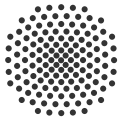

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2020

Exercise 6: Data Abstraction

(Deadline for uploading solutions: July 17, 2020, 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.csv
├── task2.csv
├── task3/
│   ├── validity.csv
│   └── behavior.csv
└── task4/
    ├── ClassA.txt
    ├── ClassB.txt
    ├── ClassC.txt
    ├── ClassD.txt
    └── ClassE.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 (25% of total points of the exercise)

This task is about **method binding**. You are provided with two code excerpts. The first excerpt defines a parent class and multiple derived classes. The second excerpt performs a sequence of assignments and method calls on instances of those classes. The code is written in a toy language with class-based object orientation and implicit references, similar to Java, and an explicit `virtual` keyword, similar to C++. A method in a subclass overrides the method of the same name in the superclass. Methods that are not overriden are inherited from the superclass. Assume all methods are public by default.

Your task is to indicate which method is being called at runtime in the second excerpt, given the class definitions of the first excerpt, under three different semantics:

- **Dynamic binding**: The language always uses dynamic method binding, similar to Java. The `virtual` keyword has no meaning for this semantics.
- **Static binding**: The language always uses static method binding. The `virtual` keyword has no meaning for this semantics.
- **Depends on `virtual`**: The language uses static binding by default, but dynamic binding for all methods that override a method in a superclass that is marked with `virtual`. This is similar to C++. [1]

To submit your answer, please fill the CSV file *exercise6/task1.csv*. The rows correspond to the marked lines in the *Method Calls* excerpt. The columns correspond to the three semantics above. To specify which method is called, use the following format: `ClassName.methodName`. Make sure to use the name of the classes, not of the objects. Also, use just the method name and do not suffix it with parentheses. The following is a sample CSV solution for a (hypothetical) code snippet with just one call:

```
              , Dynamic binding,     Static binding,   Depends on virtual
Method Call 1, DerivedClass.method, BaseClass.method, BaseClass.method
```

Please make sure that your CSV file is valid. For example, it should not contain additional commas or tabs or semicolons as delimiters. Try to stay away from Excel or LibreOffice, or double check your files with a plain-text editor after editing them in said programs.

The classes are defined as follows:

Class Definitions

```
class Employee {
    void greeting() {
        print("Hello");
    }
    virtual void duty() {
        print("work");
    }
}
class Permanent extends Employee {
    void greeting() {
        print("Hello everyone!");
    }
}
class Intern extends Employee {
    void greeting() {
        print("Hi, what's up?");
    }
    void duty() {
        print("fetch coffee");
    }
}
class Manager extends Permanent {
    void duty() {
        print("lots of meetings");
    }
}
```

[1] Since C++11, there are also additional `final` and `override` keywords, but for simplicity our toy language does not use them.

For the marked method calls in the following excerpt, submit your answer as described above.

Method Calls

```
Employee harry = new Employee();
Permanent rebecca = new Permanent();
Intern john = new Intern();
Manager lisa = new Manager();

harry.greeting();   // Method Call 1
rebecca.greeting(); // Method Call 2
rebecca.duty();     // Method Call 3
john.duty();        // Method Call 4
lisa.duty();        // Method Call 5

Employee e1 = harry;
Employee e2 = rebecca;
Employee e3 = john;
Permanent p = lisa;

e1.duty();          // Method Call 6
e2.duty();          // Method Call 7
e3.duty();          // Method Call 8
p.duty();           // Method Call 9

e1.greeting();      // Method Call 10
e2.greeting();      // Method Call 11
e3.greeting();      // Method Call 12
p.greeting();       // Method Call 13
```

Evaluation Criteria: Your solution will be compared against the correct method that gets called under the respective semantics.

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

This task is about the **visibility** of fields and methods in object-oriented languages. You are provided with a piece of code that defines several classes with fields and methods of unspecified visibility, and later (outside of those classes) performs computations on the fields and method calls on instances of the previously defined classes. The code is written in a toy language inspired by Java.

Your task is to assign each member in the class definitions a single visibility specifier, such that the overall code is valid, i.e., that all fields and methods can be accessed correctly according to their visibility specifier. Following the "principle of least priviledge", you must choose the **minimal** visibility for each member, i.e., you must assign the visibility permitting access to the least number of entities. Choose the visibilties to make only the given code valid, assume no other accesses of fields and methods. The three visibility specifiers in our toy language are defined as follows, ordered by increasing level of access.

- `private`: Members declared with this visibility can be accessed only by code in the current class, but not by code outside of the class. This is similar to Java and C++.
- `protected`: Members declared with this visibility can be accessed only by code in the current class or in derived classes, but not by superclasses or unrelated classes. This is similar to C++. [2]
- `public`: Members declared with this visibility can be accessed by any code. This is similar to Java and C++.

Assume that only `public` or `protected` methods can be overridden by subclasses (i.e., a method with the same name as a private method in a superclass is not allowed)[3] and that the visibility may not change when a method is overridden. Also assume all classes are publically visible.

To submit your answer, please fill the CSV file *exercise6/task2.csv*. Submit exactly one visibility specifier (as the value `private`, `protected`, or `public`) for each blank. Note that there is no default visibility. If you do not fill a blank, it is invalid and you will not get points for that member.

The class definitions (with blank visibility specifiers) and the code using those classes follows below. For simplicity, assume (implicit) default constructors, similar to Java. Also assume that the code after line 55 is not part of the previously defined classes.

```
1  class Animal {
2      __blank1__ String name = "unknown";
3      __blank2__ void setName(String name) { // Getters can be used to maintain
4          this.name = name.trim(); // invariants of your data, e.g., no spaces.
5      }
6      __blank3__ String getName() {
7          return this.name;
8      }
9      __blank4__ float weight = 1.0;
10     __blank5__ void setWeight(float weight) {
11         this.weight = weight;
12     }
13     __blank6__ void grow() {
14         this.weight *= 1.5;
15     }
16     __blank7__ void makeSound() {
17         print("nothing in particular");
18     }
19     __blank8__ void introduction() {
20         print("Hi, I am " + this.getName() + "and I make");
21         this.makeSound();
22     }
23     __blank9__ void bye() {
24         print("Bye bye");
25         this.makeSound();
26     }
```

[2] In Java, `protected` members can also be accessed by code in the same package. However, our toy language does not have packages.

[3] This is different from Java and C++, where one can have two private methods of the same name in an inheritance hierarchy. But there it results in *hiding*, not *overriding*.

```
}

class Lion extends Animal {
    __blank10__ boolean goodMood = true;
    __blank11__ boolean isNiceSmallCat() {
        return this.goodMood && this.weight < 100.0;
    }
    __blank12__ void makeSound() { // Overriding Animal.makeSound().
        if (this.isNiceSmallCat()) {
            print("meow");
        } else {
            print("raaawwrr!");
        }
    }
}

class Bee extends Animal {
    __blank13__ void sting(Lion lion) {
        lion.goodMood = false;
    }
    __blank14__ void grow() {      // Overriding Animal.grow().
        this.weight *= 1.05;       // Bees don't really grow that much.
    }
    __blank15__ void makeSound() { // Overriding Animal.makeSound().
        print("bzzzzz");
    }
}

// Somewhere else in the code, but outside of the previously defined classes:
Lion l = new Lion();
l.setName(" Leo  ");
l.setWeight(50);
l.introduction();
l.grow();
Bee b = new Bee();
b.setName("Berta");
b.introduction();
b.sting(l);
l.bye();
b.bye();
```

Evaluation Criteria: Your solution will be compared against the correct choice of minimal visibility specifiers.

3 Task III (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/task3/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/task3/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 in the following:

```
1  class Task3 {
2      static int fieldStatic = 0;
3
4      int fieldInstance = 0;
5
6      static void methodStatic() {
7          this.fieldStatic += 1;  // Validity 1
8          Task3.fieldStatic += 1; // Validity 2
9      }
10
11     void methodInstance() {
12         this.fieldInstance += 1;  // Validity 3
13         Task3.fieldInstance += 1; // Validity 4
14     }
15 }
16
17
18 // Somewhere else in the code, e.g., in a main method:
19 Task3 a = new Task3();
20 Task3 b = new Task3();
21
22 a.fieldStatic += 1;       // Validity 5
23 a.fieldInstance += 1;     // Validity 6
24 Task3.fieldStatic += 1;   // Validity 7
25 Task3.fieldInstance += 1; // Validity 8
26
27 value(a.fieldStatic);     // Behavior 1
28 value(a.fieldInstance);   // Behavior 2
29 value(Task3.fieldStatic); // Behavior 3
30 value(b.fieldInstance);   // Behavior 4
31
32 b.fieldStatic += 1;       // Validity 9
33 b.fieldInstance += 1;     // Validity 10
34
```

```
35 value(Task3.fieldStatic); // Behavior 5
36 value(a.fieldInstance);   // Behavior 6
37 value(b.fieldStatic);     // Behavior 7
38 value(b.fieldInstance);   // Behavior 8
39
40 a.methodStatic();         // Validity 11
41 a.methodInstance();       // Validity 12
42 Task3.methodStatic();     // Validity 13
43 Task3.methodInstance();   // Validity 14
44
45 value(Task3.fieldStatic); // Behavior 9
46 value(a.fieldInstance);   // Behavior 10
47 value(Task3.fieldStatic); // Behavior 11
48 value(b.fieldInstance);   // Behavior 12
```

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

4 Task IV (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.[4] 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. For multiple inheritance, assume that methods of the first base class (if any) come before methods of the second base class.

To submit your answers, please write the vtable entries in the plain text files *exercise6/task4/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

```
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.

```
class ClassA {
public:
    void methodA() { /* ... */ };
};

class ClassB {
public:
    virtual void methodB() { /* ... */ };
};

class ClassC: ClassA {
public:
    void methodC () { /* ... */ };
};

class ClassD: ClassB {
public:
    void methodB() override { /* ... */ };
    virtual void methodD() { /* ... */ };
};

class ClassE: ClassC, ClassD {
public:
    void methodD() override { /* ... */ };
    void methodE() { /* ... */ };
};
```

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

[4] 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.