Override
    public int hashCode() {
        return name.hashCode();
    }

    @Override
    public int compareTo(Object o) {
        return name.compareTo(((BetterEmployee)o).name);
    }
}
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface Comparable JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html]

**Question 040503**

Given the code:

```
List<String> list = Arrays.asList("C101", "B12", "A1", "C102");
list.stream()
    .sorted(Comparator.reverseOrder())
    .sorted((a, b) -> a.length() - b.length())
    .collect(Collectors.toList());
list.forEach(System.out::print);
```

What is the output?

*Options (choose 1):*

1. `C101B12A1C102`
2. `A1B12C102C101`
3. `C102C101B12A1`
4. `A1B12C101C102`

*Answer:*

The correct option is 1.

You already know that Java streams do not modify source which was used to create a stream. For example, a collection is not always used for stream creation, stream can be created from a generated random number sequence, or from a file (which can be on read-only filesystem).

If you look carefully, the stream pipeline performs some sortings and collects results to a new list, but the new list is not assigned to a variable. And the print statement prints original list, so the output is in the order of `list` declaration: `C101B12A1C102`.

Option 2 might be correct if we assigned the collected list to the `list` variable:

```
list = list.stream()
    .sorted(Comparator.reverseOrder())
    .sorted((a, b) -> a.length() - b.length())
    .collect(Collectors.toList());
```

First sorting is performed in reverse order of "natural sort order": `"C102", "C101", "B12", "A1"`, but then it overriden by another sorting based on new comparator by string length: `"A1", "B12", "C102", "C101"`. NOTE: `"C102"` and `"C101"` did no change position after first sorting, because comparator returns 0 (`4 - 4 = 0`) for them, considering them equal so no extra efforts done for reordering.

Option 3 is wrong, it could be correct if we did only first sorting (reversed natural sort order) and reassigned list:

```
list = list.stream()
    .sorted(Comparator.reverseOrder())
    .collect(Collectors.toList());
```

Option 4 is wrong, it could be correct if we did only second sorting (by string length ascending) and reassigned list:

```
list = list.stream()
    .sorted((a, b) -> a.length() - b.length())
    .collect(Collectors.toList());
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface Comparator JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html]

### 4.6. Use `Collectors` with streams, including the `groupingBy` and `partitioningBy` operation

**Question 040601**

Given the `Employee` class:

```
public class Employee {
    private String name;
    private String department;

    public Employee(String n, String d) {
        name = n;
        department = d;
```

```
    }

    public String getDepartment() {
        return department;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

and the following code fragment:

```
Employee e1 = new Employee("Mikalai", "Development");
Employee e2 = new Employee("Volha", "HR");
Employee e3 = new Employee("Anastasia", "Management");
Employee e4 = new Employee("Daria", "Management");
Employee e5 = new Employee("Ivan", "Management");
Stream<Employee> str = Stream.of(e1, e2, e3, e4, e5);
str.collect(Collectors.groupingBy(Employee::getDepartment)).forEach((a, b) ->
System.out.println(b)); // line n1
```

What is the result?

*Options (choose 1):*

1.
```
[Mikalai, Volha, Anastasia, Daria, Ivan]
```

2.
```
[Mikalai]
[Volha]
[Anastasia, Daria, Ivan]
```

3.
```
[Mikalai]
[Volha]
[Anastasia]
[Daria]
[Ivan]
```

4. Compilation fails at `line n1`.

*Answer:*

The correct option is 2.

The `collect(Collectors.groupingBy(Employee::getDepartment))` groups employees by department.

The function `Employee::getDepartment` is the classifier function of the grouping.

The `groupingBy` method yields a map whose values are lists, in our case: `Map<String, List<Employee>>`. Keys are department names (`String`).

Next, `Map.forEach(...)` is invoked, it iterates over each key and passes in a pair of key/value.

We invoke `System.out.println(List)`, which gives output by the following pattern: `[val1.toString(), ... , valN.toString()]`

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Class Collectors JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html]

**Question 040602**

Given the `Employee` class:

```
public class Employee {
    private String name;
    private int salary;

    public Employee(String n, int s) {
        name = n;
        salary = s;
    }

    public int getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

and the code fragment:

```
Employee e1 = new Employee("John", 98);
Employee e2 = new Employee("Rosie", 102);
Employee e3 = new Employee("Steven", 70);
Stream<Employee> str = Stream.of(e1, e2, e3);
System.out.print(str.collect(Collectors.partitioningBy(e -> e.getSalary() <
100)).get(e3.getSalary() > 100));
```

What is the result?

*Options (choose 1):*

1.

```
[John]
```

2.

```
[Rosie]
```

3.

```
[Steven]
```

4.

```
[John, Steven]
```

*Answer:*

The correct option is 2.

The `str.collect(Collectors.partitioningBy(e -> e.getSalary() < 100))` partitions all employees into two lists: one list contains those where the predicate function (`e -> e.getSalary() < 100`) returns `true`, and other list contains those elements where predicat returned `false`.

The `partitioningBy` method yields a map whose values are lists, in our case: `Map<Boolean, List<Employee>>`. Keys are predicate's values (`Boolean`).

Next, `Map.get(false)` is invoked, it returns list of employees whose salary greater or equal to `100`. There is one such employee - `Rosie`.

When we invoke `System.out.print(List)`, it gives output by the following pattern: `[element1.toString(), ... , elementN.toString()]`

Option 4 is wrong. It might be correct with the following code:

```
System.out.print(str.collect(Collectors.partitioningBy(e -> e.getSalary() <
100)).get(e3.getSalary() < 100));
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Class Collectors JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html]

**Question 040603**

Given the `Employee` class:

```
public class Employee {
    private String name;
    private String department;

    public Employee(String n, String d) {
        name = n;
        department = d;
    }

    public String getDepartment() {
        return department;
    }
}
```

and the code fragment:

```
Employee e1 = new Employee("Mikalai", "Development");
Employee e2 = new Employee("Volha", "HR");
Employee e3 = new Employee("Anastasia", "Management");
Employee e4 = new Employee("Daria", "Management");
Employee e5 = new Employee("Ivan", "Management");
Stream<Employee> str = Stream.of(e1, e2, e3, e4, e5);
... // line n1
```

```

```

Which two codes when inserted independently at `line n1` position will extract all distinct departments?

*Options (choose 2):*

1. `List l = str.distinct().collect(Collectors.toList());`
2. `Collection l = str.map(Employee::getDepartment).collect(Collectors.toSet());`
3. `List l = str.map(e -> e.getDepartment()).distinct().collect(Collectors.toList());`
4. `String l = str.distinct().map(e -> e.getDepartment()).collect(Collectors.joining(", "));`

*Answer:*

The correct options are 2 and 3.

Option 1 is wrong. Invoking `distinct()` on original stream will not extract distinct departments, but will keep all employees in stream and later collect them in list. Result of this option is a list of all employees from the original stream.

Option 2 is correct. First, it will map stream of employees to stream of department names (`String` values), and then collect the stream to a `Set`. Set collection expects that no diplicate elements (in our case - strings) may exist in it.

Option 3 is correct. First, stream of employees is mapped to stream of department names (`String` values), then `distinct()` operation will tell to remove all duplicates when terminal operation invoked. Then, terminal operation `collect` is invoked, which collects all distinct strings to a list.

Option 4 is wrong. Invoking `distinct()` on original stream will not extract distinct departments, but will keep all employees in stream and collect them in list. Then, department names are extracted and mapped to a new stream and collected to a list. Result of this option is a list of all departments (incuding duplicates) of all employees from the original stream.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Class Collectors JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html]

**Question 040604**

Given the code fragment:

```
Stream<String> str = Stream.of("Hi", "No", "Hey", "Yes", "Wow");
System.out.print(str.collect(Collectors.groupingBy(a -> a.length(),
Collectors.counting())).get(2)); // line n1
```

What is the result?

*Options (choose 1):*

1. `Hey`
2. `2`
3. `null`
4. Compilation error at `line n1`.

*Answer:*

The correct option is 2.

The `str.collect(Collectors.groupingBy(a -> a.length(), Collectors.counting())` will produce result of `Map<Integer, Long>`, where **key** is length of a word from the stream, and **value** is the number of occurences of the word of particular length in the stream.

```
length    count
  [2]   ->   [2]
  [3]   ->   [3]
```

The `map.get(2)` will retrieve value by key `Integer(2)`, and the value is `Long(2)`, so the output is `2`

Option 3 is wrong. It would be correct for non-existing key, like `map.get(4)` or `map.get(1)`.

Option 4 is wrong. Althought we use `Map.put()` and `Map.get()` with primitives, autoboxing automatically converts primitives to corresponding wrappers (i.e. `int` -> `Integer`)

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Class Collectors JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html]

## Chapter 5. Java File I/O (NIO.2)

### 5.1. Use `Path` interface to operate on file and directory paths

#### Question 050101

Given the code fragment:

```
Path p1 = Paths.get("app");
Path p2 = p1.getRoot();      // line n1
Path p3 = p2.resolve("lib"); // line n2
System.out.print(p3);
```

and assuming the following directories exist on filesystem:

```
C:\
••••app
••••lib
```

What is the result if the code run from working directory `C:\`?

*Options (choose 1):*

1. `lib`
2. `C:\lib`
3. Runtime exception at `line n1`.
4. Runtime exception at `line n2`.

*Answer:*

The correct option is 4. The `java.lang.NullPointerException` will be thrown at `line n2` position.

First off, presence of the directories and working directory before code is run does not matter. The `Path` is just hierachical sequence of directory and file name elements. It plays "logical" role.

When we invoke `Paths.get("app");` line we get relative path, even while we know that code run from `C:\`.

So, indeed the `p1` will point to existing physically on filesystem directory, but it still relative path.

Bacause the path is relative, the `p2` on `line n1` will be assigned `null`.

And on `line n2` invoking a method on `null` reference will give runtime exception.

`Path.getParent()` returns the parent path, or `null` if this path does not have a parent.

The parent of this path object consists of this path's root component, if any, and each element in the path except for the farthest from the root in the directory hierarchy. This method does not access the file system; the path or its parent may not exist. Furthermore, this method does not eliminate special names such as "." and ".." that may be used in some implementations.

Option 2 might be correct if we provide root element for path as follows: `Path p1 = Paths.get("C:\\app");`. And again, presence of real directories on the filesystem would not impact in this case too.

*Source:*

Interface Path JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Path.html]

**Question 050102**

Given the class running on Unix-like platform:

```
import java.nio.file.Path;

public class PathTest {
    public static void main(String[] args) {
        Path pm = Path.of("/home/mikalai");
        Path pv = Path.of("/home/volha");
        Path res = pm.resolve(pv);
        System.out.print(res);
    }
}
```

What is the output?

*Options (choose 1):*

1. `/home/mikalai`
2. `/home/volha`
3. `/home/volha/home/mikalai`
4. `/home/mikalai/home/volha`

*Answer:*

The correct option is 2. The output will be (on Unix-like systems) `/home/volha`.

Just a reminder: presence of the directories and working directory for the class running does not matter. The `Path` is just hierachical sequence of directory and file name elements.

The `static` factory method `Path.of(String first, String... more)` was added in Java 11 and allows to create `Path` instances, along with `Paths.get(...)`.

Let's review the `Path.resolve(...)` method rules:

- It used in the form `newPath = thisPath.resolve(otherPath);`
- If `otherPath` is **absolute** path (i.e. start with root path - `/` or `C:\`), then the expression always returns `otherPath`.
- If `otherPath` is **empty** path, then the expression always returns `thisPath`.
- If `otherPath` is **relative** path (i.e. does not have root path), and `thisPath` is **relative** (i.e. is considered a directory), then `otherPath` is appended to `thisPath` (i.e. `thisPath/otherPath`).
- If `otherPath` is **relative** path (i.e. does not have root path), and `thisPath` is **absolute**, then resulting path is implementation dependent and therefore unspecified.

Since `pv` is absolute path, then it is returned as a resulting path.

Option 1 is wrong. It might be correct with this code:

```
...
Path pm = Path.of("/home/mikalai");
Path pv = Path.of("");
Path res = pm.resolve(pv);
```

```
...
```

Option 3 is wrong, the `pv` may not come before `pm`.

Option 4 is wrong, it might been correct if `pv` was relative (NOTE: this result is not guaranteed, as `pm` is absolute and in this case the resulting path is implementation-dependent):

```
...
Path pm = Path.of("/home/mikalai");
Path pv = Path.of("home/volha");
Path res = pm.resolve(pv);  // MAY give /home/mikalai/home/volha
...
```

*Source:*

Interface Path JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Path.html]

**Question 050103**

Given the code:

```
... // line n1
... // line n2
System.out.print(p);
```

Which two changes when done simultaneously will make the code to print `home`?

*Options (choose 2):*

1. `Path pm = Path.of("/home/mikalai"); // line n1`
2. `Path pm = Path.of("home/mikalai"); // line n1`
3. `Path p = pm.getParent(); // line n2`
4. `Path p = pm.getRoot(); // line n2`

*Answer:*

The correct options are 2 and 3.

For the exam you need to know main methods of the `Path` interface, such as: `getParent()`, `getRoot()`, `getFileName()`.

The `Path` itself consists of the pieces which are returned by `getParent()` and `getFileName()`

The `getFileName()` returns the farthest element from the root in the directory hierarchy. For our example it would be `mikalai` for both paths. The rest on the left side would be what returned by `getParent()` method. For the path containing only root component, the `getFileName()` returns `null`:

```
Path pm = Path.of("/");
Path p = pm.getFileName(); // null !
```

The `getParent()` method returns path object consisting of this path's root component, if any, and each element in the path except for the farthest from the root in the directory hierarchy (which is "file name"). Make a note that parent path may include root component if it exists. So, for `/home/mikalai` the `getParent()` will return `/home`, and for `home/mikalai` the `getParent()` will return `home`, which is required output. With this said, the correct options are 2 and 3.

The `getRoot()` method: for absolute paths it will return "root component" (for Unix-like systems -- `/`, for Windows systems -- `C:\`), and for relative paths it will return `null`.

NOTE: the `getParent()` may return a `null` too, for example for these paths the `getParent()` returns `null`:

```
Path pm = Path.of("home");
Path p = pm.getParent(); // null !
```

```
Path pm = Path.of("/");
Path p = pm.getParent(); // null !
```

*Source:*

Interface Path JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Path.html]

## 5.2. Use `Files` class to check, delete, copy or move a file or directory

**Question 050201**

Given the filesystem structure:

```
/tmp
••••a
•   •••• file.txt
•
••••b
    •••• log.txt
```

and the code fragment:

```
Path f1 = Path.of("/tmp/a/file.txt");
Path f2 = Path.of("/tmp/b");
Files.move(f1, f2, StandardCopyOption.ATOMIC_MOVE, StandardCopyOption.REPLACE_EXISTING);
```

Which are results?

*Options (choose 2):*

1. The file `/tmp/a/file.txt` is renamed to `/tmp/b/file.txt`
2. The file `/tmp/a/file.txt` is deleted.
3. The file `/tmp/a/file.txt` is not deleted.
4. Exception is thrown at runtime.
5. State of the files is undefined after code run.

*Answer:*

The correct options are 3 and 4.

You can move a file or directory by using the `Files.move(Path, Path, CopyOption...)` method. The move fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. On UNIX systems, moving a directory within the same partition generally consists of renaming the directory. In that situation, this method works even when the directory contains files.

This method takes a varargs argument – the following `StandardCopyOption` enums are supported:

- `REPLACE_EXISTING` – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
- `ATOMIC_MOVE` – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

In our case we have `/tmp/b/log.txt` file on the filesystem, it means the `/tmp/b` (target path) is a directory. Although we do provide `StandardCopyOption.REPLACE_EXISTING`, we cannot rename a file into a directory, so the code will throw exception (`java.nio.file.AccessDeniedException` which is subclass of `IOException`) and the original file will stay unchanged. This means correct options are 3 and 4.

Based on the above we can conclude that options 1 and 2 are wrong.

When the move is performed as a non-atomic operation, and an `IOException` is thrown, then the state of the files is not defined. However, we used `ATOMIC_MOVE` copy option, and this means option 5 is wrong.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

**Question 050202**

Given the filesystem structure:

```
/tmp
••••a
     •••• file.txt
```

and the code fragment:

```
Path f1 = Path.of("/tmp/a/file.txt");
Path f2 = Path.of("/tmp/b");
Files.move(f1, f2);
Files.delete(f1);
```

Which are results?

*Options (choose 2):*

1. The file `/tmp/a/file.txt` is deleted.
2. The file `/tmp/a/file.txt` is not deleted.
3. Exception is thrown at runtime.
4. Code runs silently without exception.

*Answer:*

The correct options are 1 and 3.

First, the two paths are composed. On the filesystem tree we can see that `p1` path exists, and `p2` path does not exist.

Then, the file from `p1` is renamed to `p2`. The new file `/tmp/b` is created with content of `/tmp/a/file.txt` file. As a last step of the `move()` operation, the `/tmp/a/file.txt` file is deleted.

Finally, the code attempts to delete `p1` (`/tmp/a/file.txt`), as it does not exist at this moment, the `java.nio.file.NoSuchFileException` exception is thrown.

Based on the said above, options 1 and 3 are correct, and options 2 and 4 are wrong.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

**Question 050203**

Given the filesystem structure:

```
/tmp
••••a
•    •••• file.txt
•
••••b
    •••• log.txt
```

and the code fragment:

```
Path f1 = Path.of("/tmp/a/file.txt");
Path f2 = Path.of("/tmp/b/log.txt");
... // line n1
```

Which code added at `line n1` position will replace `log.txt` with `file.txt`?

*Options (choose 1):*

1. `Files.move(f1, f2);`
2. `Files.move(f1, f2, StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES);`
3. `Files.move(f1, f2, StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.ATOMIC_MOVE);`
4. `Files.move(f1, f2, StandardOption.REPLACE_EXISTING);`

*Answer:*

The correct option is 3.

Option 1 is wrong. The `FileAlreadyExistsException` is thrown because `/tmp/b/log.txt` file exists and cannot be replaced because the `REPLACE_EXISTING` option is not specified.

Option 2 is wrong. The `StandardCopyOption.COPY_ATTRIBUTES` option is not supported by `move()` method.

The `move(Path source, Path target, CopyOption... options)` method takes a varargs argument – the following `StandardCopyOption` enums are supported:

- `REPLACE_EXISTING` – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
- `ATOMIC_MOVE` – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

The code in option 2 will throw runtime exception.

Option 4 is wrong, the `java.nio.file.CopyOption` is an empty interface, which is implemented by `java.nio.file.StandardCopyOption` enum:

```
public interface CopyOption {
}
```

```
public enum StandardCopyOption implements CopyOption {
    REPLACE_EXISTING,
    COPY_ATTRIBUTES,
    ATOMIC_MOVE;
}
```

The code in option 4 will not compile.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

**Question 050204**

Given the filesystem structure:

```
/tmp
••••a
    •••• file.txt
```

and the code fragment:

```
Path f1 = Path.of("/tmp/a/file.txt");
... // line n1
```

Which code fragment added at `line n1` position will copy `file.txt` file to the `/tmp` directory?

*Options (choose 1):*

1.

```
Path f2 = Path.of("/tmp");
Files.copy(f1, f2, StandardCopyOption.REPLACE_EXISTING);
```

2.

```
Path f2 = Paths.get("/tmp", "file.txt");
Files.copy(f1, f2, StandardCopyOption.ATOMIC_MOVE);
```

3.

```
Path f2 = Path.of("/tmp", f1.getFileName().toString());
Files.copy(f1, f2);
```

4.

```
Path f2 = Path.of(f1.getRoot().toString(), "file.txt");
Files.copy(f1, f2);
```

*Answer:*

The correct option is 3.

The `java.nio.file.Files` class defines a `static` method `copy(Path source, Path target, CopyOption... options)`.

This method copies a file to the target file with the `options` parameter specifying how the copy is performed. By default, the copy fails if the target file already exists or is a symbolic link, except if the source and target are the same file, in which case the method completes without copying the file. File attributes are not required to be copied to the target file.

The `copy(Path source, Path target, CopyOption... options)` method takes a varargs argument – the following `StandardCopyOption` and `LinkOption` enums are supported:

- `REPLACE_EXISTING` – Performs the copy even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
- `COPY_ATTRIBUTES` – Attempts to copy the file attributes associated with this file to the target file. The exact file attributes that are copied is platform and file system dependent and therefore unspecified. Minimally, the last-modified-time is copied to the target file if supported by both the source and target file stores. Copying of file timestamps may result in precision loss.
- `NOFOLLOW_LINKS` – Symbolic links are not followed. If the file is a symbolic link, then the symbolic link itself, not the target of the link, is copied. It is implementation specific if file attributes can be copied to the new link. In other words, the COPY_ATTRIBUTES option may be ignored when copying a symbolic link.

Option 1 is wrong. Although we use `REPLACE_EXISTING`, we cannot copy a file to a file with the same name as existing directory, i.e. the code attempts to copy `/tmp/a/file.txt` to `/tmp` file, not to `/tmp/file.txt` file. Since root directory may not contain both `/tmp` directory and `/tmp` file, the code fails at runtime.

Option 2 is wrong. The `Paths.get(String first, String... more)` is a correct way to compose `Path`, but

`StandardCopyOption.ATOMIC_MOVE` may not be used with `copy(...)` method. The code will throw runtime exception.

Option 3 is correct. The `f1.getFileName()` returns `Path("file.txt")` and then converted to `String("file.txt")`. The `Path.of(...)` method accepts strings and composes the path inatance.

Option 4 is wrong. The `f1.getRoot().toString()` returns `/` on Unix-like machines, or `C:\` on Windows machines. The code runs fine, but it copies the file to `/` directory (to the file system root), not to `/tmp`.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

**Question 050205**

Given the filesystem structure:

```
/tmp
••••a
    •••• file.txt
```

and the code fragment:

```
Stream<Path> files = Files.walk(Paths.get("/tmp"), 2);
files.forEach (n -> {
    System.out.println(n);
});
```

Wha is the output?

*Options (choose 1):*

1.
```
/tmp
/tmp/a
```

2.
```
/tmp/a
/tmp/a/file.txt
```

3.
```
/tmp
/tmp/a
/tmp/a/file.txt
```

4.
```
a
a/file.txt
```

*Answer:*

The correct option is 3.

The `java.nio.file.Files` class defines several `static` methods fo walking file system: `walk(...)`, `walkFileTree(...)`.

The code demonstrates one of two `walk(...)` methods, the one which accepts a files visiting depth (`2`).

The `Files.walk(Path start, int maxDepth, FileVisitOption... options)` methods returns a `Stream` that is lazily populated with `Path` by walking the file tree rooted at a given starting file. The file tree is traversed depth-first, the elements in the stream are `Path` objects that are obtained as if by resolving the relative path against start.

The stream walks the file tree as elements are consumed. The `Stream` returned is guaranteed to have at least one element, the starting file itself. This makes options 2 and 4 wrong.

The `maxDepth` parameter is the maximum number of levels of directories to visit. A value of `0` means that only the starting file is visited, unless denied by the security manager. A value of `MAX_VALUE` may be used to indicate that all levels should be visited.

Option 1 is wrong. It might be correct if we had the `maxDepth` parameter set to `1`.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

## 5.3. Use Stream API with `Files`

**Question 050301**

Given the following folders structure:

```
C:\
 |
 tmp
    |-README.TXT
    |-code
         |-A.java
```

Which code fragment will print:

```
A.java
README.txt
```

*Options (choose 1):*

1.

```
Path f = Paths.get("C:\\tmp");
Stream<Path> stream = Files.find(f, 3, (a, b) -> b.isRegularFile());
stream.sorted().forEach(System.out::println);
```

2.

```
Path f = Paths.get("C:\\tmp");
Stream<Path> stream = Files.find(f, 3, (a, b) -> b.isDirectory());
stream.sorted().forEach(s -> System.out.println(s.getFileName()));
```

3.

```
Path f = Paths.get("C:\\tmp");
Stream<Path> stream = Files.find(f, 3, (a, b) -> b.isRegularFile());
stream.sorted().forEach(s -> System.out.println(s.getFileName()));
```

4.

```
Path f = Paths.get("C:\\tmp");
Stream<Path> stream = Files.find(f, 3, (a, b) -> b.isRegularFile());
stream.sorted().forEach(s -> System.out.println(s));
```

*Answer:*

The correct option is 3. You can get short file name from absolute path using `Path.getFileName()`.

Option 1 is wrong. It prints full file paths (both folders and filename), the output will be as follows:

```
C:\tmp\code\A.java
C:\tmp\README.txt
```

Option 2 is wrong. It prints directories names, the output will be as follows:

```
C:\tmp
C:\tmp\code
```

Option 4 is wrong. It prints full file paths, exactly as Option 1 above, the output will be as follows:

```
C:\tmp\code\A.java
C:\tmp\README.txt
```

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

Interface BasicFileAttributes JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/attribute/BasicFileAttributes.html]

Interface Path JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Path.html]

**Question 050302**

Given the following folders structure:

```
C:\
 |
 tmp
    |-README.TXT
    |-code
        |-A.java
```

and the following code fragment:

```
Path f = Paths.get("C:\\tmp");
Stream<Path> stream = Files.walk(f);
```

```
stream.forEach(s -> System.out.println(s));
```

What is the output?

*Options (choose 1):*

1.

```
C:\
C:\tmp
C:\tmp\code
C:\tmp\code\A.java
C:\tmp\README.txt
```

2.

```
C:\tmp
C:\tmp\code
C:\tmp\code\A.java
C:\tmp\README.txt
```

3.

```
C:\tmp
C:\tmp\code
```

4.

```
C:\tmp\code\A.java
C:\tmp\README.txt
```

*Answer:*

The correct option is 2.

Option 1 is wrong. The `f` variable defines a root for walking the file tree. The code will visit only directories and files under `C:\tmp`.

Option 3 is wrong. The `Files.walk(Path start, FileVisitOption... options)` adds both directories and files in the resultng stream.

Option 4 is wrong. The `Files.walk(Path start, FileVisitOption... options)` adds both directories and files in the resultng stream.


`Files.walk(..)` does NOT have an option to distinguish files and folders during walk.

`Files.find(..)` may accept `BiPredicate<Path,BasicFileAttributes> matcher` parameter to distinguish files and folders during search.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

**Question 050303**

Assuming that `f` variable is properly intialized and points to some Java class source code.

---

```
Path f = Paths.get(...);
```

Which two code fragments will print all classes/packages imported by that Java class?

*Options (choose 2):*

1.

```
Stream<String> stream = Files.lines(f);
stream.filter(s -> s.startsWith("import")).forEach(s ->
System.out.println(s.substring("import".length() + 1, s.length() - 1)));
```

2.

```
List<String> list = Files.lines(f);
list.stream().filter(s -> s.startsWith("import")).forEach(s ->
System.out.println(s.substring("import".length() + 1, s.length() - 1)));
```

3.

```
Stream<String> stream = Files.readAllLines(f);
stream.filter(s -> s.startsWith("import")).forEach(s ->
System.out.println(s.substring("import".length() + 1, s.length() - 1)));
```

4.

```
List<String> list = Files.readAllLines(f);
list.stream().filter(s -> s.startsWith("import")).forEach(s ->
System.out.println(s.substring("import".length() + 1, s.length() - 1)));
```

*Answer:*

The correct options are 1 and 4.

Option 2 is wrong. The `Files.lines(path)` method returns `Stream<String>`, not `List<String>`.

Option 3 is wrong. The `Files.readAllLines(path)` method returns `List<String>`, not `Stream<String>`.


The `Files.readAllLines(path)` method **eagerly** reads all lines from a file and returns `List<String>`.

The `Files.lines(path)` method reads all lines from a file as a `Stream<String>`. Unlike `readAllLines`, this method does NOT read all lines into a `List`, but instead populates **lazily** as the stream is consumed.

*Source:*

Class Files JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html]

## Chapter 6. Migration to a Modular Application

### 6.1. Migrate the application developed using a Java version prior to SE 9 to SE 11 including top-down and

## bottom-up migration, splitting a Java SE 8 application into modules for migration

**Question 060101**

You need to split monolithic Java 8 application into modules to create Java 11 modular application. The original JAR file contains the following types:

```
application.jar
••••coffee
•       Arabica.java
•       Coffee.java
•       CoffeeMachine.java
•       Robusta.java
•       SteamCoffeeMachine.java
•
••••drink
        Drink.java
        Machine.java
```

Which will be the best design to split classes?

*Options (choose 1):*

1.

```
coffee.jar
••••coffee
        Arabica.java
        Coffee.java
        CoffeeMachine.java
        Robusta.java
        SteamCoffeeMachine.java
```

```
drink.jar
••••drink
        Drink.java
        Machine.java
```

2.

```
coffee-base.jar
••••coffee
        Coffee.java
        CoffeeMachine.java
        SteamCoffeeMachine.java
```

```
coffee-types.jar
••••coffee
        Arabica.java
        Robusta.java
```

```
drink.jar
••••drink
        Drink.java
        Machine.java
```

```
coffee-drink.jar
••••coffee
        Coffee.java
        Arabica.java
        Robusta.java
```

```
coffee-machine.jar
••••coffee
        CoffeeMachine.java
        SteamCoffeeMachine.java
```

```
drink.jar
••••drink
        Drink.java
        Machine.java
```

4.

```
drink.jar
••••coffee
•       Arabica.java
•       Coffee.java
•       Robusta.java
•
••••drink
        Drink.java
```

```
machine.jar
••••coffee
•       CoffeeMachine.java
•       SteamCoffeeMachine.java
•
••••drink
        Machine.java
```

*Answer:*

The correct option is 1.

Options 2, 3, and 4 are wrong. A Java package may not be splitted between several modules. Different modules MAY contain the same package, but these modules may not be used together on the same module path and may not export the package. Since the new modules must be used together as part of new modular applications, they all will be on module path and JVM forbids the same package existed in several modules when one module requires another, and another module exports the package.

Option 2 might be correct if we used different package name in `coffee.types` module as follows:

```
coffee-types.jar
••••coffee
    •
    ••••types
            Arabica.java
            Robusta.java
```

Option 3 might be correct if we used different package name in `coffee.machine` module as follows:

```
coffee-machine.jar
```

```
••••coffee
    •
    ••••machine
            CoffeeMachine.java
            SteamCoffeeMachine.java
```


When code in a module refers to a type in a package then that package is guaranteed to be defined either in that module or in precisely one of the modules read by that module.

*Source:*

The State of the Module System. Section 2.3 Readability [http://openjdk.java.net/projects/jigsaw/spec/sotms/]

**Question 060102**

You need to migrate a Java 8 application to Java 11 modular application using "top down" method. The application consists of three JARs: `app.jar`, `lib1.jar`, and `lib2.jar`. Classes in the `app.jar` depend on classes in `lib1.jar` and `lib2.jar`.

Which command will be used to run the application after migration?

*Options (choose 1):*

1.

```
java --module-path ./app.jar;./lib1.jar;./lib2.jar -m app/by.boot.java.App
```

2.

```
java -cp ./app.jar;./lib1.jar;./lib2.jar by.boot.java.App
```

3.

```
java --module-path ./app.jar -cp ./lib1.jar;./lib2.jar -m app/by.boot.java.App
```

4.

```
java -cp ./app.jar --module-path ./lib1.jar;./lib2.jar by.boot.java.App
```

*Answer:*

The correct option is 1.

Top down migration refers to the order in which several JARs may be migrated to corresponding modules (where the migration is one to one).

Given JARs `app.jar`, `lib1.jar`, and `lib2.jar` where some classes in `app.jar` depend on some classes in `lib1.jar` and `lib2.jar`, top down migration involves migrating `app.jar` first.

As a consequence, `lib1.jar` and `lib2.jar` must be run on the module path as otherwise they cannot be accessed by what is now module `app`. If run on the classpath they would become an unnamed module and named modules cannot access unnamed modules.

By running `lib1.jar` amd `lib2.jar` on the module path, they becomes automatic modules with names, and can be accessed by module `app`.

Module `app` can now, in its `module-info.java` file, declare its dependence on the automatic modules created from `lib1.jar`

and `lib2.jar`. For example, if `lib1` is the name given to automatically generated module based on `lib1.jar`, module `app` can refer to this automatic module with the directive:

```
requires lib1;
```

Option 1 is correct. The `app.jar` will be a named module, while `lib1.jar` and `lib2.jar` will be automatic modules. Automatic modules export by default all their packages, so all the classes will be visible to `app` module.

Option 2 is wrong. The `app.jar` must be a named module according to top down migration process, and named module should be placed on the module path, not on the classpath. The code will run, but it demonstrates Java 8 application non-modular application.

Option 3 is wrong. The `lib1.jar` and `lib2.jar` are placed on the classpath and became part of the unnamed module. The named module `app` cannot access unnamed module classes.

Option 4 is wrong. The `app.jar` must be a named module according to top down migration process, and named module should be placed on the module path, not on the classpath.

*Source:*

Understanding Java 9 Modules. [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

**Question 060103**

You need to migrate a Java 8 application to Java 11 modular application using "bottom up" method. The application consists of three JARs: `app.jar`, `lib1.jar`, and `lib2.jar`. Classes in the `app.jar` depend on classes in `lib1.jar` and `lib2.jar`.

Which two commands will be used to run the application after migration?

*Options (choose 2):*

1.
```
java --module-path ./app.jar;./lib1.jar;./lib2.jar -module app/by.boot.java.App
```

2.
```
java -cp ./app.jar;./lib1.jar;./lib2.jar by.boot.java.App
```

3.
```
java --module-path ./app.jar -cp ./lib1.jar;./lib2.jar -add-modules app -module
app/by.boot.java.App
```

4.
```
java -cp ./app.jar --module-path ./lib1.jar;./lib2.jar --add-modules lib1,lib2
by.boot.java.App
```

*Answer:*

The correct options are 1 and 4.

Bottom up migration refers to the order in which several JARs may be migrated to corresponding modules (where the migration is one to one).

Given JARs `app.jar`, `lib1.jar`, and `lib2.jar` where some classes in `app.jar` depend on some classes in `lib1.jar` and `lib2.jar`, bottom up migration involves migrating `lib1.jar` and `lib2.jar` first.

As a consequence, `lib1.jar` and `lib2.jar` must be run on the module path. This makes options 2 and 3 wrong.

The `app.jar` can be placed to the classpath and becomes part of unnamed module. This demonstrates option 4. Make a note of `--add-modules` option. We need to use "`--add-modules lib1,lib2`" option to tell the Java runtime to include the modules by name to the default root set. It is needed as JVM cannot built proper tree of modules, because the main class (`by.boot.java.App`) is in the unnamed module and the unnamed module does not contain module descriptor with "`requires XXX`" directives.

Also, `app.jar` can be placed to the module path and become an automatic module. The automatic module exports all packages and reads all modules from the module path. This is demonstrated in option 1. The `lib1.jar` and `lib2.jar` are named modules (migrated to Java 11) and `app.jar` is automatic module (created from Java 8 JAR file).

*Source:*

Understanding Java 9 Modules. [https://www.oracle.com/corporate/features/understanding-java-9-modules.html]

## 6.2. Use `jdeps` to determine dependencies and identify way to address the cyclic dependencies

**Question 060201**

As a first step to migrating application to Java 11, you need to analyze Java 8 JAR file and determine which class (if any) misuses Java API.

Which command will you use?

*Options (choose 1):*

1.
```
jdeps -s non-modular.jar
```
2.
```
jdeps --list-deps non-modular.jar
```
3.
```
jdeps -jdkinternals non-modular.jar
```
4.
```
jdeps --generate-module-info . non-modular.jar
```

*Answer:*

The correct option is 3.

Java API can be classified as:

- **standardized** -- those public classes which found in packages exported by `java.*` modules (these can be `java.*` and `javax.*` packages).
- **supported** -- those public classes which found in packages exported by `jdk.*` modules.
- **unsupported** -- most `com.sun.*` packages and all `sun.*` packages as well as all non-public classes are internal and can change between different versions and JREs. Depending on these is the most unstable as such code could theoretically stop working on any minor update.

Your application should never depend on unsupported APIs. In order to find out which class is using unsupported APIs you can use `jdeps` JDK utility with `-jdkinternals` or (`--jdk-internals`) command line option.

Assume there a JAR file with a class which uses unsupported API:

---

```
package a;
import sun.misc.Unsafe;
public class ClassA {
    Unsafe u = null;
}
```

As was mentioned above, all `sun.*` classes are part of unsupported API.

The option `--jdk-internals` makes `jdeps` list all internal APIs that the referenced JARs depend on, including those exported by `jdk.unsupported`. The output contains:

- The analyzed JAR and the module containing the problematic API
- The specific classes involved
- The reason why that dependency is problematic

```
jdeps -jdkinternals non-modular.jar
```

Sample output will be as follows:

```
non-modular.jar -> jdk.unsupported
   a.ClassA                                          -> sun.misc.Unsafe
JDK internal API (jdk.unsupported)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependence on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                        Suggested Replacement
----------------                        ---------------------
sun.misc.Unsafe                         See http://openjdk.java.net/jeps/260
```

We can clearly see now that `a.ClassA` misuses JDK API and refers `sun.misc.Unsafe` class. Thus option 3 is correct.

Option 1 is wrong. It just gives a summary of dependency, without telling which class exactly uses unsupported API:

```
C:\1Z0-817>jdeps -s non-modular.jar
non-modular.jar -> java.base
non-modular.jar -> jdk.unsupported
```

Option 2 is wrong. It shows also only dependencies without specific class:

```
C:\1Z0-817>jdeps --list-deps non-modular.jar
   java.base
   jdk.unsupported
```

Option 4 is wrong. It created a module definition for non-modular JAR, e.g.

```
module non.modular {
    requires jdk.unsupported;
    exports a;
}
```

*Source:*

jdeps tool guide [https://docs.oracle.com/en/java/javase/11/tools/jdeps.html]

**Question 060202**

You want to automate Java 8 JAR migration to a Java 11 module and generate module definition by using `jdeps` utility. The JAR file classes have dependency injection annotations which are processed by application framework and values are injected into classes with help of Reflection API.

Which command will you use?

*Options (choose 1):*

1.

```
jdeps -dotoutput . non-modular.jar
```

2.

```
jdeps --generate-open-module . non-modular.jar
```

3.

```
jdeps --check module-name=non.modular --module-path non-modular.jar
```

4.

```
jdeps --generate-module-info . non-modular.jar
```

*Answer:*

The correct option is 2.

In order to provide reflective access to your module, Java 9 introduced `open` keyword. You can create open module by using `open` modifier in module definition. An open module grants reflective access to all of it's packages to other modules.

The question says there is a framework, that relies on reflection when dos dependency injection, so you must use `open` keyword to enable reflective access for module classes.

You can enable reflective access for specified packages of your module by using `opens` directive in package declaration:

```
module non.modular {
    opens a;
}
```

or using `open` keyword in module definition:

```
open module non.modular {
    ...
}
```

The `jdeps` utility can generate module definition for open modules, you need to provide `--generate-open-module PATH` option:

```
jdeps --generate-open-module . non-modular.jar
```

result:

```
open module non.modular {
    requires jdk.unsupported;
}
```

This makes option 2 correct.

Option 1 is wrong. It allows to generate `.dot` files describind dependencies. The `.dot` files can be visualized to pictures to provide visual representation of dependencues:

```
digraph "non-modular.jar" {
    // Path: non-modular.jar
   "a"                                                -> "java.lang (java.base)";
   "a"                                                -> "sun.misc (JDK internal API
(jdk.unsupported))";
}
```

Option 3 is wrong. It helps to generate module definition, but the classes (packages) will not be accessible via Reflection API.

Option 4 is wrong. The `jdeps` tool does not recognize automatic modules, the command will fail.

*Source:*

jdeps tool guide [https://docs.oracle.com/en/java/javase/11/tools/jdeps.html]

**Question 060203**

A modular Java 11 application has `modA` with module definition:

```
module modA {
    requires modB;
    exports pkgA;
}
```

amd `modB`:

```
module modB {
    requires modA;
    exports pkgB;
}
```

What will help to compile the application?

*Options (choose 1):*

1. Use `-Xlint:circular` command line option for `javac`
2. Use `-Xlint:module` command line option for `javac`
3. Use `-Xlint:none` command line option for `javac`
4. Create a new module `modC` that both `modA` and `modB` are dependent, remove `requires modB;` from `modA` definition.

*Answer:*

The correct option is 4.

It is not allowed to have circular dependencies between modules. In other words, If `modA` requires `modB`, then `modB` cannot also require `modA`. The module dependency graph must be an acyclic graph. In case of cyclic dependency you need to create a new module `modC` that both `modA` and `modB` are dependent, move to `modC` classes (make sure change package name when moving classes) from `modB` which `modA` depends on, and remove from `modA` definition `requires modB;` directive.

Option 1 is wrong. This is invalid command line option for `javac`, the command will fail.

Option 2 is wrong. It will enable all module system-related issues, but it will not break circular dependency between `modA` and `modB`.

Option 3 is wrong. It will disable all warnings, but it will not break circular dependency between `modA` and `modB`.

*Source:*

jdeps tool guide [https://docs.oracle.com/en/java/javase/11/tools/jdeps.html]

javac tool guide [https://docs.oracle.com/en/java/javase/11/tools/javac.html]

## Chapter 7. Local-Variable Type Inference

### 7.1. Use local-variable type inference

**Question 070101**

Which statement is true about `var` reserved type name?

*Options (select 1):*

1. `var` varialbes are effectively final
2. `var` variables are dynamically typed
3. Public `var` varialbes are inherited by subclasses
4. You can use `var` variables with diamond operator on the right-hand side

*Answer:*

The correct option is 4.

Option 1 is wrong. Local variables declared with `var` are non-`final` by default. However, the `final` modifier can be added to `var` declaration:

```
final var age = 40;
```

There is no shorthand for `final var` in Java.

Option 2 is wrong. Java (opposite to JavaScript) is a statically typed language. With `var`, the Java compiler infers the type of the variable at compile time, using type information obtained from the variable's initializer (right-hand side, RHS). The inferred type is then used as the static type of the variable. Typically, this is the same as the type you would have written explicitly, so a variable declared with `var` behaves exactly as if you had written the type explicitly.

Option 3 is wrong. Only local variables may be declared with `var` type name, so they are not inherited. Only top level members are inherited -- non-static methods and instance variables.

Option 4 is correct. It is a valid syntax, but the result is not much use:

```
var list = new ArrayList<>();
```

is equal to:

```
var list = new ArrayList<Object>();
```

If you want a collection of some explicit type (e.g. `String`), either use traditonal diamond syntax:

```
ArrayList<String> list = new ArrayList<>();
```

or use `var` with explicit type on the right hand side (RHS):

```
var list = new ArrayList<String>();
```

*Source:*

Local Variable Type Inference: Frequently Asked Questions [https://openjdk.java.net/projects/amber/LVTIFAQ.html]

**Question 070102**

Given:

```
class A {}
class B extends A {}
```

Which code fragment will FAIL to compile?

*Options (select 1):*

1.

```
{
    var a = new A();
    a = new B();
}
```

2.

```
{
    var b = new B();
    b = new B();
}
```

3.

```
{
    var c = new B() {};
    c = new B() {};
}
```

4.

```
{
    var d = (A) null;
    d = (B) null;
}
```

*Answer:*

The correct option is 3.

Option 1 is wrong. The code will successfully compile. The code is identical to:

```
A a = new A();
a = new B();
```

Variable `a` type inferred as `A` at compile time. You can safely assign `B` (subclass) instance reference to `A` (superclass) type variable.

Option 2 is wrong. The code will successfully compile. The code is identical to:

```
B b = new B();
b = new B();
```

Variable `b` type inferred as `B` at compile time. You just reassign variable to the new instance of `B`.

Option 3 is correct. The code will not compile. If you notice, both lines are creating instances of anonymous inner class, which is subclassing `B`. The pseudo code demonstrating this option will look as follows:

```
WrapperType$1 c = new WrapperType$1();
c = new WrapperType$2();
```

In the first line compiler infers that type of `c` variable is `WrapperType$1` (this is anonymous type, we don't know exactly its name). The anonymous type may not be reused to create another instance, as we don't know it's name (hence -- anonymous), so at the second line, where we create `new B() {}`, it will be some `WrapperType$2` instance, but not `WrapperType$1`. The `WrapperType$1` and `WrapperType$2` are not sub-/super-class one of another, they just sibling types (inherited from `B`) and they are not mutually assignment-compatible, so the second line will fail at compile time.

Option 4 is wrong. The code will compile fine.

Althought this is not allowed:

```
var d = null; // FAILS, cannot infer type
```

but this is OK -- compiler can infer variable type from cast expression, even when right-hand side expression is `null`:

```
var d = (A) null; // COMPILES, can infer type from cast expression
```

this is equivalent:

```
A d = null;
```

and as was mentioned above in Option 1, type `B` is assignable to variables of type `A`, so the second line will compile too.

*Source:*

Local Variable Type Inference: Frequently Asked Questions [https://openjdk.java.net/projects/amber/LVTIFAQ.html]

**Question 070103**

Which two classes will compile successfully?

*Options (select 2):*

1.

```
class A {
    var a = 1;
}
```

2.

```
class B {
    static {
        var b = 1;
    }
}
```

3.

```
class C {
    {
        var c = 1;
    }
}
```

4.

```
class D {
    {
        var d = {1,2};
    }
}
```

5.

```
class E {
    {
        var[] e = new int[] {1,2};
    }
}
```

*Answer:*

The correct options are 2 and 3.

Option 1 is wrong. The code will fail to compile. Class `A` demonstrates inatance variable, while `var` may be used as a local variable only.

Option 2 is correct. Class `B` shows static initializer block, and the variable `b` is local to this block -- the code is valid.

Option 3 is correct. It demonstrates instance initializer block and `var c` is a local variable.

Option 4 is wrong. You can use `var` for array types, but you need to specify exlicitly type of the array and new array creation:

```
{
    var d = {1,2}; // FAILS to compile
}
```

```
{
    var d = new int[] {1,2}; // SUCCESSFULLY compiles
}
```

Option 5 is wrong. The `var` is NOT a type, at is reserved type name. So, expression is `var[]` is syntactically wrong.

```
{
    var[] e = new int[]{1,2}; // FAILS to compile
}
```

```
{
    var e = new int[]{1,2}; // SUCCESSFULLY compiles
}
```

*Source:*

Local Variable Type Inference: Frequently Asked Questions [https://openjdk.java.net/projects/amber/LVTIFAQ.html]

JEP 286 [https://openjdk.java.net/jeps/286]

**Question 070104**

Given the classes:

```
 1 class User {
 2     String name;
 3 }
 4 public class UserDB {
 5     var list = new ArrayList<User>();
 6     public var displayUsers() {
 7         var list = new ArrayList<>();
 8         var user = new User();
 9         user.name = "Mikalai";
10         list.add(user);
11         for (var u : list) {
12             System.out.println(u.name);
13         }
14         return 0;
15     }
16 }
```

Which three lines FAIL to compile?

*Options (select 3):*

1. 5
2. 6
3. 7
4. 8
5. 11
6. 12

*Answer:*

The correct options are 1, 2, and 6.

Option 1 is correct. '`var`' resered type name may be used only for local vaiables, and line 5 demonstrates instance variable declaration.

Option 2 is correct. '`var`' may not be used as method return type in method definition.

Option 3 is wrong. This line demonstrates valid local vaiable declaration. Please, note:

```
var list = new ArrayList<>();
```

is syntactically equal to:

```
ArrayList<Object> list = new ArrayList<>();
```

I.e. the colection is not parameterized, it's a "raw" collection.

Option 4 is wrong. Again, valid instance variable declaration, the variable type inferred from right-hand side (RHS) of the assignment expression.

Option 5 is wrong. '`var`' may be used in as iteration variable in enhanced for-loop. Please, note:

```
for (var u : list) {
```

is syntactically equal to:

```
for (Object u : list) {
```

Option 6 is correct. The `u` variable is of `Object` type, not as `User`, because the collection is not parameterized. Although list contains `User` values, it's declared as `ArrayList<Object>`, so compiler fails when you try to access `u.name`.

*Source:*

Local Variable Type Inference: Frequently Asked Questions [https://openjdk.java.net/projects/amber/LVTIFAQ.html]

JEP 286 [https://openjdk.java.net/jeps/286]

## 7.2. Create and use lambda expressions with local-variable type inferred parameters

**Question 070201**

You are writing a pipeline which processes stream of strings. Application uses a framework with constraint validator. Which code properly assures that stream has no `null` values?

*Options (select 1):*

1.
```
list.stream().map(@NonNull s -> s.toUpperCase()).collect(Collectors.toList());
```

2.
```
list.stream().map((@NonNull s) -> s.toUpperCase()).collect(Collectors.toList());
```

3.
```
list.stream().map((@NonNull var s) -> s.toUpperCase()).collect(Collectors.toList());
```

4.
```
list.stream().map((@NotNull final v) -> v.toUpperCase()).collect(Collectors.toList());
```

*Answer:*

The correct option is 3.

As of Java 11, `var` is allowed to be used when declaring the formal parameters of implicitly typed lambda expressions. One benefit of this change is that modifiers, notably annotations, can be applied to local variables and lambda formals without losing brevity.

Option 1 is wrong. Annotation may not be applied to a lambda parameter without type.

Option 2 is wrong. Same as option 1: annotation may not be applied to a lambda parameter without concrete type or `var` reserved type name.

Option 3 is correct. As of Java 11 you can use the `var` type name as a lambda parameter type. The `var` type name was introduced in Java 10 as local variable type inference. From Java 11 `var` can also be used for lambda parameter types. Here is an example of using the Java `var` "pseudo-type" as parameter type in a lambda expression:

```
Function<String, String> toLowerCase = (var input) -> input.toLowerCase();
```

The type of the parameter declared with the `var` type name above will be inferred to the type `String`, because the type declaration of the variable has its generic type set to `Function<String, String>`, which means that the parameter type and return type of the `Function` is `String`.

Now check the `@NotNull` annotation from the package `javax.validation.constraints`:

```
@Target(value={METHOD,FIELD,ANNOTATION_TYPE,CONSTRUCTOR,PARAMETER})
@Retention(value=RUNTIME)
@Documented
@Constraint(validatedBy={})
public @interface NotNull
```

As you can see, it may be applied to a parameter, but when it applied to a parameter, it must be followed by variable type, not variable name. For this reason, `var` may be used as "placeholder" for variable type, and defining annotation on `var` is valid.

Option 4 is wrong. There is no variable type (or "type placeholder" -- `var`), and `final` is just a modifier. This code would be valid:

```
list.stream().map((@NotNull final String v) ->
v.toUpperCase()).collect(Collectors.toList());
```

or this:

```
list.stream().map((@NotNull final var v) -> v.toUpperCase()).collect(Collectors.toList());
```

*Source:*

JEP 323: Local-Variable Syntax for Lambda Parameters [https://openjdk.java.net/jeps/323]

**Question 070202**

Given the code:

```
List<String> list = List.of("B", "C", "A");
BiFunction<String, String, String> bf = (var a, b) -> a + b;
Function<String, String> f = c -> bf.apply(c, c);
list.stream().map(f).sorted().forEach(System.out::print);
```

```

```

What is the result?

*Options (select 1):*

1.

```
AABBCC
```

2.

```
BBCCAA
```

3.

```
CCBBAA
```

4. Compilation fails

*Answer:*

The correct option is 4.

Option 4 is correct, this line will fail to compile:

```
BiFunction<String, String, String> bf = (var a, b) -> a + b;
```

As of Java 11, `var` is allowed to be used when declaring the formal parameters of implicitly typed lambda expressions. However, there are some restrictions:

- The `var` must be used with each lambda parameter.
- Mixing with explicit types is not allowed.

These are **invalid** examples:

```
BiFunction<String, String, String> bf = (var a, b) -> a + b;
BiFunction<String, String, String> bf = (a, var b) -> a + b;

BiFunction<String, String, String> bf = (var a, String b) -> a + b;
BiFunction<String, String, String> bf = (String a, var b) -> a + b;

BiFunction<String, String, String> bf = (String a, b) -> a + b;
BiFunction<String, String, String> bf = (a, String b) -> a + b;
```

These are **valid** examples:

```
BiFunction<String, String, String> bf = (var a, var b) -> a + b;
BiFunction<String, String, String> bf = (String a, String b) -> a + b;
BiFunction<String, String, String> bf = (a, b) -> a + b;
```

Based on the said above, options 1, 2 and 3 are invalid. In case if `BiFunction` was declared properly, the output would be:

`AABBCC`.

*Source:*

JEP 323: Local-Variable Syntax for Lambda Parameters [https://openjdk.java.net/jeps/323]

**Question 070203**

You need to write a `BiFunction` which converts first string parameter to uppercase, second string parameter to lowercase and return concatenated string.

To ensure that `null` values handled properly, you created new annotation `@StringParam(default="default value")` which annotates lambda parameter and provides default value if any passed in parameter is `null`.

Which two are the correct syntax to declare the `BiFunction`?

*Options (select 2):*

1.

```
BiFunction<String, String, String> bf =
        (@StringParam(default="A") a, @StringParam(default="B") b) ->
        a.toUpperCase() + b.toLowerCase();
```

2.

```
BiFunction<String, String, String> bf =
        (@StringParam(default="A") var a, @StringParam(default="B") var b) ->
        a.toUpperCase() + b.toLowerCase();
```

3.

```
BiFunction<String, String, String> bf =
        (@StringParam(default="A") String a, @StringParam(default="B") String b) ->
        a.toUpperCase() + b.toLowerCase();
```

4.

```
BiFunction<String, String, String> bf =
        (@StringParam(default="A") var a, @StringParam(default="B") String b) ->
        a.toUpperCase() + b.toLowerCase();
```

*Answer:*

The correct options are 2 and 3.

Option 1 is wrong. You may not place parameter annotation directly on the lambda parameter.

Option 2 is correct. As of Java 11 you can place parameter targeted annotation (`@Target({ElementType.PARAMETER})`) on the `var` type name.

Option 3 is correct. It demonstrates pre-Java 11 approach to place annotation on lambda parameters -- by using explicit type name (`String`).

Option 4 is wrong. Even without annotation this code would fail, you may not mix `var` parameters with explicit type (e.g. `String`) parameters in a single labmda expression.

*Source:*

JEP 323: Local-Variable Syntax for Lambda Parameters [https://openjdk.java.net/jeps/323]

## Chapter 8. Lambda Expressions

### 8.1. Create and use lambda expressions

#### Question 080101

Given the code:

```
 1 interface Validator {
 2     boolean validate();
 3 }
 4
 5 public class Vehicle {
 6
 7     boolean startEngine(Driver driver) {
 8         Validator v = new Validator() {
 9             @Override
10             public boolean validate() {
11                 return !driver.isDrunk();
12             }
13         };
14         return v.validate();
15     }
16 }
```

Which code correctly replaces inner local anonymous class defined on lines 8-13 with lambda expression?

*Options (select 1):*

1.

```
Validator v = (Driver driver) -> { return !driver.isDrunk(); };
```

2.

```
Validator v = new Driver() -> { return !driver.isDrunk(); };
```

3.

```
Validator v = (driver) -> { return !driver.isDrunk(); };
```

4.

```
Validator v = () -> { return !driver.isDrunk(); };
```

*Answer:*

The correct option is 4.

Since the `Validator` is a functional interface, the boilerplate of implementing the `public boolean validate()` method can be abstracted away using a lambda expression. The general format for implementing a `Validator` with a lambda expression is as follows:

```
// no statements inside lambda body, single boolean expression

Validator v = () -> boolean_expression
```

or another syntax:

```
// when you must place multiple statements inside lambda body {...} separated by semicolon

Validator v = () -> { statement; statement; return boolean_expression; };
```

```
// can be used with a single expression too, but in this case prepend with 'return' too

Validator v = () -> { return boolean_expression; };
```

A `Validator` can be implemented by using a zero-argument lambda expression containing an expression within the lambda body. The key is that the implementation takes no arguments and returns boolean.

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

**Question 080102**

Oracle University wants you to sort all its students based on their height. Current code is shown below

```
public class Student {
    public int getHeight() {
        ...
    }
}
```

```

ArrayList<Student> students = ...

students.sort(new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getHeight() - s2.getHeight();
    }
});
```

Which three lambda expressions demonstrate equivalent sorting logic?

*Options (select 3):*

1.

```
students.sort((s1, s2) -> s1.getHeight() - s2.getHeight());
```

2.

```
students.sort((s1, s2) -> { return s1.getHeight() - s2.getHeight(); });
```

3.

```
students.sort((s1, s2) -> { s1.getHeight() - s2.getHeight(); });
```

4.

```
students.sort((Student s1, Student s2) -> { return s1.getHeight() - s2.getHeight(); });
```

5.

```
students.sort((Student s1, Student s2) ->  return s1.getHeight() - s2.getHeight(); );
```

*Answer:*

The correct options are 1, 2, and 4.

A lambda expression is composed of parameters, an arrow, and a body:

- A list of parameters - in our case it mirrors the parameters of the `compare` method of a `Comparator` - two `Student`s.

  NOTE: If the parameter types of a lambda expression can be inferred, you can omit them. Option 1 and 2 demonstrate valid examples of type inferrence.
- An arrow - the arrow `->` separates the list of parameters from the body of the lambda.
- The body of the lambda - compare two students using their heights. The expression is considered the lambda's return value.

  The basic syntax of a lambda is either - **note the curly braces absence and semicolon absence in the end**:

  ```
  (parameters) -> expression
  ```

  Or another syntax is shown below - **note the curly braces for statements and semicolon in the end**:

  ```
  (parameters) -> { statements; }
  ```

  Now, let's take a look at some examples:
  - This lambda has no parameters and returns void. It's similar to a method with an empty body - `public void doIt() { }`:

    ```
    () -> {}
    ```

  - This is a valid lambda which has no parameters and returns a `String` as an expression:

    ```
    () -> "Java 8 Rule"
    ```

  - This lambda has no parameters and returns a `String` using an explicit `return` statement. That's why curly braces and semicolon are required.

    ```
    () -> { return "Java 8 Rule"; }
    ```

  - An example of invalid labbda: `return` is a control-flow **statement**, so curly braces are a must.

    ```
    (Integer i) -> return "Age : " + i;
    ```

To make this lambda valid, curly braces are required as follows: `(Integer i) -> { return "Age : " + i; }`
- An example of invalid lambda: "`Java 9 Rule Too`" is an expression, not a statement.

```
() -> { "Java 9 Rule Too"; }
```

To make this lambda valid, you can remove the curly braces and semicolon as follows: `() -> "Java 9 Rule Too"`.

Or if you prefer, you can use an explicit `return` statement as follows: `() -> { return "Java 9 Rule Too"; }`.

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

**Question 080103**

Given the class:

```
public class StreamTest {
    @FunctionalInterface
    interface NameFilter extends Predicate<String> {
        public default boolean test(String str) {
            return str.contains("Ja");
        }
        public boolean doTest();
    }
    public static void main(String[] args) {
        List<String> strs = List.of("Jada", "Jana", "Jaelynn", "Jaylee", "Jaycee");
        Predicate<String> p1 = s -> s.length() > 4;
        NameFilter p2 = new NameFilter() {
            public boolean test(String s) {
                return s.startsWith("Jay");
            }
            public boolean doTest() { return false; }
        };
        long count = strs.stream().filter(p1).filter(p2).count();
        System.out.println(count);
    }
}
```

What is the result?

*Options (select 1):*

1. Compilation fails due to invalid syntax of `NameFilter` interface.
2. Compilation fails due to invalid syntax of `p2` definition.
3. `0`
4. `2`
5. `3`

*Answer:*

The correct option is 4. The `StreamTest` class successfully compiles and the output will be `2`.

Option 1 is wrong. The `NameFilter` interface has `@FunctionalInterface` annotation, and a functional interface must have *single abstract method* (SAM). Since `NameFilter` provides implementation of the `test(...)` method inherited from `Predicate`, it must define own abstract method in order to compilation succeeded. Compiler considers `doTest()` method as SAM and interface successfully compiles.

Option 2 is wrong. The `p2` is defined as an instance of anonymous inner class, which implements `NameFilter` interface. In order to create instance successfully, at least all `abstract` methods must be implemented (i.e. `doTest()`), also the definition may optionally override `default` interface methods (i.e. `test(...)`). Note, as `Stream.filter(...)` accepts instance of `Predicate`, only `test(...)` methods will be invoked, and `doTest()` will be fully ignored.

So far we see that compilation succeeds. The pipeline has 2 filters subsequently applied:

- The `p1` will remove strings with length 4 or less. So, we have `"Jaelynn"`, `"Jaylee"`, and `"Jaycee"` passed further.
- The `p2` will remove strings which not start with `"Jay"`, and only `"Jaylee"` and `"Jaycee"` proceed to terminal operation which counts strings. The resulting count is 2 and option 4 is correct.

Based on the said above, options 3 and 5 are wrong.

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

## 8.2. Use lambda expressions and method references

**Question 080201**

Given the `Student` class:

```
class Student {
    String name;
    int height; // in centimeters
    public Student(String n, int h) {
        name = n;
        height = h;
    }
    static int orderByHeight(Student s1, Student s2) {
        return s1.height - s2.height;
    }
    public String toString() { return name + " " + height;  }
}
```

and the following code fragment:

```
Student[] arr = new Student[4];
arr[0] = new Student("George", 167);
arr[1] = new Student("Harry", 159);
arr[2] = new Student("Fred", 167);
arr[3] = new Student("Ron", 161);
... // line n1
System.out.println(Arrays.asList(arr));
```

Which code inserted at `line n1` position will print `[Harry 159, Ron 161, George 167, Fred 167]`?

*Options (choose 1):*

1.

```
Arrays.sort(arr, Student::orderByHeight(s1, s2));
```

2.

```
Arrays.sort(arr, Student::orderByHeight());
```

3.

```
Arrays.sort(arr, Student::orderByHeight);
```

4.

```
Arrays.sort(arr, (s1, s2) -> orderByHeight(s1, s2));
```

*Answer:*

The correct option is 3.

The `java.util.Arrays` class has the following method:

```
public static <T extends Object> void sort(T[] ts, Comparator<? super T> cmprtr) {
    ...
}
```

The `Comparator` is a Functional Interface -- it has a single abstract method (SAM):

```
@FunctionalInterface
public interface Comparator<T extends Object> {

    public int compare(T t1, T t2);

    ...
}
```

So, we can compose lambda which accepts 2 `Student` arguments and returns an `int`, and pass it to `sort(...)` method as the second parameter:

```
(s1, s2) -> Student.orderByHeight(s1, s2)
```

Or by refactoring to method reference (type: Reference to a `static` method) it will look like:

```
Student::orderByHeight
```

Option 1 is wrong: method reference never has parentheses: we do not invoke method, we provide reference to it, and compiler chooses proper method.

Option 2 is wrong: same as Option 1.

Option 4 is wrong: `orderByHeight` method invoked outside `Student` class, so reference to `Student` class is required. For example, this line would produce required output too:

```
Arrays.sort(arr, (s1, s2) -> Student.orderByHeight(s1, s2));
```

*Source:*

The Java Tutorials - Method References [https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html]

**Question 080202**

Given the code:

```
public class PrimeNumbersPrinter {
```

```
    public static void main(String[] args) {
        List<Integer> primes = Arrays.asList(new Integer[]{2, 3, 5, 7, 11, 13, 17, 19});
        PrimeNumbersPrinter p = new PrimeNumbersPrinter();
        ... // line n1
    }

    ... // line n2
}
```

Which two code changes when done at the same time will allow to print prime numbers to console?

*Options (choose 2):*

1. Replace `line n1` with the line:

```
primes.forEach(p::print);
```

2. Replace `line n1` with the line:

```
primes.forEach(p::print(i));
```

3. Replace `line n1` with the line:

```
primes.forEach(PrimeNumbersPrinter::print);
```

4. Replace `line n2` with the method:

```
public void print(Supplier<Integer> s) {
    System.out.println(s.get());
}
```

5. Replace `line n2` with the method:

```
public void print(Integer i) {
    System.out.println(i);
}
```

6. Replace `line n2` with the method:

```
public void print(List<Integer> l) {
    for (Integer i : l) {
        System.out.println(i);
    }
}
```

*Answer:*

The correct options are 1 and 5.

The `ArrayList` inherits the following `default` method from the `Iterable` interface:

```
```

```
public default void forEach(Consumer<? super T> cnsmr) {
    // ...
}
```

So, we should provide a `Consumer` functional interface which will be invoked with every value in the list.

The consumer's functional method should accept a value of the list members type (i.e. `Integer`) and produce no results (i.e. return `void`).

The only method which can be used for consumer is `public void print(Integer i)`. This makes option 5 correct.

`Consumer` is defined as lambda expression on `line n1`, in "traditional" way it could be defined as follows:

```
primes.forEach((i) -> p.print(i));
```

but we use method reference syntax (type: Reference to an instance method of a particular object) and equivalent line looks as follows:

```
primes.forEach(p::print);
```

This makes option 1 correct.

Option 2 is wrong: we never use parenthese in method reference syntax.

Option 3 is wrong: it shows reference to a static method syntax. It will not compile, unless the `print` method is declared as `static`.

Option 4 and 6 are wrong: the lambda expression on `line n1` expects a method which accepts a parameter of the same type as `ArrayList` members type (i.e. `Integer`).

*Source:*

The Java Tutorials - Method References [https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html]

**Question 080203**

Given the `Student` class:

```
class Student {
    public String name;

    public Student(String s) {
        name = s;
    }
}
```

and the following code:

```
public class PopulateStudentsProcessor {

    public static List<String> names = Arrays.asList(new String[]{"Tabitha", "Sadie",
"Tristram"});
    public static List<Student> students = new ArrayList<>();

    public static void main(String[] args) {
        for (String n : names) {
            // line n1
```

```
        }
    }

    public static void registerStudent(Function<String, Student> f, String n) {
        Student s = f.apply(n);
        students.add(s);
    }
}
```

Which line inserted at `line n1` position will allow to polulate list of students?

*Options (choose 1):*

1.

```
registerStudent((n) -> new Student(n), n);
```

2.

```
registerStudent(new Function(new Student(n)), n);
```

3.

```
registerStudent(Student::new(n), n);
```

4.

```
registerStudent(Student::new, n);
```

*Answer:*

The correct option is 4.

You should use this line as a hint:

```
public static void registerStudent(Function<String, Student> f, String n) {
```

As you can see, you are providing a `Function` Functional Interface as the first parameter.

The `Function` which you should pass, must accept a `String` as parameter and return a `Student`. And this requirement is supported by the only option which uses method (constructor) reference:

```
registerStudent(Student::new, n);
```

It provides `public Student(String s)` constructor reference, which complies with the requirement - accepts string, returns `Student` instance.

The "traditional lambda syntax" equivalent to option 4 would look very similar to option 1, but option 1 is wrong: it will not complie as lambda defines duplicated local variable:

```
for (String n : names) {
    registerStudent((String n) -> new Student(n), n);
}
```

```
```

This makes option 1 wrong.

This would be a valid `Function` code and could be used as a correct option:

```
registerStudent((String s) -> new Student(s), n);
```

Option 2 is wrong: the `Function` is an interface with a single abstract method (SAM) and can not be instantiated in this way.

Option 3 is wrong: constructor reference (as well as method reference) syntax never uses parentheses.

*Source:*

The Java Tutorials - Method References [https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html]

**Question 080204**

Given the `MegaLogger` class:

```
import java.util.Date;
import java.util.function.Supplier;

public class MegaLogger {
    public static void main(String[] args) {
        // line n1
    }

    public static void logEvent(Supplier<Date> tms, String event) {
        System.out.println(tms.get() + " : " + event);
    }
}
```

Which line inserted at `line n1` position will log the event at current timestamp?

*Options (choose 1):*

1.
```
logEvent(Date()::new, "Application started");
```

2.
```
logEvent(Date::new, "Application started");
```

3.
```
logEvent(Date->new(), "Application started");
```

4.
```
logEvent(new Date(), "Application started");
```

*Answer:*

The correct option is 2.

The `logEvent(Supplier<Date> tms, String event)` method signature defines that we should provide a **date object supplier** as the first parameter. The `Supplier` is a standard Java 8 Functional Interface which accepts no arguments and produces ("supplies") an instance of some `Object` (`Date` in our case). The option 2 implements this requirement, using constructor reference (A.K.A. method reference) syntax, as `java.util.Date` has a no-args constructor (and when it is used, the date object will contain the current timestamp).

Option 1 is wrong: it has invalid syntax, parentheses never used in method (constructor) reference syntax.

Option 3 is wrong: it is invalid syntax of lambda expression, compiler will complain - `Supplier` interface assumes blank parameters list, and we pass some varible named `Date`.

Option 4 is wrong. This option might be correct if method `logEvent` had signature like that:

```
public static void logEvent(Date tms, String event)
```

But in this case implementation would fail, as `Date` does not have `get()` method.

The corrected option 4 would implemented as constructor invocation (basically same as option 2), but using different syntax:

```
logEvent(() -> new Date(), "Application started");
```

*Source:*

The Java Tutorials - Method References [https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html]

## 8.3. Use built-in functional interfaces including `Predicate`, `Consumer`, `Function`, and `Supplier`

**Question 080301**

Which of the following Java functional interfaces from `java.util.function` package has `void` return type?

*Options (choose 1):*

1. `Predicate`
2. `Consumer`
3. `Function`
4. `Supplier`
5. `UnaryOperator`

*Answer:*

The correct option is 2. The `Consumer` represents an operation that accepts a single input argument and returns no result.

The signature of functional method of the `Consumer` interface is: `void accept(Object)`.


NOTE: you don't have to memorize exact signature of the method. If you use logic, you should understand that "`Consumer`" interface "consumes" some object, i.e. receives it as input parameter. But it does not require return anything. So, the most likely answer on this question is "`Consumer`", and other options are eliminated as only 1 option is required.

Option 1 is wrong. The `Predicate` represents a boolean-valued function of one argument.

Option 3 is wrong. The `Function` represents a function that accepts one argument and produces a result.

Option 4 is wrong. The `Supplier` represents a supplier of results.

Option 5 is wrong. The `UnaryOperator` represents an operation on a single operand that produces a result of the same type as its operand.

*Source:*

Package `java.util.function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html]

**Question 080302**

Which of the following Java functional interfaces from `java.util.function` package always produces a result of the same type as its operand?

*Options (choose 1):*

1. `Predicate`
2. `Consumer`
3. `Function`
4. `Supplier`
5. `UnaryOperator`

*Answer:*

The correct option is 5. The `UnaryOperator` represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of `Function` for the case where the operand and result are of the same type.

The signature of the functional method of the `UnaryOperator` interface is: `T apply(T t)`.


NOTE: You don't have to remember exact method signature. General description of the `UnaryOperator` interface should be enough to answer the question.

Option 1 is wrong. The `Predicate` represents a boolean-valued function of one argument.

Option 2 is wrong. The `Consumer` represents an operation that accepts a single input argument and returns no result.

Option 3 is wrong. The `Function` represents a function that accepts one argument and produces a result.

Option 4 is wrong. The `Supplier` represents a supplier of results.

*Source:*

Package `java.util.function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html]

**Question 080303**

Which of the following Java functional interfaces from `java.util.function` package may produce a `Boolean` object?

*Options (choose 3):*

1. `Predicate`
2. `Consumer`
3. `Function`
4. `Supplier`
5. `UnaryOperator`

*Answer:*

The correct options are 3, 4 and 5.

Here are examples of the functional interface definitions which return `Boolean` instance:

```
// Function which can create a Boolean based on String (e.g. "true")
Function<String, Boolean> fb = s -> {  return new Boolean(s);  };

// Supplier which always returns FALSE
Supplier<Boolean> sb = () -> Boolean.FALSE;
```

```
// UnaryOperator which negates the passed in value
UnaryOperator<Boolean> uob = b -> b ? Boolean.FALSE : Boolean.TRUE;
```

NOTE: You don't have to remember exactly all method signatures. If you apply logic, you can say that:

1. The `Consumer` only "consumes" and therefore it may not return `Boolean`
2. The `Predicate` always returns primitive `boolean`, not object `Boolean`
3. The other three options are correct ones.

Option 1 is wrong. The `Predicate` represents a boolean-valued function of one argument. It returns `boolean` primitive.

Option 2 is wrong. The `Consumer` represents an operation that accepts a single input argument and returns no result.

*Source:*

Package `java.util.function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html]

**Question 080304**

What statement about `java.util.function.BiPredicate` is true?

*Options (choose 1):*

1. Its functional method returns `boolean` primitive value.
2. Its functional method accepts two boolean primitive arguments.
3. It improves performance by avoiding unnecessary auto-boxing operation.
4. It inherits from `Predicate` functional interface.

*Answer:*

The correct option is 1. All `XxxxPredicate` intarfaces (`Predicate`, `BiPredicate`, `DoublePredicate`, `IntPredicate`) have a `test` method which returns `boolean` primitive.

`BiPredicate` interface:

```
package java.util.function;

@FunctionalInterface
public interface BiPredicate<T extends Object, U extends Object> {

    public boolean test(T t, U u);

        ...
}
```

Option 2 is wrong. The `BiPredicate` accepts two object arguments.

Option 3 is wrong. As all `XxxxPredicate` return primitive, none of them perform auto-boxing, this interface is not any specific flavour of predicate. There are some primitive-oriented `XxxxFunction` interfaces, like `ToIntFunction`, `ToLongFunction`, `ToDoubleFunction` which return primitive, instead of objects, and they indeed reduce overhead by avoiding of auto-boxing.

Option 4 is wrong. The listing above shows that `BiPredicate` does not inherit from any interface.

*Source:*

Package `java.util.function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html]

**Question 080305**

Which two Java functional interfaces from `java.util.function` package produce primitives?

*Options (choose 2):*

1. `Predicate`
2. `Consumer`
3. `Function`
4. `DoubleFunction`
5. `ToDoubleFunction`

*Answer:*

Correct options are 1 and 5.

The `Predicate` represents a `boolean`-valued function of one argument. It returns a `boolean` primitive.

The `ToDoubleFunction` represents a function that produces a `double`-valued result. This is the `double`-producing primitive specialization for `Function`:

```
package java.util.function;

@FunctionalInterface
public interface ToDoubleFunction<T extends Object> {

    public double applyAsDouble(T t);

}
```


Interfaces like `ToDoubleFunction`, `ToLongFunction`, `ToIntFunction` are provided in Java 8 API to avoid the negative performance consequences of auto-boxing.

Option 2 is wrong. The `Consumer` represents an operation that accepts a single input argument and returns no result.

Option 3 is wrong. The `Function` represents a function that accepts one object argument and produces an object result.

Option 4 is wrong. The `Supplier` represents a supplier of object results.

*Source:*

Package `java.util.function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html]

**Question 080306**

Given the code:

```
class Student {
    public String name;
    public Student(String n) {
        name = n;
    }
    public String toString() {
        return name;
    }
}

public class Faculty {
    public static void main(String[] args) {
```

```
        Student p = new Student("Harry");
        Student w = new Student("Ron");
        Student g = new Student("Hermiona");
        List<Student> list = Arrays.asList(new Student[]{p, w, g});

        Function<List<Student>, Student> f1 = arg -> {
            for (Student s : arg) {
                if (s.name.startsWith("H")) {
                    s.name.toUpperCase();
                    return s;
                }
            }
            return null;
        };
        f1.apply(list); // line 1
        System.out.print(list);
    }
}
```

Assuming that all imports are correct, what is the result?

*Options (choose 1):*

1. `[Harry, Hermiona]`
2. `[HARRY, HERMIONA]`
3. `[Harry, Ron, Hermiona]`
4. `[HARRY, Ron, Hermiona]`
5. Compilation fails at `line 1`.

*Answer:*

Correct option is 3.

Some points:

- This is a valid syntax to define `Function` instance and invoke it.
- The `Function` does not modify original list, so three items remains in the `list`.
- The `Function` does not modify objects from the list, the function `toUpperCase` returns uppercased `String`, but it not assigned and lost on the next line of code.

  The following code change would make Option 4 correct:

```
for (Student s : arg) {
    if (s.name.startsWith("H")) {
        s.name = s.name.toUpperCase();
        return s;
    }
}
```

Option 1 is wrong. The original list size is not modified.

Option 2 is wrong. The objects in the list are not modified.

Option 4 is wrong. The objects in the list are not modified.

Option 5 is wrong. The `Function` interface has one abstract method `apply` with the following signature:

```
public R apply(T t)
```

*Source:*

`java.util.function.Function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html]

**Question 080307**

Given the `Advertizer` class:

```
import java.util.function.Function;
public class Advertizer {
    public static void main(String[] args) {
        String j8 = " Java 8";
        String j9 = " Java 9";
        Function<String, String> greatifier = s -> s + " is great!";
        System.out.print(addMarketing(j8, greatifier));
        addMarketing(j9, greatifier);
        System.out.print(j9);
    }
    public static String addMarketing(String s, Function<String, String> f) {
        return f.apply(s);
    }
}
```

What is the output?

*Options (choose 1):*

1. `Java 8 Java 9`
2. `Java 8 is great! Java 9`
3. `Java 8 Java 9 is great!`
4. `Java 8 is great! Java 9 is great!`

*Answer:*

Correct option is 2.

Some points:

1. When fucntion invoked with `j8` it prints result which the function returns - i.e. the modified string.
2. Then fucntion invoked with `j9` and result is not assigned to any variable and lost on the next line of code. Then original `j9` variable is printed.

   It's not possible to modify `j9` without re-assigning, as `String` is immutable, so function can not modify in it's functional method.


Don't be confused by the `apply` method. Always check on exam if the result returned by `Function.apply` is assigned or lost.

*Source:*

`java.util.function.Function` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html]

**Question 080308**

Given the `City` class:

```
public class City {
    public String name;
    public int population;
    public City(String n, int p) {
        name = n;
        population = p;
    }
```

```
}
```

and the following code:

```
City city1 = new City("Minsk", 2002600);
// line 1
```

which code fragment inserted at `line 1` will print city information?

*Options (choose 1):*

1.

```
Consumer<City> cons = city2 -> city2.name + " - " + city2.population;
cons.accept(city1);
System.out.print(city1);
```

2.

```
Consumer<City> cons = city2 -> System.out.print(city2.name + " - " + city2.population);
cons.accept(city1);
```

3.

```
Consumer<City> cons = city2 -> city2.name + " - " + city2.population;
System.out.print(cons.accept(city1));
```

4.

```
Consumer<City> cons = city2 -> city2.name + " - " + city2.population;
System.out.print(cons);
```

*Answer:*

Correct option is 2. `Consumer` acts upon a value but returns nothing. It means a `void` method. We can use a consumer to call `System.out.print` or other `void` methods.

```
interface Consumer<T> {
    void accept(T t);
}
```

Options 1, 3, and 4 are wrong because `Consumer` implementation method returns `String`, so compilation fails. The syntax is the same as follows:

```
Consumer<City> cons = city2 -> { return city2.name + " - " + city2.population; };
```

```
```

or to understand you more obviously why the code compilation fails:

```
Consumer<City> cons = new Consumer() {
    @Override
    public void accept(Object t) {
        return city2.name + " - " + city2.population;
    }
};
```

*Source:*

`java.util.function.Consumer` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html]

**Question 080309**

Given the `Programmer` class:

```
public class Programmer {
    public String name;
    public double salary;
    public Programmer(String n, double s) {
        name = n;
        salary = s;
    }
}
```

and the following code:

```
public static void main(String[] args) {
    Programmer p1 = new Programmer("Volha Zaikina", 2100);
    Programmer p2 = new Programmer("Mikalai Zaikin", 2000);
    List<Programmer> list = new ArrayList();
    list.add(p1);
    list.add(p2);
    // line 1
}
```

which two code fragments independently inserted at `line 1` will increase salary by 10%?

*Options (choose 2):*

1.

```
Consumer cons = (Programmer p) -> p.salary = p.salary * 1.1;
list.forEach(cons);
```

2.

```
list.forEach(p -> { p.salary = p.salary * 1.1; } );
```

3.

```
Consumer<Programmer> cons = p -> p.salary = p.salary * 1.1;
list.forEach(cons);
```

4.

```
list.forEach(Programmer p -> { p.salary = p.salary * 1.1; return p; } );
```

*Answer:*

Correct options are 2 and 3. A `Consumer`, unlike the rest of the Java Functional Interfaces does not have a return value. It exists for the side-effects. It consumes the value provided to it. A `Consumer` might be used for writing to files or logging or updating a mutable data structure like a `List` or `Map`.

Option 1 is wrong, it will not compile, as as wrong syntax of `Consumer` declaration. It's declared as consumer of `Object`s.

The following change of Option 1 code would compile and run as expected:

```
Consumer cons = (p) -> ((Programmer)p).salary = ((Programmer)p).salary * 1.1;
list.forEach(cons);
```

Option 4 is wrong, as it explicitly returns value. If you remove the `return p;` it will compile and work as expected:

```
list.forEach((Programmer p) -> { p.salary = p.salary * 1.1; } );
```

*Source:*

`java.util.function.Consumer` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html]

**Question 080310**

Given the `Book` class:

```
public class Book {
    String title;
    int isbn;

    public Book(String t, int i) {
        title = t;
        isbn = i;
    }

    public String toString() {
        return title + ", ISBN " + isbn;
    }
}
```

and the following code:

```
Supplier<Book> s = ...; // code here
Book b = s.get();
```

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

which code fragment inserted instead of `...; // code here` placeholder will print `Exam 1Z0-817: Upgrade to Java SE 11 Developer Study Guide, ISBN 1234567890`?

*Options (choose 1):*

1. `Book("Exam 1Z0-817: Upgrade to Java SE 11 Developer Study Guide", 1234567890)::new`
2. `Book::new("Exam 1Z0-817: Upgrade to Java SE 11 Developer Study Guide", 1234567890)`
3. `(Book) -> new Book("Exam 1Z0-817: Upgrade to Java SE 11 Developer Study Guide", 1234567890)`
4. `() -> new Book("Exam 1Z0-817: Upgrade to Java SE 11 Developer Study Guide", 1234567890)`

*Answer:*

Correct option is 4. The `Supplier` can create the object of the `Book` class. Pass class name and `new` keyword while creating `Supplier`. Call `Supplier.get()` and you will get the object of that class.

Options 1 and 2 are wrong. Some hints which help you quickly eliminate them as wrong:

- Constructor reference may not define any parameters in declaration. Parameters are passed in later, when functional interface is invoked.
- `Supplier`'s functional method accepts no parameter, so `Supplier` can not be used in combination with `Book` class which does not have a default (no args) constructor.

Option 3 is wrong. Some hints which help you quickly eliminate it as wrong:

- This is an invalid syntax for parameters - in lambda expression Java type must be followed by parameter variable.
- `Supplier` accepts no parameter, only blank parentheses are allowed (see Option 4)

*Source:*

`java.util.function.Supplier` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html]

**Question 080311**

Given the code:

```
class Phone {
    public Phone() {
        System.out.print("I can dial.");
    }
}

class Smartphone extends Phone {
    public Smartphone() {
        System.out.print("I can run Java.");
    }
}

public class PhoneTest {
    public static void main(String[] args) {
        Supplier<Phone> sp = () -> new Phone(); // line 1
        Supplier<Smartphone> ss = Smartphone::new; // line 2
        Phone phone = ss.get();
    }
}
```

What is the result?

*Options (choose 1):*

1. `I can dial.`

2. `I can dial.I can run Java.`
3. Compilation fails at `line 1`.
4. Compilation fails at `line 2`.

*Answer:*

Correct option is 2. The `ss.get();` creates an instnce of `Smartphone` because `ss` is implemented as `Smartphone::new`. Java guarantees that the constructor method of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. Thus, if the first statement in a constructor does not explicitly invoke another constructor with `this()` or `super()`, Java implicitly inserts the call `super();` that is, it calls the superclass' constructor with no arguments. Therefore, when you create instance of `Smartphone`, it implicitly calls first constructor of `Phone` which prints "`I can dial.`", then rest of `Smartphone` constructor is invoked, which prints "`I can run Java.`"

Options 1 is wrong. It would be correct if you created instance of `Phone` class.

Option 3 and 4 are wrong. These are both valid syntaxes for `Supplier` declaration. Note for `line 2`: this `Supplier` syntax (constructor reference) is allowed only for classes with default (no args) constructors, as `Supplier`'s functional method does not accept parameters.

*Source:*

`java.util.function.Supplier` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html]

**Question 080312**

Given the code:

```
UnaryOperator<Integer> uo1 = i -> i + 2;
UnaryOperator<Integer> uo2 = i -> i * 2;
Integer i = 5;
System.out.println(uo1.apply(uo2.apply(i)));
```

What is the result?

*Options (choose 1):*

1. `5`
2. `7`
3. `10`
4. `12`
5. `14`

*Answer:*

Correct option is 4. First `uo2.apply(5)` is invoked which returns `10`. Then `uo1.apply(10)` which returns `12`.

*Source:*

`java.util.function.UnaryOperator` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/UnaryOperator.html]

**Question 080313**

Given the code:

```
UnaryOperator<Integer> uo = i -> "Java " + i; // line 1
uo.apply(7); // line 2
System.out.print(uo.apply(8));
```

What is the result?

*Options (choose 1):*

1. Compilation fails at `line 1`
2. Compilation fails at `line 2`
3. Runtime exception thrown at `line 2`
4. Code prints `Java 8`

*Answer:*

Correct option is 1. The `UnaryOperator` implementation code has incompatible return type. Assumed is `Integer`, but returned `String`.


- `UnaryOperator` is inheriting from `Function`, and as `Function` has exactly one parameter and can return some value.
- `UnaryOperator` in contrast to `Function` must return value of the same type as input parameter's type.

*Source:*

`java.util.function.UnaryOperator` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/UnaryOperator.html]

**Question 080314**

Given the code fragment:

```
// line 1
List<Double> list = new ArrayList<>();
list.add(12.34);
list.add(56.78);
list.replaceAll(f);
System.out.print(list);
```

Which code inserted at `line 1` position will print `[123.4, 567.8]`?

*Options (choose 1):*

1.
```
UnaryOperator<Double> f = d -> d * 10;
```

2.
```
Function<Double> f = d -> d * 10;
```

3.
```
Supplier<Double> f = d -> d * 10;
```

4.
```
Consumer<Double> f = d -> d * 10;
```

```
ToDoubleFunction f = d -> d * 10;
```

*Answer:*

Correct option is 1. The `UnaryOperator` implementation multiplies the passed in parameter by 10 and returns new value of the same type.

The question might look confusing, but you don't have to memorize API and know what parameter accepted in `List.replaceAll(...)` to answer correctly:

- Option 2 is wrong, as `Function` declaration always define 2 generics types: for input parameter and output value. Declaration with just a single type is invalid syntax. And you must remember that `Function` requires 2 parameterized types in declaration.
- Option 3 is wrong, as `Supplier` does not accept input parameters, and you must remember this.
- Option 4 is wrong, because `Consumer` does not return anything, this implementation will not compile. You must remember that functional method of `Consumer` has `void` return type.
- Option 5 is wrong. The `ToDoubleFunction` can accept some parameter of type `T` (note the the word "function" in the functional interface name), and when the type is not defined, and it's implied to be `Object`. But implementation multiplies input parameter by 10 and it may not be done with plain `Object`. So, this implementation is invalid and won't even compile.

*Source:*

`java.util.function.UnaryOperator` JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/UnaryOperator.html]

**Question 080315**

Given the code:

```
class Photo {

    private String name;
    private int size;

    public Photo(String n, int s) {
        name = n;
        size = s;
    }

    // getters and setters go here

    public String toString() {
        return name + " " + size;
    }
}

public class FileManager {

    public static void main(String[] args) {
        List<Photo> photos = new ArrayList<>();
        photos.add(new Photo("Sea view", 123));
        photos.add(new Photo("Mountain view", 456));

        // line 1

        System.out.print(photos.stream().filter(c).collect(Collectors.toList()));
    }
}
```

Which line, inserted at `line 1` position, will enable the code to print `[Mountain view 456]`?

*Options (choose 1):*

1.

```
Photo<Predicate> c = (Photo p) -> p.getSize() > 300;
```

2.

```
Predicate<Photo> c = p -> p.getSize() > 300;
```

3.

```
Predicate c = p -> p.getSize() > 300;
```

4.

```
Photo c = p -> p.getSize() > 300;
```

*Answer:*

The correct option is 2.

In Java 8 and later, the lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface.

In our example, object type (functional interface) is `Predicate`.

Since we invoke method `p.getSize()` inside lambda body, Java is expecting a data type of `Predicate<Photo>`, so the lambda expression must be of this type in the code.

The data type that these methods expect is called the target type. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found.

Options 1 and 4 are wrong, as plain objects (like `Photo` instances) can not be used in Stream API `filter(...)` method. Here is signature of the `Stream.filter` method, which expects `Predicate` as parameter:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Option 3 could be correct, in case we don't invoke `Photo`'s specific method (`getSize`), but invoked only `Object`'s generic methods (e.g. `equals(Object o)`).

For example, this code would compile and run fine:

```
Predicate c = p -> !p.equals(null);
```

result would be:

```
[Sea view 123, Mountain view 456]
```

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

**Question 080316**

Given the code:

```
public class Teacher {
    public Teacher(String lName){
        lastName = lName;
    }
    public String lastName;
    public String toString() { return lastName; }
}
```

```
interface MagicPredicate extends Predicate<Teacher> {
    static boolean validate(Teacher t) {
        return "Snape".equals(t.lastName);
    }
    default boolean check(Teacher t) { // line 1
        return "Hagrid".equals(t.lastName);
    }
}
```

```
public class University {
    public static void main(String[] args){
        Teacher t1 = new Teacher("Snape");
        Teacher t2 = new Teacher("Lupin");
        Teacher t3 = new Teacher("Hagrid");

        List<Teacher> teachers = List.of(t1,t2,t3);

        Predicate<Teacher> f = new MagicPredicate() {
            public boolean test(Teacher t) { // line 2
                return "Lupin".equals(t.lastName);
            }
        };

        System.out.print(teachers.stream().filter(f).findAny().get());
    }
}
```

Assuming all imports are correct, what is the output?

*Options (choose 1):*

1. `Snape`
2. `Lupin`
3. `Hagrid`
4. Compilation fails at `line 1`
5. Compilation fails at `line 2`

*Answer:*

The correct option is 2.

Some points:

- This is a valid example of `Predicate` (or more precisely `MagicPredicate` functional interface).

  It implemented as anonymous inner local class, and has exactly one abstract method. The other two methods in `MagicPredicate` interface are `default` and `static` method from interface - they may exist in functional interface, as long as Single Abstract Method (SAM) exists. So, options 4 and 5 are wrong.
- You have a hint in anonymous inner class implementation -- the only abstract method which class implements is `public boolean test(Teacher t)`, so you should realize that this is the functional interface's SAM and it will be invoked by Streaming API. So, options 1 and 3 are wrong.

- Of course, you should also remember `Predicate`'s SAM signature - `boolean test(T t)`
- Don't worry about understanding "`teachers.stream().filter(f).findAny().get()`" for now, as you are told that the code either prints something, or maybe fails in other places. So, you should not spend much time on this line of code.

*Source:*

The Java Tutorials - Lambda Expressions [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html]

## 8.4. Use primitive and binary variations of base interfaces of `java.util.function` package

**Question 080401**

Given the code:

```
Map<Integer, String> months = new HashMap<>();
months.put(1, "January");
months.put(2, "February");
months.put(3, "March");
// ... more months here
months.put(12, "December");

// line 1

months.forEach(f);
```

Which line inserted at `line 1` position will print numbers of all months which name starts with "`J`"?

*Options (choose 1):*

1. `Consumer<Map.Entry> f = (m) -> { if (m.getValue().startsWith("J")) System.out.println(m.getKey()); };`
2. `BiConsumer<Integer, String> f = (i, s) -> { if (s.startsWith("J")) System.out.println(i); };`
3. `BiConsumer<int, String> f = (i, s) -> { if (s.startsWith("J")) System.out.println(i); };`
4. `Consumer<String> f = (i, s) -> { if (s.startsWith("J")) System.out.println(i); };`

*Answer:*

The correct option is 2.

The `BiConsumer<T,U>` represents an operation that accepts two input arguments and returns no result. This is the two-arity specialization of `Consumer`.

```
package java.util.function;

@FunctionalInterface
public interface BiConsumer<T extends Object, U extends Object> {

    public void accept(T t, U u);
    ...
}
```

The `Map.forEach(BiConsumer<? super K,? super V> action)` method performs the given action for each entry in this map until all entries have been processed or the action throws an exception:

```
public default void forEach(BiConsumer<? super K, ? super V> bc) {
    // ...
}
```

Option 1 is wrong: it looks like a valid `Consumer` definition, but `m.getValue()` returns `Object`, not `String`, so `Object.startsWith()` will give compilation error.

Option 3 is wrong: generics can not be declared with primitives in Java.

Option 4 is wrong: The `Consumer` accepts one parameter, and the implementation shows two parameters: `f = (i, s)`. This is invalid syntax and won't compile.

*Source:*

The `Map` interface JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Map.html]

The `BiConsumer` interface JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/BiConsumer.html]

**Question 080402**

You are writing new reporting component for the Java application for mobile phone operator. The component should accept as input a mobile phone number and generate monthly bill for the customer. Mobile phone numbers are currently stored as `long` primitives in the existing part of the application. It is expected a high volume of bulk operations every month for your component and you should avoid high memory consumption by your code.

Which Java Functional Interface from `java.util.function` package will help you to implement the new requirement?

*Options (choose 1):*

1. `LongBinaryOperator`
2. `LongUnaryOperator`
3. `ToLongFunction`
4. `LongFunction`
5. `Function`

*Answer:*

The correct option is 4.

The `LongFunction<R>` represents a function that accepts a `long`-valued argument and produces a result. This is the `long`-consuming primitive specialization for `Function`.

```
package java.util.function;

@FunctionalInterface
public interface LongFunction<R extends Object> {

    public R apply(long l);
}
```

Option 1 is wrong: `LongBinaryOperator` represents an operation upon two `long`-valued operands and producing a `long`-valued result. The requirement asks you to process single `long` value and return an `Object`.

Option 2 is wrong: `LongUnaryOperator` represents an operation on a single `long`-valued operand that produces a `long`-valued result. You need to return an `Object`.

Option 3 is wrong: `ToLongFunction` represents a function that produces a `long`-valued result.

Option 5 is wrong: `Function` represents a function that accepts one `Object` argument and produces an `Object` result. You could implement something like `Function<Long, MonthlyBill>`, but it would require boxing `long` to `Long`, thus consuming extra memory, which is explicitly prohibited by the requirement.

*Source:*

The `LongFunction` interface JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/LongFunction.html]

**Question 080403**

You are hired to add new functionality to existing Java application for mobile phone operator. The new function should accept as input BASE64-encoded string with client's information (phone number and authorization code) and return number of free minutes left on the balance as `int` primitive.

Which Java Functional Interface from `java.util.function` package will help you to implement the new requirement?

*Options (choose 1):*

1. `IntFunction`
2. `ToIntBiFunction`
3. `ToIntFunction`
4. `IntSupplier`

*Answer:*

The correct option is 3.

The `ToIntFunction<T>` represents a function that produces an `int`-valued result. This is the `int`-producing primitive specialization for `Function`.

```
package java.util.function;

@FunctionalInterface
public interface ToIntFunction<T extends Object> {

    public int applyAsInt(T t);

}
```

Option 1 is wrong: `IntFunction<R>` represents a function that accepts an `int`-valued argument and produces an `Object` result. This is the `int`-consuming primitive specialization for `Function`. The requirement asks you to do the opposite: process `String` value and return an `int`.

Option 2 is wrong: `ToIntBiFunction` represents a function that accepts two `Object` arguments and produces an `int`-valued result. You need to process a single `String` (`Object`) parameter.

Option 4 is wrong: `IntSupplier` represents a supplier of `int`-valued results. This is the `int`-producing primitive specialization of `Supplier`. As any `Supplier` it does not accept any input parameters.

*Source:*

The `ToIntFunction` interface JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/ToIntFunction.html]

## Chapter 9. Parallel Streams

## 9.1. Develop the code that use parallel streams

### Question 090101

Given the code fragment:

```
Integer[] arr = {1, 2, 3, 4, 5};
List<Integer> lst = Arrays.asList(arr);
lst.stream().parallel().peek(System.out::print).count(); // line n1
```

What is the result?

*Options (choose 1):*

1. `12345`
2. Numbers in unpredictable order.
3. Runtime exception at `line n1`.
4. Compilation fails `line n1`.

*Answer:*

The correct option is 2.

Option 1 is wrong, it would be correct with sequential stream:

```
lst.stream().peek(System.out::print).count();
```

Option 2 is correct. The pipeline prints the elements of the list in a random order. Java Stream operations use internal iteration when processing elements of a stream. Consequently, when you execute a stream in parallel, the Java compiler and runtime determine the order in which to process the stream's elements to maximize the benefits of parallel computing unless otherwise specified by the stream operation.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

**Question 090102**

Given the code fragment:

```
Stream<Integer> s = Stream.of(new Integer[]{1, 2, 3, 4, 5});
List lst = s.parallel().collect(Collectors.toList()); // line n1
System.out.print(lst);
```

What is the result?

*Options (choose 1):*

1. `[1, 2, 3, 4, 5]`
2. Numbers in unpredictable order.
3. Runtime exception at `line n1`.
4. Compilation fails at `line n1`.

*Answer:*

The correct option is 1.

The `Collectors.toList` method specifies that the returned `Collector` adds elements to the list in **encounter order**.

There are two different kinds of "ordering" in stream pipelines:

- One kind is **encounter order**, which is defined in the streams documentation. A good way to think about this is the spatial or left-to-right order of elements in the source collection. If the source is a `List`, consider the earlier elements being to the left of later elements.
- There is also **processing order** (or **temporal order**), which is not defined in the documentation, but which is the time order in which elements are processed by different threads. If the elements of a list are being processed in parallel by different threads, a thread might process the rightmost element in the list before the leftmost element. But the next time it might not.

Even when computations are done in parallel, most `Collectors` and some terminal operations are carefully arranged so that they preserve encounter order from the source through to the destination, independently of the temporal order in which different threads might process each element.

Note that the `forEach(...)` terminal operation DOES NOT preserve encounter order. Instead, it is run by whatever thread happens to produce the next result. If you want something like `forEach(...)` that preserves encounter order, use `forEachOrdered(...)` instead.

Option 2 is wrong. It might be correct with the following code demonstrating `forEach(...)` and random processing order:

```
Stream<Integer> s = Stream.of(new Integer[]{1, 2, 3, 4, 5});
List<Integer> lst = new ArrayList();
s.parallel().forEach(a -> lst.add(a));
System.out.print(lst);
```

output (numbers in unpredictable order):

```
[3, 1, 4, 5, 2]
```

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Ordering [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html#Ordering]

**Question 090103**

Which statement is correct about Java Streams?

*Options (choose 1):*

1. Type of the stream (sequential or parallel) can not be changed once the stream is created from the source.
2. Parallel streams are faster than sequential.
3. Parallel streams are thread safe.
4. Parallel stream pipelines are lazy.

*Answer:*

The correct option is 4.

Option 1 is wrong. Most of the methods in the Streams API produce sequential streams by default. To produce a parallel stream from a collection such as a `List` or a `Set`, you need to call the `parallelStream()` method of the `Collection` interface. Use the `parallel()` method on a stream to convert a sequential stream into a parallel stream. Conversely, use the `sequential()` method on a stream to convert a parallel stream into a sequential stream.

Option 2 is wrong. Sometimes using parallel streams may give you worse performance.

The Streams API uses the Fork/Join framework to process parallel streams. The Fork/Join framework uses multiple threads. It divides the stream elements into chunks, each thread processes a chunk of elements to produce partial result, and finally, the partial results are combined to give you the result. Starting up multiple threads, dividing the data into chunks, and combining partial results takes up CPU time. This overhead is justified by the overall time to finish the task. For example, a stream of six objects is going to take longer to process in parallel than in serial. The overhead of setting up the threads and coordinating them for such small work is not worth it.

Option 3 is wrong. Race condition is still possible with parallel streams. It's developer's responsibility to coordinate access to shared objects.

Option 4 is correct. Both sequential and parallel streams' pipelines are lazy and require a terminal operation to start actual processing.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

## 9.2. Implement decomposition and reduction with streams

**Question 090201**

Given the code fragment:

```
List<Integer> list = List.of(3, 7, 5, 2);
... // line n1
System.out.print(sum);
```

Which line inserted at `line n1` position will print `17`?

*Options (choose 1):*

1. `int sum = list.stream().sum();`
2. `int sum = list.stream().reduce((a, b) -> a + b);`
3. `int sum = list.stream().reduce((a, b) -> a + b).sum();`
4. `int sum = list.stream().reduce(0, (a, b) -> a + b);`

*Answer:*

The correct option is 4.

Option 1 is wrong. Regular `Stream` (a stream of `Object`s) does not have `sum()` operation. It might be correct option for `IntStream`.

Option 2 is wrong. The `reduce(...)` with 1 parameter has a signature:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

There is no memorization here -- you can just imagine that `BinaryOperator` may never be invoked (e.g. when the stream is empty) and no value of type `T` will ever be produced, for this reason operation returns an `Optional`, not `T`. The `Optional` can not be assigned to `int` -- so this code won't compile.

This code can be fixed with the following change:

```
int sum = list.stream().reduce((a, b) -> a + b).get();
```

Option 3 is wrong. See the both explanations above. Another reason - `reduce` is a terminal operation and may not be followed by another terminal operation (`sum`).

Option 4 is correct. It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

Interface IntStream JavaDoc
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html]

**Question 090202**

Given the code fragment:

```
List<String> list = List.of("H", "e", "l", "l", "o");
... // line n1
System.out.print(s);
```

Which two lines inserted independently at `line n1` position will print `Hello`?

*Options (choose 2):*

1. `String s = list.stream().reduce("H", (a, b) -> a + b);`
2. `String s = list.stream().reduce("", a.concat(b));`
3. `String s = list.stream().reduce("", String::concat);`
4. `String s = list.stream().reduce((a, b) -> a.concat(b)).get();`

*Answer:*

The correct options are 3 and 4.

Option 1 is wrong. It provides initial value ("seed value") "`H`" and will produce "`HHello`" string.

Option 2 is wrong. It won't compile as it has invalid lambda syntax: there is no parameters defined. This option would be correct if rewritten:

```
String s = list.stream().reduce("", (a, b) -> a.concat(b));
```

Option 3 is correct. This reduction, also known as a **fold**, starts with a **seed value** ("", in our case), and applies the closure to the seed and the first element in the stream, taking the result and storing it as the accumulated value that will be used as the seed for the next element in the stream.

In other words, in a stream of strings such as "H", "e", "l", "l", and "o", the seed "" (blank string) is appended with "H" and the result ("H") is stored as the accumulated value, which then serves as the left-hand value in addition to serving as the next string in the stream (`"H".concat("e")`). The result (`"He"`) is stored as the accumulated value and used in the next concatenation (`"He".concat("l")`). The result (`"Hel"`) is stored and used in the next addition (`"Hel".concat("l")`), and the result is used in the final addition (`"Hell".concat("o")`), yielding the final result `"Hello"`.

Note that the type of closure accepted as the second argument to `reduce(...)` is a `BinaryOperator`, defined as taking two `String`s and returning a `String` result.

Option 4 is correct. It performs a reduction on the elements of this stream, and returns an `Optional`, and we retrieve the `String` value using the `get()` method.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

**Question 090203**

Given the code fragment:

```
Map.Entry<Integer, String> entry = Map.entry(0, "A");
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "B");
map.put(2, "C");
Map.Entry<Integer, String> result = map.entrySet().stream().reduce(entry,
        (a, b) -> Map.entry(a.getKey()*b.getKey(), a.getValue()+b.getValue()),
        (a, b) -> Map.entry(a.getKey()*b.getKey(), a.getValue()+b.getValue()));
System.out.println(result.getKey() + result.getValue());
```

What is the output?

*Options (choose 1):*

1. `0ABC`
2. `3ABC`
3. `3A`
4. `123A`

*Answer:*

The correct option is 1. The output is `0ABC`.

As you concluded from the answer option, the code compiles fine. The `Map.entry(...)` factory method for `Map.Entry` was introduced in Java 9.

The question test the `reduce` operation of `Stream` which takes 3 parameters:

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

The terminal operation performs a reduction on the elements of the stream, using the provided identity, accumulation and combining functions.

As you notice -- the operation returns `U`, not `Optional<U>`, because we provide identity and it will be returned in case of empty stream.

First, the `entry` (`<0,A>`) is taken and accumulated with first element of the map: keys are multiplied, resulting in `0 * 1 = 0`, and values are concatenated, resulting `"A" + "B" = "AB"`, and new `Map.Entry` is created by factory method.

Next, the new map entry result is taken and accumulated with second element of the map: keys are multiplied, resulting in `0 * 2 = 0`, and values are concatenated, resulting `"AB" + "C" = "ABC"`, and new `Map.Entry` is created by the factory method.

Finally, the key and the value of the resulting map entry printed: `"0ABC"`.

The `accumulator` function takes a partial result and the next element, and produces a new partial result. The `combiner` function combines two partial results to produce a new partial result. The `combiner` part is more important in case of parallel stream processing, where the input is partitioned, and for sequential processing you can use two-argument version of `reduce`.

*Source:*

Interface Stream JavaDoc [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html]

## Chapter 10. Language Enhancements

### 10.1. Use try-with-resources construct

#### Question 100101

Given the two resource classes:

```
class Reader implements AutoCloseable {
    public void close() throws Exception {
        System.out.print("Reader closed.");
    }
    public String read() {
        throw new NullPointerException("Read failed.");
    }
}
```

```
class Writer implements AutoCloseable {
    public void close() throws Exception {
        System.out.print("Writer closed.");
    }
    public void write() {
        throw new NullPointerException("Write failed.");
    }
}
```

and the code fragment:

```
try (Reader r = new Reader(); Writer w = new Writer()) {
    r.read();
    w.write();
} catch (Exception e) {
    System.out.print(e.getMessage());
}
```

What is the output?

*Options (choose 1):*

1. `Reader closed.Writer closed.`
2. `Writer closed.Reader closed.Read failed.`
3. `Read failed.`
4. `Reader closed.Read failed.`

*Answer:*

The correct option is 2. The output is: `Writer closed.Reader closed.Read failed..`

In this sample code the try-with-resources statement contains two declarations that are separated by a semicolon: `Reader` and `Writer`. The resources are initialized without exceptions by calling their default [no-args] constructors.

Next, when `r.read()` is called, a `NullPointerException` is thrown, immediately after that code proceeds to the end of the `try` block and closes both resources. Note that the `close()` methods of resources are called in the **opposite order of their creation**, the output will be `Writer closed.Reader closed.`

After that the `catch` block is executed which additionally prints the message from the `NullPointerException`, i.e. `Read failed.`

The full output to console will be: `Writer closed.Reader closed.Read failed.`

Based on the said above, options 1, 3, and 4 are wrong.

*Source:*

The Java Tutorials. The try-with-resources Statement
[https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html]

**Question 100102**

Given the two resource classes:

```
class Reader implements AutoCloseable {
    public void close()  { throw new NullPointerException("Close failed."); }
    public String read() { throw new NullPointerException("Read failed."); }
}
```

```
class Writer implements AutoCloseable {
    public void close() { System.out.print("Writer closed."); }
    public void write() { throw new NullPointerException("Write failed."); }
}
```

and the code fragment:

```
try (Reader r = new Reader(); Writer w = new Writer()) {
    r.read();
    w.write();
} catch (Exception e) {
    System.out.print(e.getMessage());
}
```

What is the output?

*Options (choose 1):*

1. `Writer closed.Read failed.`
2. `Writer closed.Read failed.Close failed.`
3. `Writer closed.Close failed.Read failed.`
4. `Close failed.Read failed.`

*Answer:*

The correct option is 1. The output is: `Writer closed.Read failed.`.

First, the executed code throws `NullPointerException` from the `read()` method. If an exception is thrown from within a try-with-resources block, any resource opened inside the parentheses of the try block will still get closed automatically. The throwing of the exception will force the execution to leave the `try` block, and this will force the automatic closing of the resource. The exception thrown from inside the `try` block will get propagated up the call stack, once the resources have been closed.

Next, resources are closed in the opposed to opened order. The writer is closed successfully and `Writer closed.` printed to console, after that a reader is closed and code throws `NullPointerException`. If an exception is thrown both from inside the try-with-resources block, and when a resource is closed (when `close()` is called), the exception thrown inside the `try` block will be propagated up the call stack. The exception thrown when the resource was attempted closed will be *suppressed*. So, to the catch block propagated the `NullPointerException("Read failed.");` exception and next output will be `Read failed.`

Based on the said above, options 2, 3, and 4 are wrong.

Option 2 would be correct with the following code, which additionally prints suppressed exception:

```
try (Reader r = new Reader(); Writer w = new Writer()) {
    r.read();
    w.write();
} catch (Exception e) {
    System.out.print(e.getMessage());
    System.out.print(e.getSuppressed()[0].getMessage());
}
```

*Source:*

The Java Tutorials. The try-with-resources Statement
[https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html]

**Question 100103**

Given the resources:

```
class Reader implements Closeable { ... }
```

```
class Writer implements Closeable { ... }
```

and the code fragment:

```
Reader r = new Reader(); Writer w = new Writer();
try (r;w) {
    r.read();
    w.write();
} catch (Exception e) {
    ...
} finally {
    ...
```

```
}
```

Which statement is true?

*Options (choose 1):*

1. `Reader` and `Writer` must implement `AutoCloseable` interface.
2. The `Reader r = new Reader(); Writer w = new Writer();` line must be placed inside `try(...)` block.
3. A programmer can close reources inside the `catch` block if needed.
4. The try-with-resourecs may not have `finally` block.

*Answer:*

The correct option is 1.

The try-with-resources construct can work with JDK built-in classes implementing `java.lang.AutoCloseable` interface, e.g. `java.beans.XMLEncoder`. You can also implement the `java.lang.AutoCloseable` interface in your own classes and use them with the try-with-resources construct. The `AutoClosable` interface has a single method called `close()`. Below is how the interface looks:

```
public interface AutoClosable {
    public void close() throws Exception;
}
```

Also, JDK provides another interface -- `java.io.Closeable`, which has the following definition:

```
public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

So, any class which implements `Closeable` (e.g. `java.io.FileInputStream`) is implementing `AutoCloseable` too, and this makes option 1 correct.

Option 2 is wrong. Before Java 9 a resource that is to be automatically closed must be created inside the parentheses of the `try` block of a try-with-resources construct. From Java 9, this is no longer necessary. If the variable referencing the resource is `final` or *effectively final*, you can simply enter a reference to the variable inside the `try` block parentheses. With this said, the `try (r;w) { ... }` construct is valid as of Java 9.

Option 3 is wrong. You can have a `catch` block in the a try-with-resources block just like you can in a standard try block. If an exception is thrown from within the `try` block of a try-with-resources block, the `catch` block will catch it, just like it would when used with a standard try construct. NOTE: before the `catch` block is entered, the try-with-resources construct will attempt to close the resources opened inside the `try` block. So, a programmer may not attempt to close resources inside the `catch` block.

Option 4 is wrong. You can add a `finally` block to a try-with-resources block. It will behave just like a standard `finally` block, meaning it will get executed as the last step before exiting the try-with-resources block -- after any `catch` block has been executed. NOTE: in case you throw an exception from within the `finally` block of a try-with-resources construct, all previously thrown exceptions will be lost.

*Source:*

The Java Tutorials. The try-with-resources Statement
[https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html]

## 10.2. Develop code that handles multiple Exception types in a single catch block

**Question 100201**

Given the exception classes:

```
class A extends Exception {}
```

---

```
class B extends A {}
```

```
class C extends RuntimeException {}
```

```
class D extends C {}
```

And assuming the `try` block is fully valid. Which `catch` block will compile successfully?

*Options (choose 1):*

1. `catch (A | D | NullPointerException ex) { }`
2. `catch (A | B ex) { }`
3. `catch (A | C | Exception ex) { }`
4. `catch (D | Throwable ex) { }`

*Answer:*

The correct option is 1.

Since Java 7 it is possible to catch multiple different exceptions in the same catch block. This is also known as multi catch.

The exception class names in the `catch` block are separated by the pipe character |.

NOTE: you cannot both catch exception `A` and exception `B` in the same time, if exception `B` is inherited (directly or indirectly) from `A`. Compiler will complain: `The exception B is already caught by the alternative A.`

Option 1 is correct. `A` and `D` are not in the same inheritance tree branch. They do subclass from `Exception`, but belong to different inheritance trees, Java compiler cannot state that for any `D` IS-A `A`, so they do not conflict each with other. Also, `NullPointerException` is also not a superclass or subclass of `A` or `D`, so compiler will not complain too.

Option 2 is wrong, `B` is a direct subclass of `A`, so only `A` handler is sufficient and Java compiler complains.

Option 3 is wrong, the third type in multicatch block is `java.lang.Exception` and both `A` and `C` are subclasses (`A` is direct subclass, and `C` is indirect subclass), so the compilation fails as both `A` and `C` IS-A `java.lang.Exception`.

Option 4 is wrong, `java.lang.Throwable` is a superclass for all exceptions, including `D`, as a result -- compilation fails.

*Source:*

Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking [https://docs.oracle.com/javase/8/docs/technotes/guides/language/catch-multiple.html]

**Question 100202**

Given the exception classes:

```
class BusinessException extends Exception {}
```

```
class A extends BusinessException {}
```

```
class B extends BusinessException {}
```

and the code fragment:

```
try {
    ...
} catch (A | B ex) { // line n1
```

```
    ex = new BusinessException(); // line n2
    throw ex;
}
```

Assuming the `try` block and surrounding method code is valid. What is the result?

*Options (choose 1):*

1. Compilation fails due to line n1
2. Compilation fails due to line n2
3. Compilation succeeds, and if the `try` block fails the `catch` block throws `A` or `B` exception
4. Compilation succeeds, and if the `try` block fails the `catch` block throws `BusinessException`

*Answer:*

The correct option is 2.

If a `catch` block handles more than one exception type, then the catch parameter is **implicitly final**. In this example, the catch parameter `ex` is implicitly final and therefore you cannot assign any values to it within the `catch` block.

Based on the said above, options 1, 3, and 4 are wrong.

*Source:*

Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking [https://docs.oracle.com/javase/8/docs/technotes/guides/language/catch-multiple.html]

**Question 100203**

Given the exception classes:

```
class BusinessException extends Exception {}
```

```
class A extends BusinessException {}
```

```
class B extends BusinessException {}
```

and the method:

```
public void doIt() throws A, B { // line n1
    try {
        boolean someCondition = ... // calculate the flag
        if (someCondition) throw new A(); else throw new B();
    } catch (Exception ex) { // line n2
        logger.log(ex);
        throw ex;
    }
}
```

Assuming the surrounding code is valid. What is the result?

*Options (choose 1):*

1. Compilation fails at line n1. To make it compilable change the line n1 as follows:

    ```
    public void doIt() throws Exception { // line n1
    ```

2. Compilation fails at line n1. To make it compilable change the line n1 as follows:

```
public void doIt() throws BusinessException { // line n1
```

3. Compilation fails at line n2. To make it compilable change the line n2 as follows:

```
} catch (A | B ex) { // line n2
```

4. Compilation succeeds without changes.

*Answer:*

The correct option is 4.

As of Java 7, some changes were done to exceptions handling. Compiler can analyze what exceptions may be thrown inside `try` block and although you declared `catch (Exception ex)` compiler can determine that only `A` or `B` exceptions are possible. Since you do not reassign `ex` variable, it will point to instance `A` or `B` and will be rethrown in the `catch` block. For that reason the `doIt()` method declaration properly lists possible exceptions which may result and the code will compile successfully without additional changes.

All other options propose the valid code which will compile, but all of them assume original code fails to compile. Because the original code is valid without additional changes, options 1, 2, and 3 are wrong.

*Source:*

Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking [https://docs.oracle.com/javase/8/docs/technotes/guides/language/catch-multiple.html]