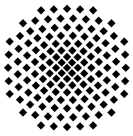

Universität Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart, Winter 2019/2020

Exercise 6: Data Abstraction and Object Orientation

(Deadline for uploading solutions: January 26, 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/
│   ├── semantics1.txt
│   ├── semantics2.txt
│   └── semantics3.txt
├── task2.csv
├── task3.zip
└── task4/
    ├── BankAccount.txt
    ├── Checking.txt
    ├── InterestChecking.txt
    └── Savings.txt
```

The submission must be compressed in a zip file (**not rar** or other formats) using the given directory structure. The names of directories and files must not be changed, otherwise the homework will not be evaluated. Late submissions will be not accepted.

There are four tasks, which contribute a specific percentage to the overall points for this exercise.

1 Toy Object-Oriented Language

In Task I and Task II, you are given several code snippets written in a simple toy language. The language has features similar to other object-oriented, class-based programming languages with pointers, such as C++. Instances of classes are implicitly initialized whenever a variable of a class type is declared.
However, the language does not support any *"visibilities"*. This results in all members defined in a class to be accessible outside the class as well. The language supports only public inheritance (i.e., all methods of the base class become methods of the derived class).

Furthermore, the language supports *"virtual methods"*. The fact whether a method in a superclass is declared as `virtual` determines whether this method can be overridden in a subclass, similar to the same keyword in C++.

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

This task is about **method binding**. You are provided with a parent class and multiple derived classes, all written in our toy language. The task is indicate which method is being called when **Task 1 Test Snippet** is executed under following semantics:

1. **Semantics 1**: The language always uses dynamic method binding.
2. **Semantics 2**: The language always uses static method binding.
3. **Semantics 3**: The language always uses static method binding, unless the method in the superclass is marked as `virtual` and the method in the subclass is marked with `override`.

To submit your answer, please fill the called methods into the files under *exercise6/task1/*. To specify a called method, use the following format "*Class Name::Method Name*". Each line in a file corresponds to one call. There must be one line per call in the given code snippet, using the order in which the calls appear in the code snippet. The following is a sample solution for a (hypothetical) code snippet that contains two method calls:

```
sampleBase::method1()
sampleDerived::method2()
```

Class Structure

```
class employee{
  void details(){ print("employee-details");}
  virtual void duty(){print("some-duties")};
}

class intern: employee{
  override void duty() {print("fetch-coffee")}
  override void details() { print("intern-details"); }
}

class permanent: employee{
  override void details() {print("permanent-details");}
}

class manager: permanent{
  override void duty() {print("managerial-duties");}
}
```

Task 1 Test Snippet

```
employee e;
intern i;
permanent p;
manager m;

employee *e1 = &p;
employee *e2 = &m;
employee *e3 = &i;

//method calls:
m.details();
p.duty();
e.details();
e1->details();
e2->details();
e3->details();
e.duty();
e2->duty();
e3->duty();
```

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

3 Extending the Toy Language

We now extend our toy language with **visibilities**, which specify which classes or functions can access members of a class. The extension is motivated by the "*principle of least privilege*", i.e., the practice of limiting access rights for an entity to the minimum permissions that entity needs to perform its work. The following visibility modifiers may be assigned to members of a class:

- **public:** All class members declared as public will be accessible from anywhere outside the class.
- **protected:** Members declared as protected may be accessed by members of either the same class or any subclass derived from the class.
- **private:** Members declared as private cannot be accessed, or even viewed from outside the class.
- **friend F:** Friend is a special visibility modifier that overrides any protected and private modifiers that may be in place. If member of class C has visibility `friend F`, where F is another class, then any method in F may access the member of C as if that member was declared within F.

  Limitation: Friendship is neither *transitive* nor *inherited*.

The syntax for assigning one or more visibilities to a member is to prepend the space-separated visibilities just before the member declaration. For example, a member declaration `private friend F int f;` declares a private integer field named `f` that may be accessed by its friend class `F`. If multiple visibilities are given for a single member, then their order is irrelevant.

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

In this task, you have to **assign visibilities** to code written in our extended toy language. In the provided skeleton file *exercise6/task2.csv*, please add the **least necessary visibility level** for each class members that allows other classes and functions to operate while ensuring that no class or function can access more members than necessary. If you do not want to add any visibility into a blank, just leave the entry empty.
Note: For the sake of simplicity assume all properties are initialized in the constructor and all relevant super class constructors have also been called.

Task II Class Structure

```
class employee{
  __blank1__ int employeeID;
  __blank2__ void details() { print("employee-details"); }
  __blank3__ virtual void duty() { print("some-duties"); }
}

class intern: employee {
  __blank4__ int internshipDuration;
  __blank5__ string educationDetails;
  __blank6__ manager m;

  __blank7__ override void duty() { print("fetch-coffee"); }
  __blank8__ override void details() { print(employeeID+"-intern"); }
  __blank9__ int getInternshipDuration() { return internshipDuration; }
}

class permanent: employee {
  __blank10__ int level;
  __blank11__ int assignedTeam;

  __blank12__ override void details() { print(employeeID+"-permanent"); }
  __blank13__ void getAssignedTeam() { print("Team-"+assignedTeam); }
}

class manager: permanent {
  __blank15__ override void duty() { print(employeeID+"managerial-duties"); }
  __blank16__ void assignTeam(permanent p, int team) { p.assignedTeam = team; }
  __blank17__ void viewEducationDetails(intern i)
```

```
29          { print(i.educationDetails); }
30    __blank18__ void setInternshipDuration(intern i, int duration)
31          { i.internshipDuration = duration;}
32  }
33
34  class helper {
35    __blank19__ void showLevel(permanent p) { print(p.employeeID+":"+level); }
36  }
```

Task II Test Snippet

```
1   permanent p1;
2   manager m;
3
4   m.assignTeam(p1, 1);
5   m.details();
6
7   intern i;
8   m.setInternshipDuration(i, 6);
9   print(i.details()+'-'+i.getInternshipDuration());
10
11  permanent p2;
12  m.assignTeam(p2, 1);
13  p2.duty();
14  p2.getAssignedTeam();
15
16  helper h;
17  h.showLevel(m);
18  h.showLevel(p2);
```

Evaluation Criteria: Your solution will be compared against a correct assignment of blanks to visibilities.

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

This task is about **overriding and overloading** in Java.

- **Overriding** is a feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its super classes.
- **Method overloading** is a feature that allows a class to have more than one method with the same name, if their argument types differ.

You are provided with a **base class** Figure and two **derived classes** Circle and Rectangle. The base class has defined six abstract methods (listed below) that you are required to provide custom overridden and overloaded implementations for in the derived classes.

Abstract Methods defined in Figure class

```
1   public abstract double getDistance(Figure figure);
2   public abstract double getDistance(int x, int y);
3   public abstract double getDistance(Point toCompare);
4   public abstract double getArea();
5   public abstract double getPerimeter();
6   public abstract String toString();
```

You are required to implement each method according to the description provided below.

Method Description for Circle class

```
//returns the distance of the center of this circle and the centre
// of a given circle
public abstract double getDistance(Figure figure);
//returns the distance of the center of this circle and x,y coordinates
public abstract double getDistance(int x, int y);
//returns the distance of the center of this circle and a Point
public abstract double getDistance(Point toCompare);
//methods that return the area of the circle
public abstract double getArea();
//methods that return the perimeter of the circle
public abstract double getPerimeter();

//returns a string description of this instance in the
//format: radius=r,center=(x,y)
public abstract String toString();
```

Method Description for Rectangle class

```
//returns the distance of the starting point of this rectangle
//and starting point of other rectangle.
public abstract double getDistance(Figure figure);
//returns the distance of the starting point of this rectangle
//and x,y coordinates
public abstract double getDistance(int x, int y);
//returns the distance of the starting point of this rectangle
// and a Point
public abstract double getDistance(Point toCompare);
//methods that return the area of the rectangle
public abstract double getArea();
//methods that return the perimeter of the rectangle
public abstract double getPerimeter();
//returns a string description of this instance in the
// format: starting=(x,y),length=l,width=w
public abstract String toString();
```

An Eclipse project for the overriding implementation is provided: *exercise6/task3*. You can import the project template (.zip) into Eclipse as follows: *File-Import-General-Existing project into Workspace-Select archive file-Finish*.

You find a JUnit test in folder *exercise6/task3/src/*. Use the test to check whether your code works. We will use additional tests to evaluate your solution, and you are advised to also add additional tests for your own testing.

For the submission, your project must be exported into a .zip archive using Eclipse: *File-Export-General-Archive File*, and then added to the tree structure specified above.

Evaluation Criteria: Your code will be evaluated against a set of test cases that exercise the code and check each individual subclass. Some example tests are provided along with this exercise. We will test your code with additional tests. During the evaluation, we will use Java 11.

6 Task IV (25% of total points of the exercise)

This task is about **vtables**. You are provided with a **base class** BankAccount and three **subclasses** Checking, InterestChecking, and Savings. You can find the Banking system implementation under *exercise6/task4/Banking.cpp*. Your task is to give a vtable for each of the classes. For constructing the vtables, assume that the compiler works as described in the lecture. In the absence of other constraints for the order of vtable entries, assume that the compiler will order methods alphabetically.
For each vtable, submit a separate text file, as provided in template. Each vtable entry must be given in one line, and each line must start with **pointer-to**, followed by a space and the method it points to. To indicate a method, please use **ClassName::methodName**. For example, if the n-th line in a file is **pointer-to A::m**, then this means that the n-th vtable entry points to the code of method `m` of class `A`. The entries should follow the

order assumed by the compiler.

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