Visibility

Visibility and Security

visibility on class level (Java, C#, C++, ...)

→ focus on **responsibilities** of class

visibility on object level (variables in Eiffel)

→ focus on **substitutability** and assertions

visibility as a **security concept** (not in mainstream languages)

→ constraints on references to objects:

uniqueness: only one reference to object can exist
ownership: only referred to by one object
confinement: only referred to within a specific region

Escape Analysis

compiler (including JIT) analyses object use

if object stays in local scope (non-escape), then

- confined to scope
- object variables on stack or in other object
- cheap allocation, further optimizations possible

else if object stays within thread, then

- confined to thread
- no synchronization necessary

used e.g. in Java HotSpot VM for optimization
and in language extensions / tools to give security guarantees

Value Types (Compared to Reference Types)

struct in C# (and in C++ when not used as reference) has value semantics, other non-elemenatry objects (usually) have reference semantics

value copied on parameter passing (not just reference to value)

variables allocated on stack or as part of other objects

similar to non-escape objects

advantageous if small or used only locally

Templates in C++

Some Properties

constants usable as generic parameters:

```
template<typename T, int n> class buffer { ... };
```

(partial) specialization with “pattern matching”:

```
template<int n> class buffer<bool, n> { ... };
template<> class buffer<bool, 0> { ... };
```

implicit instantiation if no appropriate specialization available

code for methods produced only if used

Expressiveness

can be used to evaluate expressions statically (Turing-complete):

```
template<int x, int n> struct power {
    static const int r = x * power<x, n-1>::r;
};
template<int x> struct power<x, 0> {
    static const int r = 1;
};
```

algebraic data structures expressible:

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons<int, Cons<float, Nil> > list_of_types;
```

Policy-Based Programming

```
#include <iostream>
template<typename lang> class HelloWorld : public lang {
   public: void Run() { cout << Message() << endl; }
};          // Message inherited from generic parameter

#include <string>
class HelloWorld_Msg_German {
   protected: string Message() { return "Hallo Welt!"; }
};

typedef HelloWorld<HelloWorld_Msg_German> MyHelloWorld;
MyHelloWorld hello;
hello.Run();
```

Alternative: Strategy Design Pattern (Java)

```
interface IMessage { String message(); }
class DMessage implements IMessage {
    public String message() {return "Hallo Welt!";}
}

class HelloWorld {
    private IMessage msg;
    public HelloWorld(IMessage msg) {this.msg = msg;}
    public void run() {System.out.println(msg.message());}
}

class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld(new DMessage());
        hello.run();
    }
}
```

Functional versus Object-oriented Programming

Functional Object-oriented Programming

possible point of view: object-orientation = organisational paradigm,
model of computation orthogonal (imperative, functional, etc.)

contradiction: object has state, behaviour, identity;
referential transparency = no state, no identity

in practice: no referential transparency for input/output

→ state and identity also in functional programming,
but most inner program parts are referentially transparent

state of the art: functional part separated from the object-oriented part

pattern matching also usable in object-oriented programming,
genericity related to referential transparency

Delegates in C#

```
using System;
delegate void Sample(string message);
class MainClass {
    static void Method(string message)
        { Console.WriteLine(message); }
    static void Main() {
        // Instantiate delegate with named method:
        Sample d1 = Method;
        // Instantiate delegate with anonymous method:
        Sample d2 = delegate(string message)
                    { Console.WriteLine(message); };
        d1("Hello");
        d2(" World");
    }
}
```

Lambda Expressions in Java

```
interface X { void anyname(String s); }
interface Y { int op(int a, int b); }

X x1 = (String p) -> System.out.println(p);
X x2 = p -> { System.out.println(p); System.out.println(p); };
Y y1 = (l, r) -> l + r;
Y y2 = (l, r) -> { return l * r; };

x1.anyname("Hello world!");
x2.anyname(y1.op(1,2) + " " + y2.op(3,4));
```

Lambda Expression

is a **closure** encapsulating a method in an object;
local context is kept (in contrast to function pointer);
anonymous, only accessible through object

simple syntax = syntactic suggar + intelligence

environment implicitly created (as inner class)

type inference for parameters (because of no subtyping)

important in applicative programming styles