

Concepts of Programming Languages (INFOTECH)

Summer Semester 2006
Steffen Keul, Stefan Staiger

Solutions to Assignment #3

1 Name Binding and Method Calls

1.1 Dynamic name binding

- Call (1) is legal: The name "M" is bound to method M of class NT, since this is the dynamic type of OT. Passing OT as parameter is okay since every object of class NT can be seen as object of class T, too. So call (1) produces the output "NT".
- Call (2) is also legal: The name "M2" is bound to method M2 of class NT (the dynamic type of OT).
- Call (3) is legal: It calls method M of class NT, this time with controlling argument ONT. As in call (1), OT can be seen as object of class T and can thus be passed as argument. The output is "NT".
- Call (4) is legal.

Note that if some call were illegal, this would result in a runtime error.

1.2 Static name binding

- Call (1) is legal: static name binding binds the name to the method signature found in class T, and the call matches this signature. But it depends on some factors which implementation of M actually will be called. In C++, if M were declared "virtual", and in Java, there would be a dispatching according to the dynamic type of OT, so again method M of class NT would be called and the output would be "NT". But if the method M of class T were not declared virtual in C++, the call would be bound statically to the implementation in class T and thus the output would be "T".
- Call (2) is *illegal*: Static name binding tries to bind the name "M2" to some method in class T (which is the static type of OT), but there is none. So here the compiler would not compile your program and give you an error message about the problem.
- Calls (3) and (4) are legal and call the methods of class NT, since this is the static type of ONT.

1.3 Advantages of static name binding

Static name binding detects errors early (when compiling, not at runtime) and produces faster (and sometimes also smaller) code, since many calls can be directly bound to some implementation.

1.4 Overloading and Redefinition

Overloading means that we have more than one function with the same name, but with a different signature and probably different implementations. For example, in C++ we can have two functions

```
int add (int x, int y) {...}
complex add (complex x, complex y) {...}
```

Although these functions both have the name "add", their signature differs. As you can see, overloading is independent from object-oriented concepts. On the other hand, *redefinition* is mainly a feature from object-oriented programming. In the example program on the exercise sheet, class T introduces a method M. Now all classes that inherit from T also inherit this method, but they can provide their own implementation. This happened in class NT: The method M is redefined there, that is, we have a method M with the same signature as in class T but with a new method body (a new definition, thus redefinition). Note that the controlling object ("this" in C++) does not count for the same signature. So in short:

- Overloading: same name, different signature, different implementation, independent functions
- Redefinition: same name, same signature (apart from "this"), different implementation, replacing the old one

That is, a redefined method has the same interface to the outer world as the redefined method, and it replaces the old method, but overloaded methods only more or less happen to have the same name, they are otherwise independent.

1.5 Signature inheritance in C++

In C++, we have

- Covariant inheritance for the controlling object ("this")
- Invariant inheritance for other parameters
- Normally invariant inheritance for the return type, with a small exception: If the return type is a pointer or reference to a class, you can redefine the method with a covariant return type. In the example, if `return_type` were `C*` (and the method were `virtual`) then we could have

```
class D : public C {
    D* methodname (param_type1 param1, param_type 2 param2);
}
```

and this would be a redefinition.

2 Exceptions

Let's see what happens at runtime step-by-step:

- The program starts with Main, which calls *P* (line 29)

- *P* calls *Q* (line 14)
- *Q* raises *E2*, which is caught by the handler `when others` in *Q* itself
- So the first output is `Q: other exception` and then *E1* is raised
- That exception is caught by the handler in *P* (line 16), so the next output is `P/1: E1` and *Q* is called again.
- As before, this results in raising *E2* which is caught in line 9 and outputs `Q: other exception`
- Next, *E1* is raised, but this time, the handler in line 24 will catch it. The next output is thus `P/3: E1` and *E1* is re-raised.
- This activates the exception handler in line 31, which prints `Main: E1`. Then, the program finishes.

So the order of execution is this (line numbers): 29 - 14 - 06 - 09 - 16 - 06 - 09 - 24 - 31

The output of the program is

1. `Q: other exception`
2. `P/1: E1`
3. `Q: other exception`
4. `P/3: E1`
5. `Main: E1`

As you can see, exceptions heavily influence the flow of control.