LightHouseIT

Oracle认证考试　オラクル認定試験

- Oracle认证考试详细介绍　オラクル認定試験詳細
- 考点知识总整理　試験内容整理
- 考试知识点查漏补缺　試験内容の漏れを補おう
- Oracle模拟测试与讲解　オラクル模擬試験と解説

LightHouseIT

Oracle Certified Java Programmer, Silver SE 11 認定資格

- 官网概述:

概要：

「Java SE 11」は、2017年9月に発表された新しいリリース・モデルへの移行後初の LTS リリースであり、企業システムやクラウド・サービス、スマート・デバイスなどで活用されるアプリケーション開発の生産性向上に重点をおいています。この資格を取得することで、業界標準に準拠した高度なスキルを証明します。

Oracle Certified Java Programmer, **Silver** SE 11 認定資格は、Javaアプリケーション開発に必要とされる基本的なプログラミング知識を有し、上級者の指導のもとで開発作業を行うことができる開発初心者向け資格です。日常的なプログラミング・スキルだけでなく、さまざまなプロジェクトで発生する状況への対応能力も評価することを目的としています。Oracle Certified Java Programmer, **Silver** SE 11 認定資格を取得するためには、Java SE 11 Programmer I (1Z0-815-JPN) 試験の合格が必要です

試験詳細

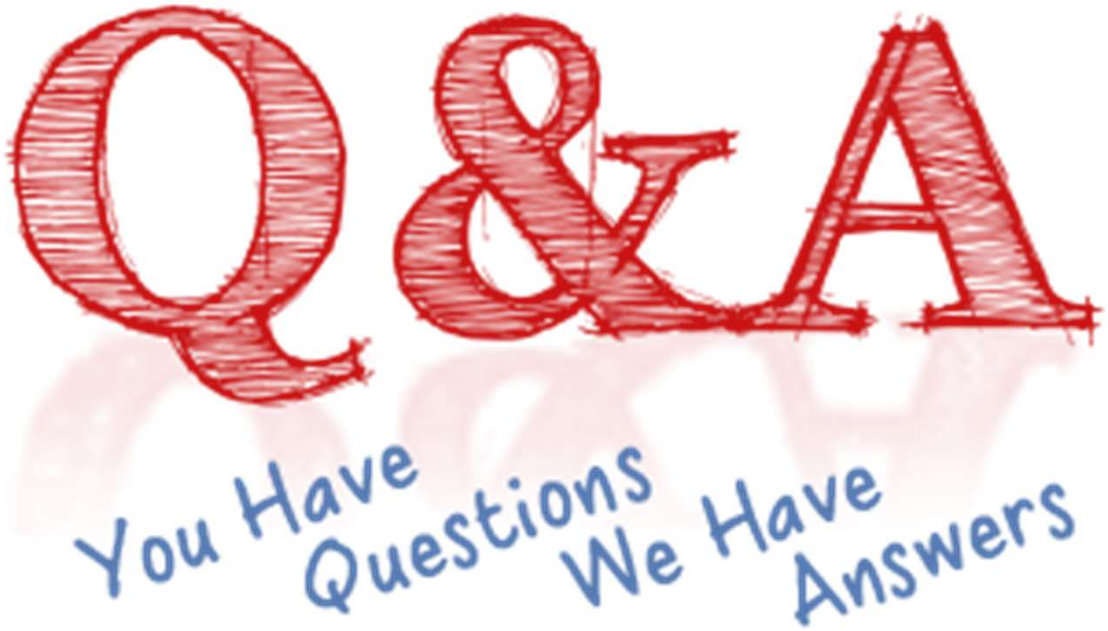
- 試験名： Java SE 11 Programmer I
- 試験番号： 1Z0-815-JPN
- 関連資格： Oracle Certified Java Programmer, Silver SE 11
- 受験料（税込）： 32,340 円
- 出題形式： 選択問題
- 試験時間： 180 分
- 出題数： 80 問
- 合格ライン： 63 %

报名须知

- 試験番号：1Z0-815-JPN
- 报名手顺：https://www.pearsonvue.co.jp/Clients/Oracle.aspx
- 考试语言一定要选择日语

- Oracle Certified Java Programmer 認定資格として認定されるためには**日本語試験**を受験いただく必要があります。英語試験で受験された場合、Oracle Certification Program としての認定対象とはなりますが、 Oracle Certified Java Programmer 資格として同時認定の対象とはなりませんのでご注意ください。

- 可以选择在宅线上考试或者去考场考试
- 选择在宅考的同学一定要仔细阅读对设备和网络等线上环境的要求。



LightHouseIT

考试范围

- Javaテクノロジと開発環境についての理解（Java开发环境和java技术）
- Javaの基本データ型と文字列の操作（Java基本类型和字符串操作）
- 配列の操作（数组操作）
- メソッドの作成と使用（方法的创建和调用）
- 継承による実装の再利用（继承）
- 例外処理（异常处理）
- 簡単なJavaプログラムの作成（创建java程序）
- 演算子と制御構造（运算符和条件循环语句）
- クラスの宣言とインスタンスの使用（类的声明与实例的使用）
- カプセル化の適用（封装）
- インタフェースによる抽象化（基于接口的抽象）
- モジュール・システム（module）

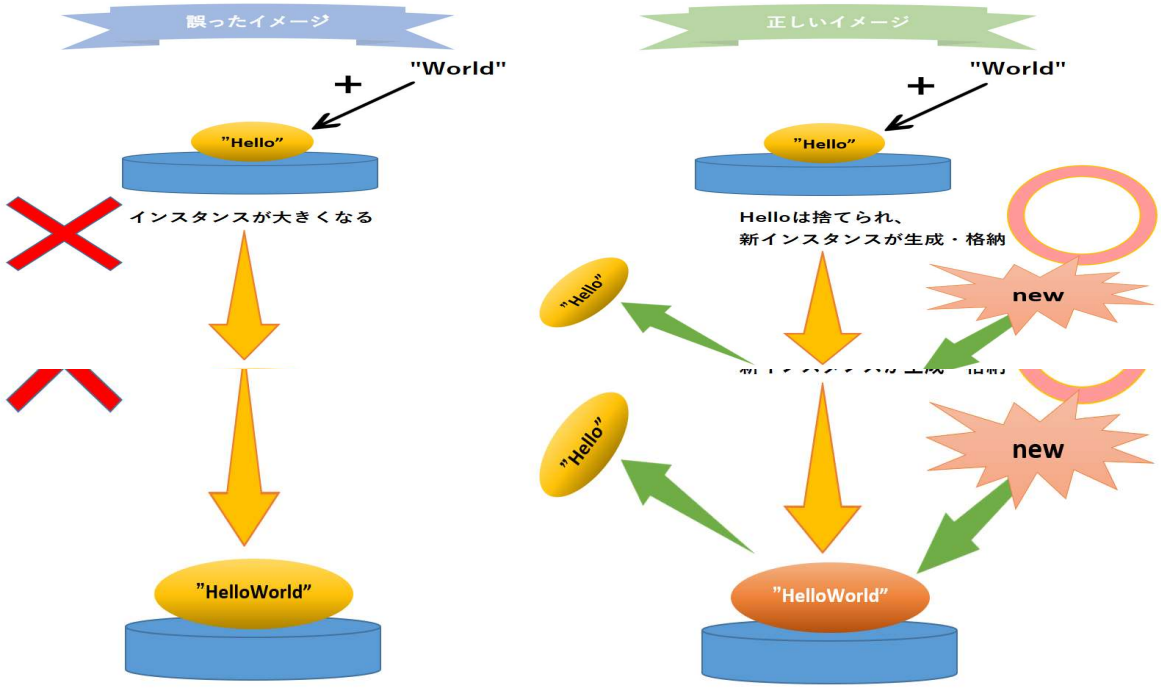
CheckList

- 考试常见日语词汇中文对照：LHP_Java_jpterms.pdf
- 详细考试内容checklist：LHP_Java_checklist.pdf

LightHouseIT

String类 不可变（immutable）

- String类的对象一旦创建出来，则不能改变其内容，只能重新创建一个新的对象。
- 如果需要可改变（mutable）的字符串[文字列]，可以使用StringBuilder类。



StringBuilder类 考点

- 可代替String类使用，并且可以对对象本身的字符串[文字列]进行改动。

```
java.lang.Object
 ↳ java.lang
    ↳ Class StringBuilder
```

- 创建对象：

```
// create a StringBuilder object
// with a String pass as parameter
StringBuilder str
    = new StringBuilder("AAAABBBCCCC");
```

- 尝试代码： Stringbuilder.java

StringBuilder类 方法

- 可以使用toString方法轻松转换成String类型的对象：

```
// print string
System.out.println("String = "
                   + str.toString());
```

- 提供append(boolean/char/char[]/double/String/...)方法，在现有字符串[文字列]后面追加多种类型的值

```
StringBuilder sbf1 = new StringBuilder("Geeks");
System.out.println("String Builder 1 = " + sbf1);

StringBuilder sbf2 = new StringBuilder("forgeeks ");
System.out.println("String Builder 2 = " + sbf2);

// Here it appends String Builder2 to String Builder1
sbf1.append(sbf2);

System.out.println("After appending the result is = "+sbf1);
```

```
String Builder = Geeksfor
After appending result is = Geeksforgeeks
```

StringBuilder类 方法

- capacity()，返回还可以增加的字符的数量
- insert(int offset, boolean/char/String…), 将第二个参数转换成字符串[文字列]并插入现有字符串[文字列]的offset指定位置
- replace(int start, int end, String str)，将start和end之间的字符串[文字列]替换成str
- reverse()将现有字符串[文字列]替换成逆序字符串[文字列]
- delete(int start, int end)，删除start和end之间的字符

Local Variable Type接口 考点

- JDK 10加入的新特性
- 内容：在创建对象的时候，只要进行了非null的初始化（non-null initialize），则前面可以用var来代替类型名称。（var属于类型名称而不是关键字）
- 例：
  - 考虑下面的代码：

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

  - 使用此新特性重写上面代码：

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

Local Variable Type接口

- 更多例子:

```
var list = new ArrayList<String>();    // infers ArrayList<String>
var stream = list.stream();            // infers Stream<String>
var path = Paths.get(fileName);        // infers Path
var bytes = Files.readAllBytes(path);  // infers bytes[]


List<String> myList = Arrays.asList("a", "b", "c");
for (var element : myList) {...}  // infers String


try (var input =
     new FileInputStream("validation.txt")) {...}
```

- 编译器会通过判断初始化的内容以及方式来自动判断变量类型。

Java main方法解读 考点

- 为了运行程序，至少需要在一个class里有一个main方法

public static void main(String[] args)
- String[] args是java命令行的参数，即通过命令行运行程序时，传递给程序的参数。

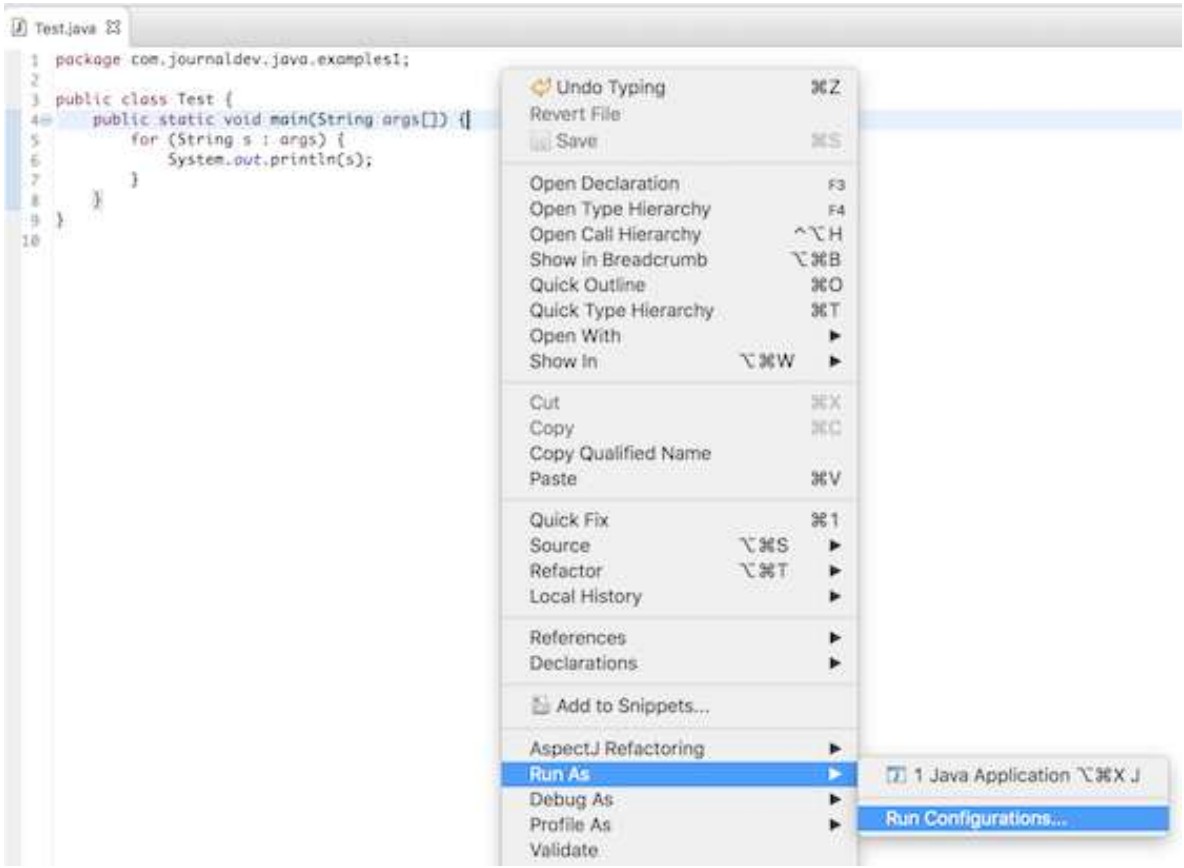
```
public class Test {

public static void main(String[] args){

    for(String s : args){
        System.out.println(s);
    }

    }
}
```

```
$ javac Test.java
$ java Test 1 2 3
1
2
3
$ java Test "Hello World" "Pankaj Kumar"
Hello World
Pankaj Kumar
$ java Test
$
```

- 尝试代码： Main.java (请打开命令行编译运行)

Java main方法解读

- 前面的代码也可以用编译器运行



函数式接口[関数型インタフェース]考点

- 定义：一个普通的接口，但包含一个且仅一个抽象的（未实现）的方法。（可以包含多个非抽象方法）
- 比如这是一个函数式接口[関数型インタフェース]：

```
public interface MyFunctionalInterface {
    public void execute();
}
```

- 这也是一个函数式接口[関数型インタフェース]，因为它仅包含一个抽象方法：

```
public interface MyFunctionalInterface2{
    public void execute();

    public default void print(String text) {
        System.out.println(text);
    }

    public static void print(String text, PrintWriter writer) throws IOException {
        writer.write(text);
    }
}
```

函数式接口[関数型インタフェース]

- 在创建函数式接口[関数型インタフェース]的实例时，必须要实现接口中的抽象方法。这就是函数式接口对实现类的约束。
- 该抽象方法可以使用lambda表达式进行实现。
- 比如我们之前学的创建Thread时使用到的Runnable接口就是一个函数式接口[関数型インタフェース]，因此我们必须要实现它的run（）方法：

```
// Java program to demonstrate Implementation of
// functional interface using lambda expressions

class Test
{
  public static void main(String args[])
  {

    // lambda expression to create the object
    new Thread(()->
       {System.out.println("New thread created");}).start();
  }
}
```

函数式接口考点： Function 接口

- 最常用的接口，功能很简洁：接收一个参数，返回一个值，只有一个apply方法
- Function接口源码，T，R分别表示参数和返回值的类型：

```
public interface Function<T,R> {

    public <R> apply(T parameter);
}
```

- 使用该接口的时候，仅需实现apply方法：

```
public class AddThree implements Function<Long, Long> {

    @Override
    public Long apply(Long aLong) {
        return aLong + 3;
    }
}
```

Java虽然不存在多继承，但使用接口可以实现多继承的效果

尝试代码： AddThree.java

函数是接口考点： Predicate接口

- 功能：接收一个参数，返回一个布尔值，只有一个test方法。
- 源码：

```
public interface Predicate {
   boolean test(T t);
}
```

- 实现方式：

```
import java.util.function.Predicate;

public class CheckForNull implements Predicate<Object> {
    @Override
    public boolean test(Object o) {
        return o != null;
    }
}
```

- 或者使用lambda表达式：

```
Predicate<Object> predicate = (value) -> value != null;
```

尝试代码： CheckForNull.java

函数是接口考点：Supplier接口

- 功能：不接收任何参数，返回一个结果，只有一个get方法
- 源码：

```
public interface Supplier<T>{
    T get();
}
```

- 用lambda表达式实现，则不需要implements关键字：

```
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        // This function returns a random value.
        Supplier<Double> randomValue = () -> Math.random();

        // Print the random value using get()
        System.out.println(randomValue.get());
    }
}
```

- 尝试代码：SupplierExample.java

函数是接口考点：Consumer接口

- 功能：和Supplier正好相反，接收一个参数，没有返回值，抽象方法名为accept（T t）
- 实现：

```
Consumer<Integer> consumer = (value) -> System.out.println(value);
```

- 尝试代码：ConsumerExample.java

Module[モジュール] 考点

- Java9引入的一种java代码的集合形式的单位： Java Platform Module System
- 一个module可以包含一个或多个包。
- 一个java module需要有一个独一无二的名字比如：
  - com.xxx.mymodule
- 这部分可以通过做考试题进行学习

Module Descriptor (module-info.java)

- 每个module需要有一个module-info.java文件来描述此module，为其定义一些东西。
- 存放路径： src/com.xxx.mymodule/module-info.java
- 主要可以定义两个内容：
  - 哪些包对外部可见
  - 此module依存于哪个module

```
module com.xxx.mymodule {

}
```

Module Exports

- exports关键字指定哪些包对module的外部可见

```
module com.xxx.mymodule {
    exports com.xxx.mymodule.app;
    exports com.xxx.mymodule.util;
}
```

Module requires

- 定义此module依靠哪个module

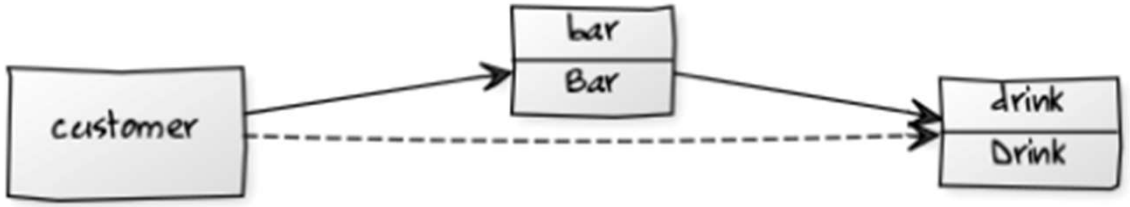
```
module com.xxx.mymodule {
    exports com.xxx.mymodule.app;
    exports com.xxx.mymodule.util;
    requires javafx.graphics;
}
```

- mymodule可以访问javafx.graphics里面exports的包

Module requires transitive

- 比如有三个module分别是customer, bar, drink
- 如果bar requires transitive drink并且customer requires bar
- 则customer 间接 requires drink（不需定义customer requires drink）

```
module bar {
    requires transitive drink
}
```



编译运行module

- 需要java9及以上版本
- 例:
  - 将编译结果存入out文件夹里:
    - javac -d out --module-source-path src/main/java --module com.xxx.mymodule
  - 编译完运行:
    - java --module-path out --module com.xxx.mymodule/com.xxx.mymodule.Main

LightHouseIT

通过Google Form进行测试