Exercise 6: Data Abstraction

(Deadline for uploading solutions: July 16, 2021, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the directory structure and the templates that must be used for the submission**.

The directory structure is:

```
Exercise6/
├─ Task1/
│  └─ Task1.zip
├─ Task2/
│  ├─ validity.csv
│  └─ behavior.csv
└─ Task3/
   ├─ Goku.txt
   ├─ ChiChi.txt
   ├─ Gohan.txt
   ├─ MisterSatan.txt
   └─ Videl.txt
```

The submission must be compressed in a zip file using the given directory structure. The name of files and directories must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (**not rar**, 7z, etc.).
- Your zip file should contain exactly one top-level directory `exercise6/`, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions in them.

1 Task I (50% of total points of the exercise)

This task is about **static and dynamic method binding**. We remind you that an object has a static type and might have a different dynamic type, such as in the following Java example, where `obj` has static type `A` and dynamic type `B`.

Different static and dynamic type

```
Class A {};
Class B extends A {};
A obj = new B();
```

For this task, we consider a simple model of a an **object-oriented, class-based language with single inheritance and method overriding**. You do not have to care about the syntax of this language, but only reason about abstract entities: classes, methods, and objects. Classes can inherit from up to one superclass, i.e., they are arranged in a hierarchy. A class may have methods. Instantiating a class yields an object, on which one can call methods. We consider two variants of our language, one using static method binding and the other using dynamic method binding. In the given code, these concepts are implemented in three classes: `PLClass`, `PLMethod`, `PLObject`. An object has both a statically declared type and a dynamic type. These types are set when instantiating the `PLObject` class.
You have to implement two methods of `PLObject`:

- `bindMethodStatically(String methodName)`
- `bindMethodDynamically(String methodName)`

These methods should return the `PLMethod` object that represents the method to call, depending on whether the language uses static or dynamic method binding.
For example, have a look at the given test case, which creates two classes, two methods, and four objects. The `setUp` method of the test class defines the classes and methods. Class `A` is superclass of class `B`, and there is a method called `foo` in each class. In other words, `B.foo` is overriding `A.foo`. Each of the `test*` methods creates one object with specific static and dynamic types, and then calls `foo` on this object. The call to `bindMethod*` is supposed to return the appropriate method that will be called.
For simplicity, you can assume the following about the language:

1. Methods do not have parameters
2. No method overloading
3. No two methods with the same name in a single class
4. No visibilities (protected, public, private, etc.)
5. No multiple inheritance
6. No two class with the same name

For testing your solution, we strongly recommend to implement additional tests, e.g., with more classes, more methods, and different ways to combine them.
Evaluation Criteria: Your solution implementation will be tested against additional tests.

2 Task II (25% of total points of the exercise)

This task is about **class vs. instance data**. You are provided with a piece of Java code that contains a class with `static` and non-static fields and methods. Your task is split into two parts:

1. In the first subtask, you will look at **validity at compile time** of static and non-static field accesses and method calls. That is, for marked lines in the source code below, you will determine whether the line is

   - an **error**, that is, the program cannot be executed because this construct is a semantic error and hence not valid Java code. Java compilers would have to reject a program containing this line.
   - a **warning**, because a static member is accessed through a non-static object reference. Even though this is not an error, it could lead to confusion for other developers reading the code and is commonly accepted as *bad style*. Many Java compilers will issue a warning to help developers avoid such constructs (even though they are not required to do so).
   - **valid**, that is, the construct is well-defined and considered idiomatic in Java.

   For each respective marked line in the source code below, please submit your answer in the file *Exercise6/Task2/validity.csv*. The rows correspond to the lines marked with `Validity` in the source code below. Submit your answers in the second column of the CSV file as `error`, `warning`, or `valid`.

2. In the second subtask, you will look at the program's **behavior at runtime**. For that, first assume that all lines of the first subtask which result in an *error* are *removed from the program*. Valid lines and lines with warnings are kept in the program. Then, at each of the marked lines in the code below, you shall give the current value of certain fields. Please submit your answers in the file *Exercise6/Task2/behavior.csv*. The rows correspond to the lines marked with `Behavior` in the source code below. Submit your answers as the integer (e.g., `0` or `42`) that is passed to the `value` function in the respective lines.

The source code of the program is the following:

```java
class MercedesBenzArena {
  static int totalMatchDisputed = 0;
  int vfbStuttgartHomeGoals = 0;

  static void createTicketsNewMatch() {
      this.totalMatchDisputed += 1;                 // Validity 1
      MercedesBenzArena.totalMatchDisputed += 1;    // Validity 2
  }

  void stuttgartScore() {
      this.vfbStuttgartHomeGoals += 1;                      // Validity 3
      MercedesBenzArena.vfbStuttgartHomeGoals += 1;         // Validity 4
  }
}


// Somewhere else in the code, e.g., in a main method:
MercedesBenzArena a = new MercedesBenzArena();
MercedesBenzArena b = new MercedesBenzArena();

a.stuttgartScore();                             // Validity 5
a.createTicketsNewMatch();                      // Validity 6
MercedesBenzArena.stuttgartScore();             // Validity 7
MercedesBenzArena.createTicketsNewMatch();      // Validity 8

value(a.totalMatchDisputed);         // Behavior 1
value(a.vfbStuttgartHomeGoals);      // Behavior 2
value(b.totalMatchDisputed);         // Behavior 3
value(b.vfbStuttgartHomeGoals);      // Behavior 4

MercedesBenzArena.totalMatchDisputed += 1;          // Validity 9
MercedesBenzArena.vfbStuttgartHomeGoals += 1;       // Validity 10
a.totalMatchDisputed += 1;       // Validity 11
a.vfbStuttgartHomeGoals += 1;    // Validity 12
```

```

value(a.totalMatchDisputed);       // Behavior 5
value(a.vfbStuttgartHomeGoals);    // Behavior 6
value(b.totalMatchDisputed);       // Behavior 7
value(b.vfbStuttgartHomeGoals);    // Behavior 8

b.totalMatchDisputed += 1;     // Validity 13
b.vfbStuttgartHomeGoals += 1; // Validity 14

value(a.totalMatchDisputed);       // Behavior 9
value(a.vfbStuttgartHomeGoals);    // Behavior 10
value(b.totalMatchDisputed);       // Behavior 11
value(b.vfbStuttgartHomeGoals);    // Behavior 12
```

Evaluation Criteria: Your solution will be compared against the correct choice of compile time validity and correct values at runtime.

3 Task III (25% of total points of the exercise)

This task is about **virtual (function) tables or vtables**. You are provided with a piece of C++ code that contains several classes with virtual and non-virtual methods. Your task is to give the vtable layout for each of the classes. For constructing the vtables, assume that the compiler works as described in the lecture. In particular, assume that vtables only contain pointers to virtual methods declared in the source code.[1] Also assume that the pointers are layed-out in the declaration order of the methods and that methods of the superclass come before methods of the derived class.
To submit your answers, please write the vtable entries in the plain text files *Exercise6/Task3/Class*.txt*, where the filename corresponds to the class declaration in the source code below. Write one line per pointer in the vtable. Specify to which method implementation a pointer points to as `ClassName::methodName`. If a class has no vtable, you must submit a single line in the corresponding file with the contents `no vtable`.
As an example of a simple vtable, consider the following `Example` class. Its vtable would contain one pointer for the method `Example::methodExample1`. The submitted answer file would hence contain only a single line `Example::methodExample1`.

Example Class

```cpp
class Example {
public:
    virtual void methodExample1() { /* ... */ };
    void methodExample2() { /* ... */ };
};
```

The source code with the class definitions for the task follows below. The code makes use of the C++11 `override` keyword to clarify when a derived class polymorphically overrides a virtual method of the base class.

```cpp
class Goku {
public:
    virtual void kaioken() { /* ... */ };
    virtual void kamehameha() { /* ... */ };
};

class ChiChi : Goku{
public:
    virtual void nothing() { /* ... */ };
};

class Gohan: ChiChi {
public:
    virtual void superSayan() { /* ... */ };
};

class MisterSatan {
public:
    void sleep() { /* ... */ };
    virtual void fight() { /* ... */ };
};

class Videl: MisterSatan {
public:
    void fight() override { /* ... */ };
    virtual void fightALot() { /* ... */ };
};
```

Evaluation Criteria: Your solution will evaluated against the correct vtables for each class.

[1] Real C++ compilers generate additional pointers in vtables for more advanced language features, for example runtime type information (RTTI) or virtual inheritance. For more information, see `https://stackoverflow.com/questions/5712808/what-is-the-first-int-0-vtable-entry-in-the-output-of-g-fdump-class`. Additionally, be advised that real-world C++ code should use virtual destructors if objects of a base class are going to be used polymorphically, see `https://stackoverflow.com/questions/461203/when-to-use-virtual-destructors`. Neither of those topics is relevant to the exercise, but mentioned here for completeness.