Mechanisms for Direct Code Reuse

Inheritance versus Delegation

```
class A {
    public void x() { z(); }
    protected void z() { /* A-Code */ ... }
}
class B extends A {
    protected void z() { /* B-Code */ ... }
    public void y() { delegate.x(); }
    private A delegate = new A();
}
```

inheritance: `new B().x()` → B-Code

delegation: `new B().y()` → A-Code

if no method overridden as done for z, there is no difference

Trait

a trait provides implemented methods to be reused

a trait requires methods (provided elsewhere)

a trait neither has nor uses object variables

ordering does not matter when composing software from traits

use of traits as if methods were defined locally

example: default implementation in Java interface

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

Final Remarks

History of Object-oriented Programming

Languages: Simula, Smalltalk, Objective-C, C++, Eiffel, Self, CLOS, Oberon, Java, C#, Python, Ruby, ...

Concepts: structured programming, abstraction, inheritance, substitutability, interface specifications, parametrisation (genericity, annotations, aspects, ...)

Methods: factorization, use cases, graphical representation (UML), design patterns, pair programming, ...

Conflicts: functional programming, relational databases, collections and covariant problems, formal complexity, concurrency

Trends: object-based, object-oriented, (partially automated), typed, team+architecture-integrated, layers and frameworks, back to the roots

Future of Object-oriented Programming

OOP omnipresent → no longer innovativ

splitting into many details and side issues

topics of the near future: concurrency, distributed programming, data integration and big data, cloud computing, complex behavioural interfaces, deeply layered architectures, security, . . .

currently more open questions than answers

language support expected when most important questions answered
→ language support mainly for topics that are no longer up-to-date?