COOP Summary

Ulla Aeschbacher

16.12.18

Contents

1 Introduction

1.1 Requirements

1.1.1 New Requirements in SW-Technology: These requirements are not satisfied in imperative languages like C, Pascal, Fortran, ...

- Reuse: Quality, Documented Interfaces, Extendibility and Adaptability
- Computation as Simulation: Modeling Entities of the Real World, Describing Dynamic System Behaviour, Running Simulations
- GUIs: Adaptable Standard Functionality, Concurrency
- Distributed Programming: Concurrency, Communication, Distribution of Data and Code

1.1.2 Core Requirements:

- Cooperating Program Parts with Well-Defined Interfaces: Objects (data and code), Interfaces, Encapsulation
- Classification and Specialization: Classification, Subtyping, Polymorphism, Substitution principle
- Highly Dynamic Execution Model: Active objects, Message passing
- Correctness: Interfaces, Encapsulation, Simple powerful concepts

1.2 Core Concepts

1.2.1 The Object Model: A software system is a set of cooperating objects. Objects have state and processing ability. Objects exchange messages.

Objects	Values
State	Can't change state
Identity	Don't have identity
Lifecycle	Don't need to initialize
Location	Don't have a location

1.2.2 Interfaces and Encapsulation: Objects have well-defined interfaces (publicly accessible fields and methods). Implementation is hidden behind the interface. The interfaces are the basis for describing behaviour.

1.2.3 Classification and Polymorphism:

- Classification: Hierarchical structuring of objects. Objects belong to different classes simultaneously
- Substitution principle: Subtype objects can be used wherever supertype objects are expected.
- Polymorphism: A program part is polymorphic if it can be used for objects of several classes.

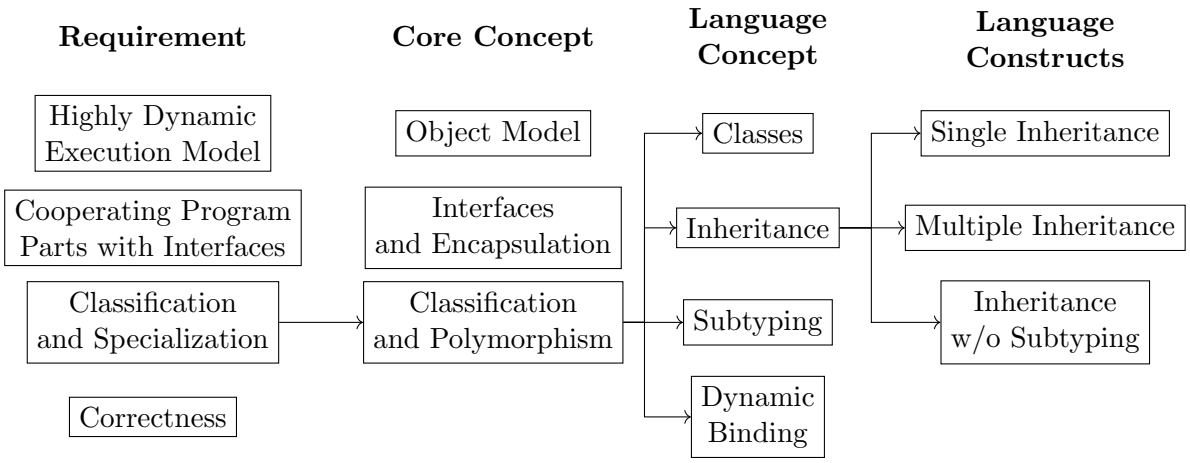
- Subtype polymorphism: A direct consequence of the substitution principle. Program parts working with supertype objects work as well with subtype objects.
- Other forms of polymorphism (These are not core concepts): Parametric polymorphism (generic types), Ad-hoc polymorphism (method overloading)

1.2.4 Specialization: Adding specific properties to an object or refining a concept by adding further characteristics. Requires the behaviour of the specialized object to be compliant to the behaviour of more general objects. Program parts that work for the more general objects work as well for specialized objects.

1.2.5 Summary: Core concepts are abstract concepts to meet the new requirements. To apply the core concepts we need ways to express them in programs. Language concepts enable and facilitate the application of the core concepts.

1.3 Language Concepts

Appropriate language support is needed to apply object-oriented concepts.



1.4 Language Design Goals

1.4.1 Simplicity: Simplicity: Syntax and semantics can easily be understood by users and implementers of the language. Examples: Basic, Pascal, C

1.4.2 Expressiveness: Language can easily express complex processes and structures. Examples: C#, Scala, Python

- Expressiveness vs. Simplicity: e.g C++ Inheritance, Reflections, yield in Python

1.4.3 Static Safety: Language discourages errors and allows errors to be discovered and reported, ideally at compile time. Examples: Java, C#, Scala.

- Safety vs. Expressiveness: e.g. static safety says it's wrong if it isn't sure

1.4.4 Modularity: Language allows modules to be compiled separately. Examples: Java, C#, Scala

- Modularity vs. Expressiveness: e.g. automatic types would be great, but one would need to know the whole program

1.4.5 Performance: Programs written in the language can be executed efficiently. Examples: C, C++, Fortran

- Performance vs. Simplicity: e.g. memory models give compiler more optimization possibilities, but they are hugely complicated.
- Performance vs. Safety: e.g. array bound checks in Java, null pointer checks, garbage collectors
- Performance vs. Modularity: e.g. if one knows the whole program, one can optimize lots

1.4.6 Productivity: Language leads to low costs of writing programs. Examples: Visual Basic, Python

- Productivity vs. Static Safety: e.g. types give safety but slow down code writing
- Productivity vs. Performance: e.g. static types give more performance but less productivity, more performance when going nearer to the hardware but Assembly is just hard to write

1.4.7 Backwards Compatibility: Newer language versions work and interface with programs in older versions. Examples: Java, C (not Python, Scala)

- Backwards Compatibility vs. Simplicity: e.g. keep features not used anymore
- Backwards Compatibility vs. Expressiveness: biggest problem, cannot introduce features that would break backwards compatibility
- Backwards Compatibility vs. Performance: e.g. have to emulate old behaviour

2 Types and Subtyping

2.1 Types

2.1.1 Type: A type is a set of values sharing some properties. A value v has type T if v is an element of T.

2.1.2 Type Systems: A type system is a tractable syntactic method for proving absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

2.1.3 First Dimension: Weak and Strong Type Systems:

- Untyped languages: Do not classify values into types. Examples: assembly languages
- Weakly-typed languages: Classify values into types, but do not strictly enforce additional restrictions. Can cast anything to anything. Examples: C, C++
- Strongly-typed languages: Enforce that all operations are applied to arguments of the appropriate types. Examples: C#, Eiffel, Java, Python, Scala, Smalltalk

2.1.4 Second Dimension: Nominal and Structural Types:

- Nominal types: based on type names. Examples: C++, Eiffel, Java, Scala
- Structural types: based on availability of methods and fields. Here the names of the types do not matter, so we omit them. Also the behaviour and return type of the methods does not matter, two methods are the same if they have the same name and same argument types. Examples: Python, Ruby, Smalltalk, Go, O'Caml

2.1.5 Third Dimension: Static and Dynamic Type Checking:

- Static Type Checking: each expression of a program has a type. Types of variables and methods are declared explicitly or inferred. Types of expressions can be derived from the types of their constituents. Type rules are used at compile time to check whether a program is correctly typed. Statically type-safe OO-languages guarantee the following type invariant: In every execution state, the type of the value held by a variable v is a subtype of the declared type of v. Most static type systems (except C) rely on dynamic checks for certain operations. Common example: type conversions by casts. Some static type systems provide ways to bypass static checks (e.g. dynamic in C#). Here type safety is preserved via run-time checks.
- Dynamic Type Checking: Variables, methods and expressions of a program are typically not typed. Every object and value has a type. Run-time system checks that operations are applied to expected arguments. Dynamic checkers support on-the-fly code generation and dynamic class loading. Example: JavaScript eval takes a string, interprets it as code and runs it. This can only be done if runtime checks are present, a static type language has no way of dealing with this.

Static checking	Dynamic checking
Static safety: More errors found at compile time	Expressiveness: No correct program is rejected by the type checker
Readability: Types are excellent documentation	Low overhead: No need to write type annotations
Efficiency: Type information allows optimizations	Simplicity: Static type systems are often complicated
Tool support: Types enable auto-completion, support for refactoring, etc.	

2.1.6 Overview of Type Systems in OO-languages:

	Static	Dynamic
Nominal	Sweet spot: Maximum static safety. Examples: C++, C#, Eiffel, Java, Scala	Why should one declare all the type information but then not check it statically? Still used for certain features of statically-typed languages like casts in Java.
Structural	Overhead of declaring many types is inconvenient and problems with semantics of subtypes. Examples: Go, O'Caml	Sweet spot: Maximum flexibility. Examples: JavaScript, Python, Ruby, Smalltalk

2.2 Subtyping

2.2.1 Classification in Software Technology:

- Syntactic classification: Subtype objects can understand at least the messages that supertype objects can understand. Languages are like this.
- Semantic classification: Subtype objects provide at least the behaviour of supertype objects. Used in special cases.

2.2.2 Nominal and Structural Subtyping:

- Nominal type systems: Determine type membership based on type names. Determine subtype relations based on explicit declarations.
- Structural type systems: Determine type membership and subtype relations based on availability of methods and fields.

2.2.3 Nominal Subtyping and Substitution: Subtype objects can understand at least the messages that supertype objects can understand. Subtype objects have wider interfaces than supertype objects.

- Existence: Subtypes may add, but not remove methods and fields.
- Accessibility: An overriding method must not be less accessible than the method it overrides.

- Parameter Types: An overriding method must not require more specific parameter types than the method it overrides (contravariant). Note: Java does not allow contravariant parameters because of overloading.
- Result Types: An overriding method must not have a more general result type than the method it overrides (covariant). Out-parameters and exceptions are results.
- Fields: Subtypes must not change the types of fields.
- Immutable Fields: Immutable fields can be specialized in subclasses. Not permitted in most languages (Scala has it), because this doesn't work if the supertype constructor initializes the field with a supertype value, which is not allowed. But this is a very common place to assign immutable fields.
- Arrays: In Java and C#, arrays are covariant. Then each array update requires a run-time type check, but covariant arrays allow one to write methods that work for all arrays. Generics allow a solution that is expressive and statically safe, but backwards compatibility forces them to keep it.

2.2.4 Shortcomings of Nominal Subtyping:

- Nominal subtyping can impede reuse: Cannot easily add a superclass of several classes. One way to solve this is by using an adapter pattern, but this requires boilerplate code and causes memory and run-time overhead. Another way is generalization, but it does not match well with inheritance.
- Nominal subtyping can limit generality: Many method signatures are overly restrictive. A method may use only some methods of a class, but requires a type with lots of methods. One way to solve this is to use additional supertypes, but there is an overhead for declaring supertypes and subtyping. Another way is to use optional methods, but with that, static safety is lost.

2.2.5 Structural Subtyping and Substitution: Subtype objects can understand at least the messages that supertype objects can understand. Structural subtypes have by definition wider interfaces than their supertype. There is no need for a compiler to check subtyping relations, because he is the one who defines the subtype relations. This solves both the reuse and the generality problem of nominal subtyping.

2.3 Behavioral Subtyping

2.3.1 Contracts:

- Preconditions: have to hold in the state before the method is executed.
- Postconditions: have to hold in the state after the method body has terminated.
- Old-expressions: can be used to refer to prestate values from the postcondition. For parameters, old and new is the same, because if the callee changes that parameter, then the caller is never informed about that.
- Invariants: describe consistency criteria for objects. They have to hold in all states in which an object can be accessed by other objects.
- History constraints: describe how objects evolve over time. They always relate visible states and have to be reflexive and transitive.

2.3.2 Static and Dynamic Contract Checking:

Static checking Program verification	**Dynamic checking** Run-time assertion checking
Static safety: More errors are found at compile time	Incompleteness: Not all properties can be checked efficiently at run-time.
Complexity: Static contract checking is difficult and not yet mainstream	Efficient bug-finding: Complements testing
Large overhead: static contract checking requires extensive contracts	Low overhead: Partial contracts are useful
Examples: Spec#, .NET	Examples: Eiffel, .NET

2.3.3 Rules for Subtyping:

- Preconditions: Overriding methods of subtypes may have weaker preconditions than the corresponding supertype methods (contravariance).
- Postconditions: Overriding methods of subtypes may have stronger postconditions than the corresponding supertype methods (covariance).
- Invariants: Subtypes may have stronger invariants (covariance).
- History Constraints: Subtypes may have stronger history constraints (covariance).

2.3.4 Static Checking of Behavioral Subtyping: Have to check these for all paramters, heaps and results. Entailment is normally undecidable.

- Preconditions: $Pre_{Super.m} \Rightarrow Pre_{Sub.m}$
- Postconditions: $old(Pre_{Super.m}) \Rightarrow (Post_{Sub.m} \Rightarrow Post_{Super.m})$
- Invariants: $Inv_{Sub} \Rightarrow Inv_{Super}$
- History Constraints: $Cons_{Sub} \Rightarrow Cons_{Super}$

2.3.5 Specification Inheritance: Behavioral subtyping can be enforced by inheriting specifications from supertypes. The run-time checker can check the effective contracts.

- Effective Preconditions: The effective precondition $PreEff_{Sub.m}$ of a method m in class Sub is the disjunction of the precondition $Pre_{S.m}$ declared for the method and the preconditions $Pre_{Super.m}$ declared for the methods it overrides.

$$\begin{aligned} PreEff_{Sub.m} = & Pre_{Sub.m} || \\ & Pre_{Super.m} || \\ & Pre_{Supersuper.m} || \\ & \ldots \end{aligned}$$

  With this, overriding methods have weaker effective preconditions.

- Effective Postconditions: The effective postcondition $PostEff_{Sub.m}$ of a method m in class Sub is the conjunction of implications $(old(Pre_{Super.m}) \Rightarrow Post_{Super.m})$ for all

types Super such that Super declares Sub.m or Sub.m overrides Super.m.

$$\begin{aligned} PostEff_{Sub.m} =& (old(Pre_{Sub.m}) \Rightarrow Post_{Sub.m})\&\& \\ & (old(Pre_{Super.m}) \Rightarrow Post_{Super.m})\&\& \\ & (old(Pre_{Supersuper.m}) \Rightarrow Post_{Supersuper.m})\&\& \\ & \ldots \end{aligned}$$

With this, overriding methods have stronger effective postconditions.

- Invariants: The invariant of a type S is the conjunction of the invariant declared in S and the invariants declared in the supertypes of S. So by definition, subtypes have stronger invariants.
- History Constraints: Analogous to invariants.

2.3.6 Behavioral Structural Subtyping: Until now, everything was for nominal subtyping. With dynamic type checking, callers have no static knowledge of contracts. So pre- and postconditions are not better than an assert, because the caller can never see them. With static structural type checking, callers could state which signature and behaviour they require. Can check contract statically or dynamically. Behavioral subtyping needs to be checked when the type systems determines a subtype relation. Static checking is possible, but in general not automatic. Dynamic checking is not possible.

2.3.7 Types as Contracts: Types can be seen as a special form of contract, where static checking is decidable. Example:

```
class Types {
    Person p;

    String foo(Person q){...}
}
```

```
class Types {
    //invariant type(p) <: Person
    p;

    //requires type(q) <: Person
    //ensures type(result) <: String
    foo(q){...}
}
```

This is not exactly the same. The invariant only holds in visible states, while the type would have to hold in all states. Also with these rules, fields would be covariant instead of invariant.

2.3.8 Invariants over Inherited Fields: Invariants over inherited field f can be violated by all methods that have access to f. Static checking of such invariants is not modular. One would need to check the whole package (if it is protected) or all the code in the world (if it is public). Even without qualified field accesses (x.f = e vs. this.f = e) one needs to re-check all inherited methods. This is an unsolved problem.

2.3.9 Immutable Types: Objects of immutable types do not change their state after construction. Advantages are that there are no unexpected modifications of shared objects, no inconsistent states and most importantly no thread synchronization is necessary.

2.3.10 Subtype Relation of Immutable and Mutable Types:

- Immutable types should be the subtype: Not possible because the mutable type has a wider interface.
- Mutable types should be the subtype: The mutable type does not specialize the behaviour. We also can't guarantee the immutability of the immutable type.

- No subtype relation between mutable and immutable types. The exception is Object, which works because it has no fields.

3 Inheritance

3.1 Inheritance and Subtyping

3.1.1 Inheritance vs. Subtyping: Subtyping expresses classification, Inheritance is a means of code reuse. Inheritance is usually coupled with subtyping. Example for subtyping without inheritance: Java Interfaces. Example for inheritance without subtyping: C++ private inheritance. Subclassing = Subtyping + Inheritance.

3.1.2 Sets and Bounded Sets:

```
class Set{
    int size;

    void add(Object o){
        //add o
    }

    boolean contains(Object o){
        ...
    }
}
```

```
class BoundedSet{
    int size;
    int capacity;

    void add(Object o){
        //add o if there is space
    }

    boolean contains(Object o){
        ...
    }
}
```

BoundedSet is not a behavioral subtype of Set, but Set is also not a behavioral subtype of BoundedSet. We would still like to reuse the code. General answer: Subtyping dominates inheritance!

- Aggregation: BoundedSet uses Set, method calls are delegated to Set. Here it works, but there are other examples, where we need subtyping.
- Creating New Objects: Let BoundedSet be a subclass of Set. In add, just return a new larger BoundedSet with all the elements, if it wouldn't fit in the old one. This makes the set not really bounded anymore, so this is not what users of BoundedSet want.
- Weak Superclass Contracts: Just have the weakest contracts in a new superclass, and make the two classes behavioral subtypes. Could even introduce static contracts, which specify a given method implementation. Dynamically-bound calls use the standard contract, while statically-bound calls use a combination of static and standard contracts. With those, this is good design.
- Inheritance w/o Subtyping: Like private/protected inheritance in C++ or non-conforming inheritance in Eiffel.

3.2 Dynamic Method Binding

3.2.1 Method Binding:

- Static binding: At compile time, a method declaration is selected for each call based on the static type of the receiver expression. Examples: C++ (because of performance), C#(because of versioning). This is one of the most fundamental differences between C# and Java.

- Dynamic binding: At run time, a method declaration is selected for each call based on the dynamic type of the receiver object. This enables specialization and subtype polymorphism. However we have an performance overhead of method look-up at run-time and it makes it harder to evolve code without breaking subclasses. Examples: Eiffel, Java, Scala, all dynamically-typed languages.

3.2.2 Fragile Baseclass Scenario: Subclasses can be affected by changes to superclasses. How should we apply inheritance to make our code robust against revisions of superclasses?

- Subclass: Using inheritance, rely on interface documentation not implementation. Override all methods that could break invariants. Avoid specializing classes that are expected to be changed often.
- Superclass: Do not change calls to dynamically-bound methods.
- In C++ methods are bound statically by default. Potential overrides must be declared as either override or new (default). This prevents accidental overriding.

3.2.3 Overloading:

- In Java, overloading resolution chooses the most specific method declaration from all methods that are available. So adding methods to a superclass may affect clients of the subclass.
- In C++, overloading resolution chooses the most specific method declaration in the class of the receiver, then in the superclass etc. Adding methods to a superclass does not affect overloading resolution.

3.2.4 Binary Methods: Binary methods take a receiver and one explicit argument. Often behaviour should be specialized depending on the dynamic types of both arguments. But covariant parameter types are not statically type-safe.

- Explicit Type Tests: Type test and conditional for specialization based on dynamic type of explicit type argument. Problems: tedious to write, code is not extensible, requires type cast.
- Double Invocation (Visitor Pattern): Additional dynamically-bound call for specialization based on dynamic type of explicit argument. This does a flip of the arguments: First we have t1.calls(t2) and then t2.calls(t1), so for each of the parameters of the initial binary method, there is a dynamically bound call where the parameter is the receiver. Problems: even more tedious to write, requires modification of superclass. So this is only applicable if one has the whole subclass hierarchy under control.
- Overloading plus Dynamic: Cast parameters to dynamic. Dynamic resolution depends on dynamic types of both arguments. Problems: not entirely type-safe, overhead for run-time checks. Also we do not know what it returns, so we might run into a run-time error.
- Multiple Dispatch: Some research languages allow method calls to be bound based on the dynamic type of several arguments. Problems: Performance overhead of method look-up at run-time, extra requirements are needed to ensure there is a unique best method for every call.

3.3 Multiple Inheritance

3.3.1 Simulating Multiple Inheritance: In Java and C#, we can simulate multiple inheritance via aggregation and delegation.

3.3.2 Problems of Multiple Inheritance:

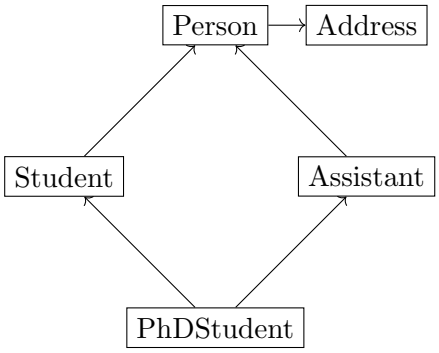
- Ambiguities: Superclasses may contain fields and methods with identical names and signatures. Which version should be available in the subclass?
- Repeated inheritance (diamond of death): A class may inherit from a superclass more than once. How many copies of the superclass members are there? How are the superclass fields initialized?

3.3.3 Ambiguity:

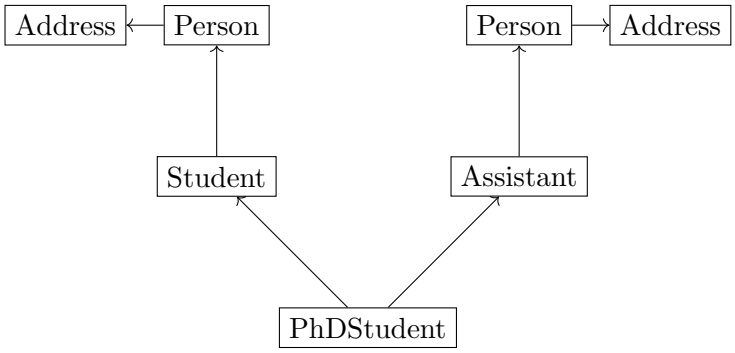
- Explicit Selection: Ambiguity is resolved by client. Clients need to know implementation details. C++ does it like this with the scope operator.
- Merging Methods: Related inherited methods can be merged into one overriding method. This does not work for fields and unrelated methods. The subclass should provide both methods, but with different names.
- Renaming: Inherited methods can be renamed. Dynamic binding takes renaming into account. This is the cleanest solution and is used by Eiffel and C++/CLI, a dialect of C++.

3.3.4 Repeated Inheritance: How many address fields should PhDStudent have? How are they initialized?

```
class Person{
    Address address;
    ...
};
class Student: public Person{
    ...
};
class Assistant: public Person{
    ...
};
class PhDStudent: public Student, public Assistant{
    ...
};
```



Virtual inheritance:
Default in Eiffel.
Virtual inheritance in C++.



Non-Virtual Inheritance:
Renaming field in Eiffel.
Default in C++.

3.3.5 Inheritance and Object Initialization: Normally, superclass fields are initialized before subclass fields. This call can also be implicit like in Java. This helps preventing use of uninitialized fields. Order is typically implemented via mandatory call of superclass constructor at the beginning of each constructor.

- Non-Virtual Inheritance: With non-virtual inheritance, there are two copies of the superclass field. Superclass constructor is called twice to initialize both copies.
- Virtual Inheritance: With virtual inheritance, there is only one copy of the superclass field. Who gets to call the superclass constructor? If both do it, we get a number of problems. First, we could have two different arguments to the constructor. Second, final fields would get initialized twice and thirdly, there could be side-effects in the constructor. The solution in C++ is that the smallest subclass (e.g. PhDStudent in the example) needs to call the constructor of the virtual superclass directly. So programmers need foresight and constructors cannot rely on the virtual superclass constructors they call. Eiffel does not force constructors to call superclass constructors. So subclasses have to initialized inherited fields. Subclasses also need to understand the implementation of the whole superclass tree. The policy is to always call the superclass "init" method, then constructors of repeated superclasses get called twice. This can be problematic.

3.3.6 Summary:

Pros	**Cons**
Increases expressiveness	Ambiguity resolution
Avoids overhead of delegation pattern	Repeated inheritance
	Complicated!

3.4 Linearization

3.4.1 Mixins and Traits: Mixins and traits provide a form of reuse. Methods and state can be mixed into various classes. Main applications are to make thin interfaces thick and stackable specifications. To avoid multiple inheritance among classes, the class must be a subclass of its traits' superclasses. Each trait defines an abstract type. Extending or mixing-in a trait introduces a subtype relation. Traits can be mixed-in upon class declaration or upon

class instantiation. Languages that support mixins or traits: Python, Ruby, Scala, Squeak Smalltalk.

3.4.2 Scala Trait Example:

```
class Cell{
    var value: Int = 0

    def put(v: Int) = {value = v}
    def get: Int = value
}
trait Backup extends Cell{
    var backup: Int = 0

    override def put(v: Int) = {
        backup = value
        super.put(v)
    }
    def undo = {super.put(backup)}
}
```

```
object Main1{
    def main(args: Array[String]){
        val a = new Cell with Backup
        a.put(5)
        a.put(3)
        a.undo
    }
}

class FancyCell extends Cell with Backup{
    ...
}
```

3.4.3 Ambiguity Resolution: Ambiguity is resolved by merging. If two inherited methods override a common superclass method, merging is not required.

3.4.4 Linearization: The key concept to understand the semantics of Scala traits: bring types in a linear order. Define overriding and super-calls according to this order. For a class or template Sub extends Super with T1 ... with Tn, we have the linearization

$$L(Sub) = Sub, L(Tn) \bullet \ldots \bullet L(T1) \bullet L(Super)$$

with

$$\epsilon \bullet B = B \qquad (A, B) \bullet (C) = \begin{cases} A, (B \bullet C) & if A \notin C \\ A \bullet C & \text{otherwise} \end{cases}$$

So every class shows up exactly once in the linearization. Subclass inherits only one copy of a repeated superclass. Like Eiffel and virtual inheritance in C++. Classes and traits are initialized in the reverse linear order. Each constructor is called exactly once. Arguments to superclass constructors are supplied by the immediately preceding class (not trait) in the linearization order.

3.4.5 How to do Linearization:

1. Put main class first.
2. For the rest of the classes and traits, go from right to left.
3. For each of those, do the first two points recursively
4. Go from right to left and ignore all that we have already.

Example:

```
class A { print("A") }
class B extends A { print("B") }
trait C { print("C") }
```

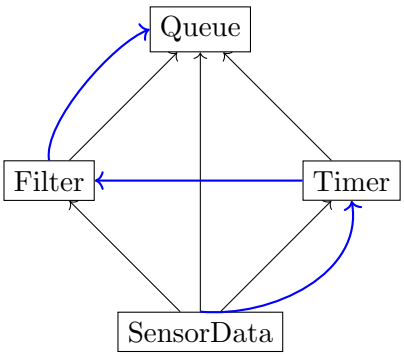
```
trait D extends C { print("D") }
trait E { print("E") }
trait F extends E with C { print("F") }
class X extends B with F with D { print("X") }

new X
```

1. Put main class first:
$$X$$
2. For the rest of the classes and traits, go from right to left:
$$X \bullet D \bullet F \bullet B$$
3. For each of those, do the first two points recursively:
$$X \bullet (D, C) \bullet (F, C, E) \bullet (B, A)$$
4. Go from right to left and ignore all that we have already:
$$X, D, F, C, E, B, A$$

Note that the command `new X` would print the reversed linearization order when run.

3.4.6 Stackable Specializations: With traits, specializations can be combined in flexible ways. With multiple inheritance, methods of repeated superclasses are called twice. Example

```
class Queue{
    def put(x: Data){...}
}
trait Timer extends Queue{
    override def put (x: Data){
        x.SetTime(...);
        super.put(x);
    }
}
trait Filter extends Queue{
    override def put(x: Data){
        if(x.Time > ...);
        super.put(x);
    }
}
class SensorData extends Queue
    with Filter with Timer{}
```



In Eiffel or C++, we would either only get one specialization or put the object into the queue twice. But we need to derive the order of the traits, so we need the code of them.

3.4.7 Reasoning about Traits: Traits are very dynamic, which complicates state reasoning. Traits do not know which methods they override. Traits do not know where super-calls are bound to. Also two classes with the same traits but in another order have the same type. This is a mistake!

4 Types

4.1 Bytecode Verification

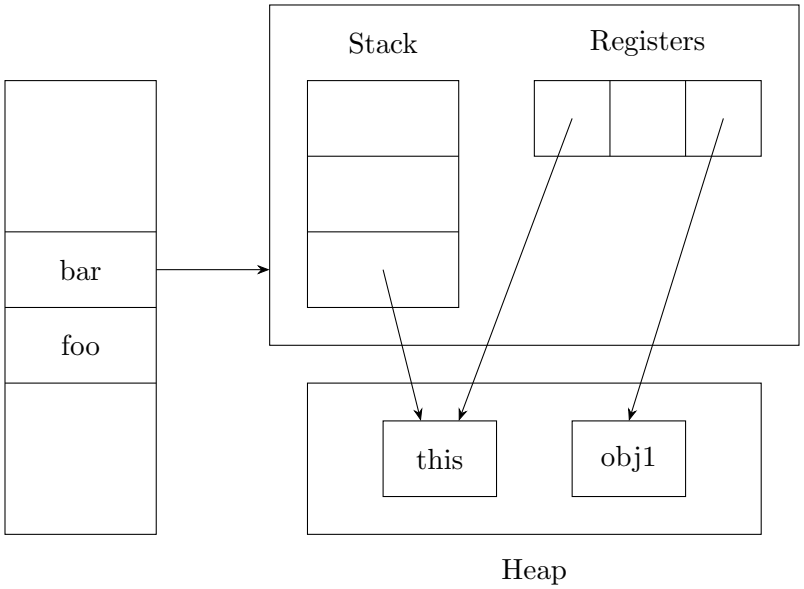
4.1.1 Mobile Code: In distributed computing, code mobility is the ability for running programs, code or objects to be migrated (or moved) from one machine or application to another. This is the process of moving mobile code across the nodes of a network as opposed to distributed computation where the data is moved. Examples of code mobility include scripts downloaded over a network (for example JavaScript), Java applets, Flash animations, Shockwave movies, and macros embedded within Microsoft Office documents.

4.1.2 Class Loaders: Programs are compiled to bytecode. This is a platform-independent format organized into class files. The bytecode is interpreted on a virtual machine. Class loader gets code for classes and interfaces on demand. Programs can contain their own class loaders.

4.1.3 Security for Java Programs: Sandbox: Applets get access to system resources only through an API, access control can be implemented. This security relies on type safety and that the code does not by-pass the sandbox.

4.1.4 Security in Mobile Environments: Mobile code cannot be trusted. Code may not be type safe, code may destroy or modify data, code may expose personal information, code may crash the underlying VM, code may purposefully degrade performance.

4.1.5 Java Virtual Machine: JVM is stack-based. Most operations pop operations from a stack and push a result. Registers store method parameters and local variables. Stack and registers are part of the method activation record. So in Java there are not only compile-time and run-time checks but also load-time checks in between (that is the bytecode verification part).



4.1.6 Java Bytecode: Instructions are typed (not like in Assembly where they are untyped). Load and store instructions access registers. Control is handled by intra-method branches. The compiler figures out the maximum stack size (MS) and the maximum number of regis-

ters needed (MR) and puts that information into the class file to be available at load time. Example:

```
iconst 5     //defines int 5 on the stack
istore 1     //stores the int on the stack in register 1
aload 0      //loads the reference from register 0 into the stack
astore 2     //stores the reference on the stack in register 2
return       //returns
```

4.1.7 Bytecode Verification: Proper execution requires that each instruction is type correct, only initialized variables are read, no operand stack over- or underflow occur, etc. The JVM guarantees these properties by bytecode verification when a class is loaded and by dynamic checks at runtime.

4.1.8 First Possibility: Bytecode Verification via Type Inference: The bytecode verifier simulates the execution of the program. Operations are performed on types instead on values. For each instruction, a rule describes how the operand stack and local variables are modified. Errors are denoted by the absence of a transition. This can happen if we have a type mismatch or a stack over- or underflow. Types of the Inference Engine: Primitive types (int, float, not boolean), object and array reference types, null type, $\top$ for uninitialized registers. Example rules:

$$i : (S, R) \to (S', R')$$

$$iadd : (int.int.S, R) \to (int.S, R)$$

$$iconst\ n : (S, R) \to (int.S, R),\ \text{if } |S| < MS$$

$$iload\ n : (S, R) \to (int.S, R),\ \text{if } 0 \leq n < MR \wedge R(n) = int \wedge |S| < MS$$

$$astore\ n : (t.S, R) \to (S, Rn \to t),\ \text{if } 0 \leq n < MR \wedge t <: Object$$

$$invokevirtual\ C.m.\sigma : (t_n ... t_1.t.S, R) \to (r.S, R),\ \text{if } \sigma = r(t_1, ..., t_n) \wedge t <: C \wedge t_i <: t_i$$

4.1.9 Smallest Common Supertype: Branches lead to joins in control flow. Instructions can have several predecessors. Smallest common supertype is selected ($\top$ if no other common supertype exists).

4.1.10 Handling Multiple Subtyping: With multiple subtyping, several smallest common supertypes may exist. The JVM solution is to ignore interfaces and treat all interface types as Object. This works because of single inheritance of classes. The problem is that $invokeinterfaceI.m$ cannot check whether the target object implements I and a run-time check is necessary.

4.1.11 Type Inference Algorithm:

```
in(0) := ([],[P_0,...,P_n,⊤,...,⊤])
//P_0 = this, P_1,...,P_n = parameter types.
worklist := {i | instr_i is an instruction of the method}
while worklist ≠ ∅ do
    i := min(worklist)
    remove i from worklist
    out(i) := apply_rule(instr_i, in(i))
    foreach q in successors(i) do
        in(q) := pointwise_scs(in(q), out(i))
        if in(q) has changed then
```

```
            worklist := worklist ∪ {q}
    end
end
pointwise_scs([s_1,...,s_k],[t_0,...,t_n],([s'_1,...,s'_k],[t'_0,...,t'_n]))
    = ([scs(s_1,s'_1),...,scs(s_k,s'_k)], [scs(t_0,t'_0),...,scs(t_n,t'_n)])
pointwise_scs(λ, out(i)) = out(i)
```

`pointwise_scs` is undefined for stacks of different heights.

4.1.12 Summary of Type Inference:

Advantages	Disadvantages
Determines the most general solution that satisfies the typing rules	Fixpoint computations may be slow.
Might be more general than what is permitted by the compiler	Solution for interfaces is imprecise and requires run-time checks
Very little type information required in class file	

4.1.13 Second Possibility: Bytecode Verification via Type Checking: Extend the class file to store type information. The compiler figures out the types for each start of a basic block, which are either a jump target or the entry points of the exception handler. The computation of the scs is no longer necessary. This avoids fixpoint computation and the interface problem.

4.1.14 Type Checking Algorithm:

```
foreach basic block of a method body do
    in := types(start)
    foreach {i | inst_i is an instruction of basic block} do
        in := apply_rule(instr_i, in)
        foreach q in successors(i) do
            if types(q) is declared then
                check that in is assignable to types(q)
            end
        end
    end
end
```

4.1.15 Until here for midterm on 9.11.18:

4.1.16 Type Inference for Source Programs: Type inference can also be done on source code. For example, C# 3.0 and Scala infer types of local variables. This reduces annotation overhead, especially with generics. Type annotations can still be used to support inference. Example from Scala:

```
def sum(a: Array[Int]): Int = {
    val it = a.elements //this is a constant
    var s = 0           //this is a mutable variable
    while (it.hasNext) {s = s + it.next}
}
def client = {
    var a = 1
```

```
    a = "Hello"
    //this does not work, the compiler just looks at the
    //first assignment and infers the type from that
}
def client = {
    var a:Any = 1
    a = "Hello"
    //this works but cannot call any Int or String methods on a
}
```

4.1.17 Type Inference vs. Dynamic Typing: Type inference determines the static type automatically and the performs static type checking. Dynamic typing does not require a static type and does not perform static type checking.

4.1.18 Inference of Method and Field Types: Inference of method signatures generally requires knowledge of all implementations. Inference of field types generally requires knowledge of all assignments to the field. Thus inference of these types is non-modular or based on speculation.

4.2 Parametric Polymorphism i.e. Generics

4.2.1 Polymorphism: Not all polymorphic code is best expressed using subtype polymorphism: Recovering precise information requires downcasts and subtype relations are sometimes not desirable, like in covariant arrays.

4.2.2 Parametric Polymorphism: Classes and methods can be parametrized with types. Clients provide instantiations for type parameters. The generic code is checked once and for all without knowing the instantiations. So it has to be type-safe for all possible instantiations.

4.2.3 Type Checking Generic Code: Type checking a generic class often requires information about its type arguments. Constraints can be expressed by specifying upper bounds on type parameters. Example from Java:

```
interface Comparable<T> {
    int compareTo(T o);
}
class Queue<T extends Comparable<T>>{
    T elem;
    Queue<T> next;
    void enqueue(T e){
        if (next == null){...}
        else {
            if (e.compareTo(elem) <= 0){
                next.enqueue(elem);
                elem = e;
            } else next.enqueue(e);
        }
    }
}
```

4.2.4 Subtyping and Generics: Generic types are subtypes of their declared supertypes. Type variables are subtypes of their upper bounds. For different instantiations of the same generic class, we have three possibilities:

- Covariant Type Arguments: $S <: T \Rightarrow C<S> <: C<T>$, completely analogous to the Java array situation. Covariance is unsafe when a generic type argument is used for variables that are written by clients, e.g. mutable fields and method arguments. This is the solution in Eiffel, which is consistent, but consistently bad.

```
class Queue<T>{
    void enqueue(T e){...}
    T dequeue(){...}
}
void put (Queue<Object> q){
    q.enqueue("Hello"};
    //not type safe if q has type Queue<Integer>
}
```

- Contravariant Type Arguments: $S <: T \Rightarrow C<T> <: C<S>$. Contravariance is unsafe when a generic type argument is used for variables that are read by clients, e.g. fields and method results.

```
String get (Queue<String> q){
    return q.dequeue();
    //not type safe if q has type Queue<Object>
}
```

- Non-Variance: Generic types in Java, C# and Scala are non-variant. This is statically type safe, there are no run-time checks needed. But non-variance is sometimes overly restrictive.

4.2.5 Generics vs. Arrays: An array T[] is not much different from a class Array¡T¿. But covariant generics would need run-time checks for field updates and argument passing while covariant arrays only need run-time checks for updates.

4.2.6 Variance Annotations: In Scala, programmers can supply variance annotations to allow co- and contravariance. The type checker imposes restrictions on the use of variance annotations.

- Covariance Annotations: A covariance annotation (+) is useful when the type variable occurs only in positive positions, i.e. as a result type or the type of an immutable field. The type checker prevents other occurrences.

```
class Random[+T]{
    def next: T = {...}
    def initialize(i: T) = {...}
    //this would be forbidden by the type checker
}
```

- Contravariance Annotations: A contravariance annotation (-) is useful when the type variable occurs only in negative positions, i.e. as a parameter type. The type checker prevents other occurrences.

```
class OutputChannel[-T]{
```

```
        def write(x: T) = {...}
        def lastWritten: T = {...}
        //this would be forbidden by the type checker
    }
```

4.2.7 Working with Non-Variant Generics: How can we write code that works with many different instantiations of a generic class?

4.2.8 Solution 1: Additional type parameters: This is the solution in 90% of the cases.

```
static <T> void printAll (Collection<T> c){
    for(T e: c) {System.out.println(e);}
}
foo(Collection<String> p){
    printAll(p);
}
```

But in the following case this does not work:

```
interface Comparator<T>{
    int compare(T fst, T snd);
}
class TreeSet<E>{
    TreeSet (Comparator<E> c){...}
}
class Person{...}
class Student extends Person{...}
class PersonComp implements Comparator<Person>{
    int compare (Person fst, Person snd){...}
}
TreeSet<Student> s = new TreeSet<Student>(new PersonComp());
//this gives a compile-time error
//because PersonComp is not a subtype of Comparator<Student>
```

We could fix it by adding an additional parameter:

```
class TreeSet<F, E extends F>{
    TreeSet (Comparator<F> c){...}
}
TreeSet<Person, Student> s
        = new TreeSet<Person, Student> (new PersonComp());
```

But the additional type argument is a nuisance for clients and reveals implementation details. Also the type-instantiation of Comparator cannot be changed at runtime, for instance to Comparator¡Object¿.

4.2.9 Solution 2: Wildcards: A wildcard represents an unknown type. Interpret it as "There exists a type argument T such that c has type Collection¡T¿".

```
static Collection<?> id (Collection<?> c){
//the ? are two different existential types.
    return c;
}
```

```
Collection<String> c = new ArrayList<String>();
Collection<String> d = id(c);
//this does not compile
```

The compiler may assume: $\exists T : type(c) <: Collection < T >$

The compiler needs to show: $\exists S : type(c) <: Collection < S >$

? is not fixed. If it was, we would have the problem, that we would need to know every caller before being able to set the type. We can also have constrained wildcards like we have seen before, but additionally we can also have lower bounds. Java does not support lower bounds for type parameters.

```
class Cell<T> {
    T value;
    void copyFromT(Cell<T> other){
        value = other.value;
    }
    void copyFrom(Cell<? extends T> other){ //"covariance"
        value = other.value;
    }
    void copyTo(Cell<? super T> other){      //"contravariance"
        other.value = value;
    }
}
```

This allows clients to decide on variance at the use site, as opposed to Scala's declaration-site variance. The bounds for a wildcard determine the set of possible instantiations. For types Sub and Super with the same class or interface, Sub is a subtype of Super if for each type argument, the set set of possible instantiations for Sub is a subset of the set of possible instantiations for Super. Whether Sub is a subtype of Super is undecidable in Java due to this. In fact, Java generics are Turing-complete.

4.2.10 Type Erasure: Java introduced generics in 1.4 and because of backwards compatibility, Sun did not want to change the virtual machine. Thus generic type information is erased by the compiler. The type is translated to its upper bound and we add casts where necessary. Only one classfile and only one class object represent all instantiations of a generic class. Because of this, we can't have a number of things:

- Generic types are not allowed with instanceof, because at run-time we do not have the necessary information.
- Class object of generic types is not available, because it does not exist.
- Arrays of generic types are not allowed. This is because of a combination of two mistakes: First we need a run-time check, because arrays are covariant. But because the run-time check is like a instanceof, we cannot actually do the check.
- Strange compiler errors:

```
void main(){
    Cell<Object> co = new Cell<Object>();
    co.value = new Integer(5);
    demo(co);
```

```
}
String demo (Cell<?> c){
    Cell<String> cs = (Cell<String>) c;      //this will work fine
    ...
    return cs.value;                         //compiler will say here that c
}
```

This is because demo looks like this in bytecode:

```
String demo (Cell c){
    Cell cs = (Cell) c;
    ...
    return (String) cs.value;
}
```

- Also, static fields are shared by all instantiations of a generic class.

C# changed the bytecode and didn't care about backwards compatibility. Thus in C# we can do all of those things.

4.2.11 C++ Templates: Templates allow classes and methods to be parametrized. Clients provide instantiations for template parameters.

```
template<class T> class Queue{
    T elem;
    Queue<T>* next;
public:
    void enqueue (T e){...};
    T dequeue (){...}
};
Queue<int> *q;
q = new Queue<int>();
```

The compiler generates a class for given template instantiations. Type checking is done for the generated class, not for the template. So the template code is not type checked, you cannot guarantee type safety. The compiler does not check the availability of methods and only compiles and type checks the methods that are actually used in client code. So one has to type check a library extensively before giving it to a client, effectively testing all possible instantiations. There is no need for upper bounds on type parameters, because the availability of methods is not checked anyway. Different instantiations of templates are unrelated.

4.2.12 Template Meta-Programming: Template parameters need not be types, they can be values. We can also specialize templates, to be used in case a certain value is given. With this, we can let the compiler compute values because they are actually constants. See the following example:

```
template<int n> class Fact{
public:
    static const int val = Fact<n-1>::val* n:
};
template<> class Fact<0>{
public:
    static const int val = 1;
```

```
};
int main() {
    printf("factorial 3 = %d\n", Fact<3>::val);
    //the compiler generates Fact<3>::val
    return 0;
}
```

5 Information Hiding and Encapsulation

5.1 Information Hiding

5.1.1 Information Hiding: Information hiding is a technique for reducing the dependencies between modules: The intended client is provided with all the information needed to use the module correctly and with nothing more. The client uses only the (publicly) available information.

5.1.2 Objectives:

- Establish strict interfaces
- Hide implementation details
- Reduce dependencies between modules

5.1.3 Different Interfaces of a Class:

- Client Interface: The client sees the class name, the type parameters and their bounds, the super class, the super interfaces, the signatures of exported methods and fields and the client interface of the direct superclass.
- Subclass Interface: The subclass also has efficient access to the superclass fields and to auxiliary superclass methods.
- Friend Interface: The friend also has mutual access to implementations of cooperating classes.

5.1.4 Java Access Modifiers:

- public: client interface
- protected: subclass interface and friend interface
- Default access: friend interface
- private: implementation (i.e. all objects in class)

Eiffel also has "none", where only the "this"-object can access.

5.1.5 Why Information Hiding?: Because we want safe changes. We can consistently rename hidden elements. We can modify hidden implementations as long as the exported functionality is preserved. (Still have to keep fragile baseclass problem in mind). We know which classes might be affected by a change because of access modifiers.

5.1.6 Method Selection in Java:
At compile time:

1. Determine static declaration
2. Check accessibility
3. Determine invocation mode (virtual/non-virtual)

At run time:

1. Compute receiver reference
2. Locate method to invoke that overrides statically determined method

5.1.7 Problem with Protected Methods: Protected in the subclass does not always provide at least as much access as protected in the superclass. With this, we can access methods that should not be accessible from where we are. Also, while private methods are statically bound, protected methods aren't, which can lead to methods now overriding, where they weren't before. In C#, this would not happen, because we happen, because we have to explicitly declare overrides.

5.2 Encapsulation

5.2.1 Encapsulation: Encapsulation is a technique for structuring the state space of executed programs. Its objective is to guarantee data and space consistency by establishing capsules with clearly defined interfaces. Capsules can be individual objects, object structures, a class, all classes of a subtype hierarchy or even a package. Encapsulation requires a definition of the boundary of a capsule and the interfaces at the boundary.

5.2.2 Objective: A well-behaved module operates according to its specification in any context in which it can be reused. Implementations rely on consistency of internal representations. Reuse contexts should be prevented from violating consistency.

5.2.3 Consistency of Objects: Objects have external interfaces and internal representations. The internal representation of an object is encapsulated if it can be manipulated only by using the object's interfaces. Example: Exported fields allow objects to manipulate the state of other objects. We can apply information hiding to remedy that. But then subclasses can introduce new or overriding methods that break consistency. The solution to this is behavioral subtyping. To achieve consistency of objects, use the following guidelines:

1. Apply information hiding: Hide internal representation wherever possible.
2. Make consistency criteria explicit: Use contracts or informal documentation to express consistency criteria.
3. Check interfaces: Make sure that all exported operations of an object, including subclass methods, preserve all documented consistency criteria.

5.2.4 Checks for Invariants: Textbook Solution: Assume that all objects o are capsules and only methods executed on o can modify o's state. The invariant of object o refers only to the encapsulated fields of o. For each invariant we have to show that all exported methods preserve the invariants of the receiver object and that all constructors establish the invariants of the new object.

5.2.5 Checks for Invariants: Java Simple Solution: In Java, declaring all fields private does not guarantee encapsulation on the level of individual objects, because objects of the same class can break the invariant. Thus we simply assume that the invariants of object o may refer only to private fields of o. Then for each invariant we have to show that all exported methods and constructors of class T preserve the invariants of all objects of T and that all constructors in addition establish the invariants of the new object. This is practically not possible, because first we would need to find all objects of T.

6 Object Structures and Aliasing

6.0.1 Object Structures: Objects are the building blocks of object-oriented programming. However, interesting abstractions are almost always provided by sets of cooperating objects. An object structure is a set of objects that are connected via references.

6.1 Aliasing

6.1.1 Aliasing: A name that has been assumed temporarily. In programming we use it when several variables refer to the same memory location. This can lead to unexpected side-effects.

6.1.2 Static Aliasing: An alias is static if all involved variables are fields of objects or static fields, i.e. if all pointers to the object live in the heap.

6.1.3 Dynamic Aliasing: An alias is dynamic if it is not static, i.e. if at least one of the pointers to the object lives on the stack.

6.1.4 Intended Aliasing: Aliasing can be intentional: not copying data structures makes OO-programming efficient. Also, objects can be shared to make modifications to a shared state.

6.1.5 Unintended Aliasing:

- Capturing occurs when objects are passed to a data structure and then stored by the data structure. The problem is that an alias can be used to by-pass the interface of the data structure.
- Leaking occurs when data structures pass a reference to an object which is supposed to be internal to the outside. The problem is again that the alias can be used to by-pass the interface of the data structure.

6.2 Problems of Aliasing

6.2.1 Leaking: Difficult to prevent: Information hiding is not applicable to arrays, restriction of identity objects is not effective, read accesses are permitted and run-time checks are too expensive.

6.2.2 Other Problems:

- Synchronization in concurrent programs: The monitor of each individual object has to be locked to ensure mutual exclusion.
- Distributed programming: For instance parameter passing for remote method invocation.
- Optimizations: For instance object inlining is not possible for aliased objects.

6.2.3 Alias Control in Java:

- LinkedList: All fields are private. Entry is a private inner class of LinkedList, which is good because then subclasses cannot leak Entry-objects, but they also cannot manipulate them which is bad. The ListItr is a private inner class of LinkedList, so again subclasses cannot leak ListItr-objects, but they also cannot manipulate them which is bad. So all in all subclassing is severely restricted.

- String: All fields are private. References to internal character-array are not passed out. Subclassing is prohibited.

6.3 Readonly Types

6.3.1 Drawbacks of Alias Prevention: Aliases are helpful to share side-effects. Also cloning is not efficient. In many cases it suffices to restrict access to shared objects, commonly granting read access only.

6.3.2 Requirements for Readonly Access: We have mutable objects, but some clients can mutate the object while others cannot. Thus the access restrictions apply to the references, not the whole object. We prevent field updates and calls of mutating methods. We also need transitivity, so that access restrictions extend to references to sub-objects.

6.3.3 First Solution via Supertypes: We create a readonly interface and then a mutable class that implements it. The object remains mutable and we have no field updates and no mutating methods in the interface. But reused classes might not implement a readonly interface. Also interfaces do not support arrays, fields and non-public methods. Also transitivity has to be encoded explicitly and requires sub-objects to implement readonly interfaces. This solution is also not safe. There are no checks that methods in the readonly interface are actually side-effect free. And clients can just use casts to get full access!

6.3.4 Second Solution via const Pointers: C++ supports readonly pointers, through which we cannot do field updates or mutator calls. But const-ness can just be cast away! Also const pointers are not transitive, the const-ness of sub-objects has to be indicated explicitly.

6.3.5 Third Solution via Pure Methods: We tag side-effect free methods as pure. These must not contain field updates, must not invoke non-pure methods, must not create objects and can be overridden only by pure methods. Each class or interface T introduces two types: a readwrite (rw) type (denoted "T") and a readonly (ro) type (denoted "readonly T").

- Subtyping among readwrite and readonly types is as follows:

$$rw\ T\ <:\ ro\ T$$

$$S\ <:\ T \Rightarrow rw\ S\ <:\ rw\ T$$

$$S\ <:\ T \Rightarrow ro\ S\ <:\ ro\ T$$

- Accessing a value of a readonly type or through a readonly type should only yield a readonly value, so we have transitivity. If the access is through a receiver, we only give a readwrite value if the receiver and the field are of readwrite type. In all other cases we give a readonly value.
- Expressions of readonly types must not occur as receivers of field updates, array updates or an invocation of a non-pure method. Also readonly types must not be cast to readwrite types. To do this we disallow downcasts entirely.

Readwrite aliases can still occur, for example by capturing.

6.4 Ownership Types

6.4.1 Object Topologies: To fix the leaking and capturing problem with pure methods, we need to distinguish internal references from other references.

6.4.2 Roles in Object Structures:

- Interface objects that are used to access the structure
- Internal representation of the object structure, these must not be exposed to clients
- Arguments of the object structure, these must not be modified

6.4.3 Ownership Model: Each object has zero or one owner objects. The set of objects with the same owner is called a context. The ownership relation is acyclic. The heap is thus structured into a forest of ownership trees. We use types to express ownership information:

- "peer" types for objects in the same context as "this"
- "rep" types for representation objects in the context owned by "this"
- "any" types for argument objects in any context

Example:

```
class LinkedList{
    private rep Entry header;
    ...
}
class Entry {
    private any Object element;
    private peer Entry previous, next;
    ...
}
```

6.4.4 Type Safety: Run-time information consists of the class of each object and the owner of each object. The type invariant we have is as follows: The static ownership information of an expression e reflects the run-time owner of the object o referenced by e's value. If e has type "rep T", then o's owner is "this". If e has type "peer T", then o's owner is the owner of "this". If e has type "any T", then o's owner exists, but we don't know who it is.

6.4.5 Subtyping and Casts: For types with identical ownership modifier, subtyping is defined normally:

$$S <: T \Rightarrow rep\ S <: rep\ T$$

$$S <: T \Rightarrow peer\ S <: peer\ T$$

$$S <: T \Rightarrow any\ S <: any\ T$$

We also have the following additional rules:

$$rep\ T <: any\ T$$

$$peer\ T <: any\ T$$

We can check these at run-time because every object stores its owner. We define the owner when creating the object and it stays the same for the whole lifetime of the object. The ownership information is relative to "this".

6.4.6 The lost Modifier: Some ownership relations cannot be expressed in the type system. We have the internal modifier "lost" for a fixed but unknown owner. Reading locations with lost ownership is allowed, but updating them is unsafe.

6.4.7 Field Access Type Rules: A field read $v = e.f$ is correctly typed if e is correctly typed and $\tau(e) \blacktriangleright \tau(f) <: \tau(v)$. The field write $e.f = v$ is correctly typed if e is correctly typed, $\tau(v) <: \tau(e) \blacktriangleright \tau(f)$ and $\tau(e) \blacktriangleright \tau(f)$ does not have "lost" modifier. We have analogous rules for method invocations: Argument passing is analogous to field write, result passing is analogous to field read.

6.4.8 The self Modifier: Internal modifier only for the "this" literal. We have an additional subtyping rule

$$self\ T <: peer\ T$$

This gives us the finished table for field access modifiers:

▶	peer T	rep T	any T
peer S	peer T	lost T	any T
rep S	rep T	lost T	any T
any S	lost T	lost T	any T
lost S	lost T	lost T	any T
self S	peer T	rep T	any T

We fixed capturing, but leaking is still possible.

6.4.9 Owner-As-Modifier Discipline: Based on the ownership type system we can strengthen encapsulation with extra restrictions to prevent modifications of internal objects. We treat any and lost as readonly types, while treating self, peer and rep as readwrite types. We now have additional rules: The field write $e - f = v$ is valid only if $\tau(e)$ is self, peer or rep. The method call $e.m(...)$ is valid only if $\tau(e)$ is self, peer or rep, or the called method is pure. A method may now modify only objects directly or indirectly owned by the owner of the current "this" object. A call stack now always has to include at least all owners. With this we now have also no leaking.

6.4.10 Achievements: rep and any types enable encapsulation of whole object structures. Encapsulation cannot be violated by subclasses , via casts, etc. The technique fully supports subclassing in contrast to solutions with private inner or final classes. Ownership types express heap topologies and enforce encapsulation. Owner-as-Modifier is helpful to control side effects and with this maintain object invariants and prevent unwanted modifications. Other applications also need restrictions of read access to enable the exchange of implementations and for thread synchronization.

7 Initialization

7.1 Simple Non-Null Types

7.1.1 Main Usage of Null-References: Null is used to terminate a recursion, like in a linked list. It is also used to initialized fields and to indicate the absence of an object for example when checking if an object is in a data structure. But most variables hold non-null values. Studies show that 70% of variables are always non-null.

7.1.2 Non-Null Types: Non-null type T! consists of references to T-objects. Possibly-null type T? consists of references to T-objects plus null. This corresponds to the usage of T in most languages. A language designer would need to choose a default, T? for backwards compatibility, but T! would be better.

7.1.3 Type Safety: (Simplified) type invariant: If the static type of an expression e is a non-null type then e's value at run time is different from null. The goal is to prevent null-dereferencing statically. We require non-null types for the receiver of each field access, array access and method call. This is analogous to preventing "method not understood" errors with classical type systems.

7.1.4 Subtyping and Casts: The values of a type T! are a proper subset of T?

$$S <: T \Rightarrow S! <: T!$$

$$S <: T \Rightarrow S? <: T?$$

$$T! <: T?$$

Downcasts from possibly-null types to non-null types require run-time checks.

7.1.5 Type Rules: Most type rules of Java remain unchanged, but we have an additional requirement: expressions whose value gets dereferenced at run-time must have a non-null type. These are the receiver of a field access, the receiver of an array access, the receiver of a method call and the expression of a throw statement.

7.1.6 Code Example:

```
class Map {
    Map? next;
    ...
    Object? get (Object! k){
        ...
        Map? n = next;
        if (n==null) return null;
        return n.get(k);
    }
}
```

```
class Map {
    Map? next;
    ...
    Object? get (Object! k){
        ...
        Map? n = next;
        if (n==null) return null;
        return ((Map!) n).get(k);
    }
}
```

The second one is already better, but we would like to be able to automatically use n as a non-null type without the cast.

7.1.7 Data Flow Analysis: Data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph is used to determine those parts of a program to which a particular

value assigned to a variable might propagate. With this technique we can for example guarantee that the call on n in the previous example is safe. If we did not assign next to n, but instead checked next directly for non-nullness, the data flow analysis actually would reject the previous example. This is because between the check for non-nullness and the dereferencing of the object the object could change to null. Because of this, data flow analysis does not even track values of heap locations, only of local variables.

7.2 Object Initialization

7.2.1 Constructing New Objects: The idea is to make sure that all non-null fields are initialized when the constructor terminates. We cannot rely on non-nullness of fields of objects under construction. The definite assignment rule (every local variable must be assigned to before it is first used) is used for local variables in Java and C#. We now want to apply this rule for fields in the constructor.

7.2.2 Problems:

- Method calls: A subclass calls the constructor of the superclass first thing in its own constructor. If the superclass has a dynamically bound method call, this call might now access fields that have not been initialized.
- Escaping via method calls: maybe the superclass calls a method that adds the object being created to a list. Then operations on this list might happen while we are still constructing the new object which might access fields that have not been initialized.
- Escaping via field updates: we might also have attached the object being created to a object that is already created. Over that object we might access the object still under construction and access fields that have not been initialized.

Thus the simple definite assignment checks for fields are sound only if partially-initialized objects do not escape from the constructor. They shall not be passed as the receiver or argument to a method call and shall not be stored in a field or an array.

7.2.3 Tracking Object Construction: The idea is to design a type system that tracks which objects are under construction. For simplicity, we track whether the construction has completed even if all non-null fields may have been initialized earlier.

7.2.4 Construction Types: For every class or interface T, we introduce different types for references: first, to objects under construction, second to objects whose construction is completed. To achieve this, we introduce six types for each class or interface T: T! and T? (committed types), free T! and free T? (free types), unc T! and unc T? (unclassified types). For subtyping we have the normal !-? subtyping rules plus:

$$T! <: unc\ T!, free\ T! <: unc\ T!$$

$$T? <: unc\ T?, free\ T? <: unc\ T?$$

There are no casts from unclassified to free or committed types, because we cannot perform the necessary run-time checks. If we had those, we could have a cross-type alias where A has a committed pointer to an object C, while B has a free pointer to it. A would expect all objects in C to be committed, but B could write to them whatever it wants. This is problematic. We would need to check the whole class structure for committedness and the whole heap and stack to look for pointers to any of those objects in C, which is not feasible. Fields do not have a construction type, as also new expressions do not.

7.2.5 Requirements:

- Local initialization: An object is locally initialized if its non-null fields have non-null values. If the static type of an expression e is a committed type then e's value at run-time is locally initialized.
- Transitive initialization: An object is transitively initialized if all reachable objects are locally initialized. If the static type of an expression e is a committed type then e's value at run-time is transitively initialized.
- Cyclic structures: We can assign "this" to an object in the constructor. This object has not finished construction until also "this" has finished construction.

7.2.6 Type Rules: Field Write: A field write $e_1.f = e_2$ is well-typed if e_1 and e_2 are well-typed, e_1's type is a non-null type, e_2's class and non-null type conform to the type of $e_1.f$ and e_1's type is free or e_2's type is committed.

		Type of e_2		
		committed	free	unc
Type of e_1	committed	yes	no	no
	free	yes	yes	yes
	unc	yes	no	no

7.2.7 Type Rules: Field Read: A field read expression $e.f$ is well-typed if e is well-typed and e's type is a non-null type.

		Declared type of f	
		T!	T?
Type of e	S!	T!	T?
	free S!	unc T?	unc T?
	unc S!	unc T?	unc T?

7.2.8 Type Rules: Constructors: Constructor signatures include construction types for all parameters. The receiver has a free non-null type. Constructor bodies must assign non-null values to all non-null fields of the receiver. Now our invariant holds at every point.

7.2.9 Type Rules: Methods and Calls: Method signatures include construction types for all parameters. The construction type of the receiver is written after the return type. Calls are type checked as usual. Overriding requires the usual co- and contravariance.

7.2.10 Object construction: We do not actually know that the construction is finished when the constructor terminates, as there might be subclass constructors which have not yet executed. We also do not know that the construction is finished when the new-expression terminates, as the constructor might initialize fields with free references. We make an assumption that the new-expression only takes committed arguments, but nested new-expressions can take arbitrary arguments. After the new-expression, all new objects are locally initialized, so the new objects only reference transitively initialized objects and each other. Thus all objects are transitively initialized.

7.2.11 Type Rules: new-Expressions: An expression new $C(e_i)$ is well-typed if all e_i are well-typed and class C contains a constructor with suitable parameter types. The type of new $C(e_i)$ is committed $C!$ if the static types of all e_i are committed or free $C!$ otherwise.

7.2.12 Problems Revisited:

- Method calls: This now gives a compile-time error, as the construction type of the receiver of the method has to be declared as free and thus field accesses on it are not allowed.
- Escaping via method calls: We cannot add the object to the list, as it is not committed yet.
- Escaping via field updates: We cannot attach the object currently being created to an object that is already created, as it is not committed yet.

7.2.13 Lazy Initialization: Creating objects and initializing their fields is time consuming. This leads to a long application start-up time. With lazy initialization we initialize fields only just before they are first used. This spreads the initialization effort over a longer time period.

7.2.14 Non-Null Arrays: An array type describes two kinds of references: The reference to the array object and the references to the array elements. Both can be non-null or possibly-null. We thus get four different type combinations. The problem is that our solution for non-null fields does not work for non-null array elements, because there is no constructor for arrays and so they are typically initialized using loops which are ignored by static analyses because they are too hard to reason about. In general definite assignment cannot be checked by the compiler. There are a few partial solutions. First we could require arrays to be initialized with values when they are declared, but this is not practical. Second, we could prefill the array like Eiffel does, but this just replaces null with the default value which does not give us much functionality. Third we could insert an assertion that from this point on the array is initialized and check it at run-time. This is what Spec# does. We can then only store committed objects in the array because the committedness cannot be checked at run-time.

7.3 Initialization of Global Data

7.3.1 Global Data: Most software systems maintain global data to use for factories, caches, flyweights, singletons, ... The min issues are how do clients access the global data and how is it initialized. Our design goals are threefold. The most important one is effectiveness, to ensure that global data is initialized before the first access. The second one is clarity, to ensure that the initialization has a clean semantics and facilitates reasoning. The third one is laziness to ensure that global data is initialized lazily to reduce start-up time.

7.3.2 Solution 1: Global Vars and Init-Methods: Global variables store references to global data. The initialization is done by explicit calls to init-methods. These methods are called directly or indirectly from main-method. To ensure effective initialization, main needs to know the internal dependencies of all modules to know what variables it needs to initialize. This solution is very error-prone as everything has to be done manually. It also compromises information hiding and laziness has to be done manually. C++ does a variation of this, where initializers are executed before the execution of the main method. Thus we do not need explicit calls, but there is no support for laziness. Also the order of execution is determined by the order of appearance in the source code so the programmer still has to manage dependencies.

7.3.3 Solution 2: Static Fields and Initializers: Here we have static fields that store references to global data. The static initializers are executed by the system immediately before a class is used. This means a class C's static initializer runs immediately before the first creation of a C instance, a call to a static method of C, an access to a static field of C and before static initializers of C's subclasses. With this solution the system manages dependencies and we have some form of laziness. A problem that persists is when there are

mutual dependencies between two static initializers. There will be either an infinite loop or a run-time error. Another problem are side effects from static initializers. The code in an initializer can be arbitrary and as they are called automatically, we would need to know when the initializers are run to reason about programs, which makes it non-modular. Scala has singleton objects, but because the initialization of it is translated to a java class with a static initializer, it inherits all pros and cons of static initializers.

7.3.4 Solution 3: Eiffel's Once Methods: Once methods are executed only once. The result of the first execution is cached and returned for subsequent calls. In Eiffel, also static fields are once methods that return a pointer to the data. Mutual dependencies lead to recursive calls, which return the current values of the result which is typically not meaningful. Arguments to once method are used only for the first execution and are ignored for subsequent calls. This solution does give us full laziness though, as the once methods are only executed when the result is needed.

7.3.5 Summary: There is no solution that ensures that global data is initialized before it is accessed. There is also no solution that handles mutual dependencies.

8 Reflection

8.0.1 Reflection: A program can observe and modify its own structure and behaviour.

8.1 Introspection

8.1.1 Class Objects: In Java, one can get the class-object for a class by the predefined class-field. This class includes methods to get all methods of the class, to get the superclass, etc... Safety checks have to be made at run-time for example to see if an object actually has the field we are trying to access and if the client is allowed to access that field. This gives two downsides: we lose static safety and this is up to a factor ten slower. We can suppress Java's access checking by setting $setAccessible(true)$ for the field. One can also configure the security manager of the Java VM to disallow this setting.

8.1.2 Unit Testing: With introspection we can write a generic test driver that executes tests. This is the basic mechanism behind JUnit. We essentially create a new object of the class and run all methods on it. Creating a new object comes with some run-time checks to check if the class-object actually represents a concrete class, if the class has a parameter-less constructor and if the class and the parameter-less constructor are accessible.

8.1.3 Double Invocation Revisited: We can use the visitor pattern, but replace the accept methods at the classes being visited, because we can do all of that with dynamic method binding and finding the right method by going through the methods of "this". This gives us much simpler code and a flexible look-up mechanism. But it is not statically safe and much slower because of the run-time checks.

8.1.4 Java Generics: Generic type information is not represented at run-time, so we cannot us them in finding methods with introspection, because that happens at run-time.

8.2 Reflective Code Generation

8.2.1 Motivation: If code is represented as data, we can as well allow programs to create code from data. We can generate code dynamically according to user input and execution environment.

8.2.2 C# Expression Trees: Expression trees represent the abstract syntax tree of C# expressions. It can be created like any other data structure. The class Expression provides a compile-method which compiles expression trees to executable code. The main application is the generation of SQL queries.

8.3 Dynamic Code Manipulation

8.3.1 Motivation: If code is represented as data, we can as well allow programs to modify the code. We can adapt the program dynamically according to user input and the execution environment. We can do this for example in Python and use $setattr(class, methodName, newMethodCode)$. Dynamic code manipulation is only available in dynamically typed languages and typically requires dynamic checking.