注意事项：

- 本次测验会在同一个工程下进行工作。
- 全程请使用 Git 进行代码管理。小步提交。
- 每做完一问，请找到 Coach 进行评定。合格之后再进行下一问。
- 请严格和 AC 的要求保持一致！需要保证之前每一问的结论仍然是正确的。

题目1：创建工程

- AC1：请创建一个 Gradle Java 工程。其中包含代码和测试。
- AC2：请书写一个 Greeting 类，其中包含方法 greet。该方法返回 “Hello world!”。书写测试确保该方法的行为是正确的。

题目2：根据用户注册的无依赖类型创建对象

- AC1：请删除题目 1 中的代码和测试。
- AC2：定义一个接口 ContainerBuilder。其中有如下的函数定义：

```
public interface IoCContext {
  void registerBean(Class<?> beanClazz);
  <T> T getBean(Class<T> resolveClazz);
}
```

若现在有含有默认构造函数的类型 MyBean。那么我应当能够通过如下的方式创建其实例：

```
IoCContext context = new IoCContextImpl();
context.registerBean(MyBean.class);
MyBean myBeanInstance = context.getBean(MyBean.class);
```

- AC3：在 registerBean 过程中，如果传入的 beanClazz 为 null 或者并不是能够实例化的 Class，则抛出 IllegalArgumentException，并带有不同的信息提示：（beanClazz is mandatory 以及 $bean full name$ is abstract.）
- AC4：在 registerBean 过程中，如果传入的 beanClazz 代表的类型没有默认构造器，则抛出 IllegalArgumentException 并带有信息提示（$bean full name$ has no default constructor.）。
- AC5：在 registerBean 过程中，如果传入的 beanClazz 已经被注册过了，则该方法直接返回，并不报告错误。

- AC6：如果在 getBean 过程中，若 resolveClazz 为 null 则抛出 IllegalArgumentException。若当前 IoCContext 对象并未 register 指定的 resolveClazz，则抛出 IllegalStateException。
- AC7：如果在 getBean 过程中，在调用的构造函数中抛出了异常，则该方法应当继续抛出该异常。
- AC8：一旦开始 getBean 就不能再调用 registerBean 否则将抛出 IllegalStateException

题目3：区分接口类型和实现类型

- AC1：现在我们需要向 IoCContext 接口中添加额外的注册方法以区分接口类型和实现类型：

  ```
  <T> void registerBean(Class<? super T> resolveClazz, Class<T> beanClazz);
  ```

  那么我现在应当可以采用如下的方式来创建对象：

  ```
  context.registerBean(MyBeanBase.class, MyBean.class);

  // 创建了 MyBean 的实例
  MyBeanBase myBeanInstance = context.getBean(MyBeanBase.class);
  ```

- AC2：registerBean 的两个重载若注册了相同的 resolveClazz。那么后注册的类型会覆盖前一个注册的类型。

  ```
  context.registerBean(MyBeanBase.class, MyBean.class);
  context.registerBean(MyBeanBase.class, MyBeanCooler.class);
  // 得到 MyBeanCooler
  MyBeanBase myBeanInstance = context.getBean(MyBeanBase.class);
  ```

- AC3：新的 registerBean 重载应当仍然遵守 题目1 中的行为。

题目4：创建有依赖的类型（不考虑继承）

- AC1：假设 MyBean 依赖于 MyDependency，且我们使用如下的方式声明依赖：

  ```
  class MyBean {
  ```

```
  @CreateOnTheFly
  private MyDependency dependency;
}
```

那么我们可以使用如下方式创建 MyBean 的实例并同时创建 MyDependency 实例并赋值给 MyBean.dependency 字段。我们的依赖目前仅仅支持字段。

- AC2：如果 MyBean 的任何一个依赖并没有事先进行注册，则在 getBean 时应当抛出 IllegalStateException。

题目5：创建有依赖的类型（考虑继承）

- AC1：在创建类型的实例时，不但应当实例化本类型中标记为 @CreateOnTheFly 的字段，而且还应该在之前恰当的初始化父类中标记为 @CreateOnTheFly 的字段。并且按照先父类再子类的顺序初始化字段。
- AC2：之前的所有的异常情况在这里都应当沿袭。

题目6：管理当前容器内的对象生命周期

- AC1：若当前 IoCContext 实例使用 getBean 方法创建了对象的实例，而该对象恰好实现了 AutoCloseable 接口。则当 IoCContext 实例在 close 时也调用该对象的 close() 方法。本问中，IoCContext 接口的声明变为了：

```
public interface IoCContext extends AutoCloseable {
  void registerBean(Class<?> beanClazz);
  <T> T getBean(Class<T> resolveClazz);
}
```

- AC2：在 IoCContext 的 close 方法调用时，应当按照创建顺序的反向顺序依次调用 close() 方法（如果实现了 AutoCloseable 的话）。
- AC3：就算是某一个 close() 方法抛出了异常，也必须保证调用所有实现了 AutoClosable 接口的实例的 close() 方法。并在所有调用完成之后抛出第一个出现的异常。