Concepts of Object-Oriented Programming

Summary of the course in fall of 2010 by Prof. Dr. Peter Müller

Stefan Heule

2011-02-10

Table of Contents

1 Introduction 4
1.1 Requirements 4
1.2 Core Concepts 4
1.3 Language Design 4
2 Subtyping 5
2.1 Types 5
2.1.1 Weak and Strong Type Systems 5
2.1.2 Nominal and Structural Types 5
2.1.3 Type Checking 5
2.2 Subtyping 6
2.2.1 Nominal Subtyping and Substitution 6
2.2.2 Covariant Arrays 6
2.2.3 Shortcomings of Nominal Subtyping 7
2.2.4 Structural Subtyping and Substitution 7
2.2.5 Type Systems in OO-Languages 7
2.3 Behavioral Subtyping 7
2.3.1 Rules for Subtyping 8
2.3.2 Specification inheritance 8
2.3.3 Types as Contracts 9
2.3.4 Immutable Types 9
3 Inheritance 10
3.1 Inheritance and Subtyping 10
3.1.1 Problems with Subclassing 10
3.2 Dynamic Method Binding 11
3.2.1 Fragile Baseclass Problem 12
3.2.2 Binary methods 12
3.3 Multiple Inheritance 13
3.4 Inheritance and Object Initialization 13
3.5 Traits 15
3.5.1 Linearization 15
3.5.2 Reasoning about traits 16
3.5.3 Summary 16
4 Types 17
4.1 Bytecode Verification 17
4.1.1 Java Virtual Machine and Java Bytecode 17
4.1.2 Bytecode Verification via Type Inference 17
4.1.3 Bytecode Verification via Type Checking 19
4.2 Parametric Polymorphism 19
4.2.1 Wildcards 20
4.2.2 Method Type Parameters 21
4.2.3 Type Erasure 21
4.2.4 C++ templates 22
5 Information Hiding and Encapsulation 24
5.1 Information Hiding 24
5.2 Encapsulation 25
5.2.1 Consistency of Objects 25

5.2.2 Achieving Consistency of Objects 26
5.2.3 Invariants 26
6 Object Structures and Aliasing 27
6.1 Object Structures 27
6.2 Aliasing 27
6.2.1 Intended Aliasing 27
6.2.2 Unintended Aliasing 27
6.3 Problems of Aliasing 27
6.4 Encapsulation of Object Structures 28
6.4.1 Simplified Programming Discipline 28
7 Ownership Types 30
7.1 Readonly Types 30
7.1.1 Readonly Access via Supertypes 30
7.1.2 Readonly access in Eiffel 30
7.1.3 Readonly access in C++ via `const` pointers 31
7.1.4 Readonly Types and Pure Methods 31
7.2 Topological types 32
7.2.1 Owner-as-Modifier Discipline 34
7.2.2 Consequences 34
8 Initialization 36
8.1 Simple Non-Null Types 36
8.2 Object Initialization 36
8.2.1 Constructors Establish Type Invariant 36
8.2.2 Construction Types 37
8.3 Initialization of Global Data 39
9 Object Invariants 43
9.1 Call-backs 43
9.1.1 Spec# Solution 44
9.2 Invariants of Object Structures 45
10 Reflection 47
10.1 Introspection 47
10.1.1 Visitor-Pattern via Reflection 47
10.2 Reflective Code Generation 48
10.3 Summary 49
11 Language features 50
11.1 C++ 50
11.2 Eiffel 50
11.3 Java 50

1 Introduction

1.1 Requirements

We can classify requirements into four different categories

- Cooperating program parts with well-defined interfaces (e.g. quality, documented interfaces, modeling entities of the real world, communication, distribution of data and code)
- Classification and specialization (e.g. extendibility, adaptability, adaptable standard functionality, modeling entities of the real world)
- Highly dynamic execution model (e.g. communication, describing dynamic system behavior, running simulations, concurrency)
- Correctness (e.g. quality)

1.2 Core Concepts

- **Object model**. A software system is a set of cooperating objects with state and processing ability, where objects exchange messages.
- **Classification** is a general technique to hierarchically structure knowledge about concepts, items, and their properties. The result is called classification.
- **Substitution principle**. Objects of subtypes can be used wherever objects of supertypes are expected.
- **Polymorphism**. A program part is polymorphic, if it can be used for objects of several types.
    - Subtype polymorphism is a direct consequence of the substitution principle: Program parts working with supertype objects work as well with subtype objects.
    - Other forms of polymorphism (not core concepts)
        - Parametric polymorphism (generic types)
        - Ad-hoc polymorphism (method overloading)
- **Specialization**. Adding specific properties to an object or refining a concept by adding further characteristics.

1.3 Language Design

There are several design goals to be considered when designing a language. A good language should resolve design trade-offs in a way *suitable for its application domain*.

- **Simplicity**. Syntax and semantics can easily be understood by users of the language.
- **Expressiveness**. Language can (easily) express complex processes and structures.
- **(Static) safety**. Language discourages errors and allows errors to be discovered and reported, ideally at compile time.
- **Modularity**. Language allows modules to be compiled separately.
- **Performance**. Programs written in the language can be executed efficiently.
- **Productivity**. Language leads to low costs of writing programs.
- **Backwards compatibility**. Newer language versions work and interface with programs in older versions.

2 Subtyping

2.1 Types

- A **type system** is a tractable syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
    - Syntactic: Rules are based on form, not behavior.
    - Phrases: Expressions, methods, etc. of a program.
    - Kinds of values: Types.
- A **type** is a set of values sharing some properties. A value v has type T if v is an element of T.
    - These properties differ in different languages (e.g. nominal vs. structural type systems)

2.1.1 Weak and Strong Type Systems

- Untyped languages (e.g. assembler)
    - Do not classify values into types
- Weakly-typed languages (e.g. C, C++)
    - Classify values into types, but do not strictly enforce additional restrictions
- Strongly-typed languages (e.g. C#, Eiffel, Java, Python, Scala)
    - Enforce that all operations are applied to arguments of the appropriate types

2.1.2 Nominal and Structural Types

- Nominal types are based on *names* (e.g. C++, Eiffel, Java, Scala)
- Structural types are based on *the availability of methods and fields* (e.g. Python, Ruby or Small-talk)

2.1.3 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
- Dynamic type checking
    - Variables, methods, and expressions of a program are typically not typed
    - Every object and value has a type
    - The *run-time system* checks that operations are applied to expected arguments
- A programming language is called **type-safe** if its design prevents type errors.
- Statically type-safe object-oriented 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.
    - However, most static type systems rely on dynamic checks for certain operations, e.g. for type conversions by casts.

Advantages of static checking	Advantages of dynamic checking
- Static safety: More errors are found at compile time - Readability: Types are excellent documentation - Efficiency: Type information allows optimizations	- Expressiveness: No correct program is rejected by the type checker - Low overhead: No need to write type annotations - Simplicity: Static type systems are often complicated

2.2 Subtyping

- Substitution principle: Objects of subtypes can be used wherever objects of supertypes are expected.
- Syntactic classification: Subtypes can *understand at least the messages* that supertype objects can understand.
- Semantic classification: Subtype objects *provide at least the behavior* of supertype objects.
- We defined types as sets of values with some common property. The subtype relation then corresponds to the subset relation.

2.2.1 Nominal Subtyping and Substitution

- Subtype objects have a *wider interface* than supertype objects.
    - Existence of methods and fields: Subtypes may add, but not remove methods and fields.
    - Accessibility of methods and fields: An overriding method must not be less accessible than methods it overrides.
    - Types of methods and fields.
        - *Contravariant parameters*: An overriding method must not require a more specific parameter type.
        - *Covariant results*: An overriding method must not have a more general result type than the methods it overrides (out-parameter and exceptions are results as well).
        - *Non-variance of fields*: Subtypes must not change the types of fields.
        - Types of immutable fields can be specialized in subtypes (works only in the absence of inheritance, as the supertype constructor will initialize the field).
    - Eiffel allows narrowing interfaces in three ways (removing methods, overriding with covariant parameter types and specializing field types), all possibly resulting in a run-time exception called “catcall detected”.

2.2.2 Covariant Arrays

- Java and C# have covariant arrays, that is if `S <: T`, then `S[] <: T[]`.
- Each array update requires a run-time type check.
- The designers resolved the trade-off between expressiveness and static safety in favor of expressiveness. Generics allow a solution that is both expressive and statically safe.

2.2.3 Shortcomings of Nominal Subtyping

1. Nominal subtyping can impede reuse: If two library classes have the same interface, but no (useful) common supertype, a workaround has to be used.
    - Adapter pattern: An interface is created, and an adapter keeps a private reference to its adaptee, to which all calls are forwarded.
        - Requires boilerplate code and causes memory and run-time overhead.
    - Generalization
        - Instead of top-down development, some languages support bottom-up development with generalization: A supertype can be declared after the subtype has been implemented. However, this approach does not match well with inheritance, and is not part of any mainstream programming language.
2. Nominal subtyping can limit generality as many method signatures are overly restrictive. For instance consider a method `printAll` that expects a collection. Such a method uses only two methods of the collection interface, but requires the type to have all 13 methods.
    - It is possible to make type requirements weaker by declaring interfaces for useful supertypes, but many useful subsets of operations usually exist.
    - Java documentation makes some methods optional: Their implementation is allowed to throw an unchecked exception. This approach clearly loses static safety.

2.2.4 Structural Subtyping and Substitution

- Structural subtypes have by definition a wider interface than their supertypes.
- Generality with structural subtyping: the example with `printAll`
    - Static type checking: Additional supertypes approach applies as well, but the supertypes must only be declared, the subtype relation is implicit (this helps only very little).
    - Dynamic type checking: Arguments to operations are not restricted, similar to optional methods approach (possible run-time errors).

2.2.5 Type Systems in OO-Languages

	Static	**Dynamic**
Nominal	Sweetspot: Maximum static safety	Why should one declare all the type information, but then not statically check it?
Structural	Overhead of declaring many types is inconvenient; Problems with semantics of subtypes (seen later)	Sweetspot: Maximum flexibility

2.3 Behavioral Subtyping

In the definition of type (see section 2.1), *properties* of values are used to characterize them. So far, these properties have been syntactic properties, but the behavior of objects should also be included. The behavior is expressed as interface specifications, i.e. contracts.

- Preconditions have to hold in the state before the method body is executed.
- Postconditions have to hold in the state after the method body has terminated.

- Old-expressions can be used to refer to pre-state values from the postcondition.
- Object invariants describe consistency criteria for object and have to hold in all states, in which an object can be accessed by other objects. That is, invariants have to hold in pre- and post-states of method executions, but may be violated temporarily between. The pre- and post-states are also called *visible states*.
- History constraints describe how objects evolve over time and relate visible states. Constraints must be reflexive and transitive.

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

2.3.1 Rules for Subtyping

- Subtype objects must *fulfill contracts* of supertypes, but:
  - Subtypes can have stronger invariants
  - Subtype can have stronger history constraints
  - Overriding methods of subtypes can have weaker preconditions and stronger postconditions than the corresponding supertype methods
- This concept is called **behavioral subtyping** and is often implemented via specification inheritance.
- Static checking of behavioral subtyping
  - For each override `S.m` of `T.m` check for all parameters, heaps, and results that
    $$Pre_{T.m} \Rightarrow Pre_{S.m} \text{ and } Post_{S.m} \Rightarrow Post_{T.m}$$
  - For each subtype `S <: T` check for all heaps that
    $$Inv_S \Rightarrow Inv_T \text{ and } Cons_S \Rightarrow Cons_T$$
  - Problem: entailment is undecidable
- Dynamic checking of behavioral subtyping
  - Checking entailment for all parameters, heaps, and results is not possible at run-time, but we can check those properties subsequent code relies on. For each method call `o.m(..)` we check
    - that the precondition of `m` in `o`'s dynamic type (which the executed body relies on) is fulfilled.
    - that the postcondition of `m` in `o`'s static type (which the caller relies on) is fulfilled.

2.3.2 Specification inheritance

- Behavioral subtyping can be enforced by inheriting specification from supertypes.

- The invariant of type `S` is the conjunction of the invariant declared in `S` and the invariants declared in the supertypes of `S`. Analogous for history constraints.

2.3.2.1 Simple Inheritance of Method Contracts

- The *precondition* of an overriding method is the *disjunction* of the precondition declared for the method and the preconditions declared for the methods it overrides.
- The *postcondition* of an overriding method is the *conjunction* of the postcondition declared for the method and the postconditions declared for the methods it overrides.
- Problem: The rule for postconditions becomes too restrictive. Consider a method `remove` of a class `Set` with precondition `contains(x)`, and a postcondition that says the size of the set decreased by one. If one would want to write a subclass with `true` as precondition, the method is not able to fulfill the postcondition.

2.3.2.2 Improved Rule for Postcondition Inheritance

- A method must satisfy its postcondition only if the caller satisfies the precondition. That is, every postcondition is implicitly interpreted as $\texttt{old}(\texttt{Pre}_{\texttt{T.m}}) \texttt{ => } \texttt{Post}_{\texttt{T.m}}$
- The postcondition of a method is the conjunction of these implications for the declared and inherited contracts.

2.3.3 Types as Contracts

- Types can be seen as a special form of contracts, where static checking is decidable. Consider an operator `type(x)` that yields the dynamic type of x.
  - For a field `p` of type `P` we have an invariant `type(p) <: P`
  - For a method with parameter `a` of type `A` we have a precondition `type(a) <: A`
  - For a method with return value `r` of type `R` we have a postcondition `type(r) <: R`
- Subtyping now gives us covariance for fields and method results (stronger invariant and postconditions) and contra-variance for method arguments (weaker preconditions).

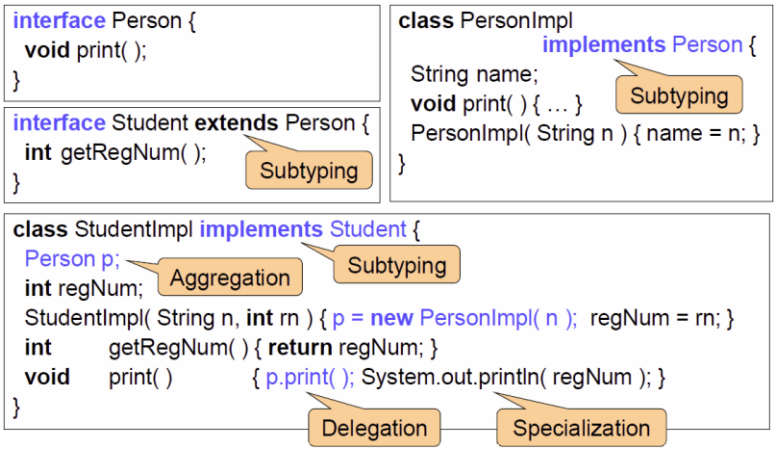
2.3.4 Immutable Types

- Objects of immutable types do not change their state after construction, which has several advantages:
  - No unexpected modifications of shared objects
  - No thread synchronization necessary
  - No inconsistent states
- Subtype relation of mutable and immutable types
  - Immutable types as subtype: Not possible because mutable type has wider interface.
  - Mutable types as subtype: The mutable type does not specialize the behavior.
  - The clean solution is to have *no* subtype relation between mutable and immutable types. The only exception is `Object`, which has no history constraint.

3 Inheritance

3.1 Inheritance and Subtyping

- Subtyping expresses *classification* (substitution principle and subtype polymorphism)
- Inheritance is a means of *code reuse* (through specialization)
- Inheritance is usually coupled with subtyping, and we use the term *subclassing* to refer to the combined mechanism of subtyping and inheritance.
- Simulation of subclassing with delegation
    - Subclassing can be simulated by a combination of subtyping and aggregation (which can useful in languages with single inheritance)
    - Example for workaround with aggregation



    - OO-programming can do without inheritance, but not without subtyping. Inheritance is thus not a core concept, but subtyping is.

3.1.1 Problems with Subclassing

- Consider two classes `Set` and `BoundedSet`, where `BoundedSet` specializes the behavior of the add method to have a precondition `size < capacity`.
    - Syntactically, the two classes could be subtypes of each other, but semantically, not:
        - `BoundedSet` is not a behavioral subtype of `Set`, as the precondition of the `add` method is strengthened.
        - Set is not a behavioral subtype of `BoundedSet`, as `Set` cannot guarantee an invariant that is at least as strong as `size <= capacity` (the invariant of `BoundedSet`).
    - However, large parts of the implementation are identical, and thus this code should be reused.
- Solution 1: Aggregation
    - `BoundedSet` *uses* `Set` and delegates most calls to `Set`.
    - No subtype relation, and thus no polymorphism or behavioral subtyping requirements.

    - Only possible if subtype relation is not needed. Consider classes `Polygon` and `Rectangle`. Clearly, `Rectangle` should be a subtype of `Polygon`, but `Polygon` might have a method `addVertex`, which cannot be supported by `Rectangle`.
- Solution 2: Creating new objects.
    - The `addVertex` method returns its result (of type `Polygon`). Instead of just adding a vertex to its own objects, the `addVertex` method in `Rectangle` can create a new object of type `Pentagon` with the vertex added as needed.
    - For `BoundedSet`, this solution might work (add of `BoundedSet` can return a normal `Set` if capacity is exceeded), but this is most likely not what the user wants or expects. Of what use is a bounded set, if by adding elements it just gets converted to a regular set?
- Solution 3: Weak superclass contract
    - Behavioral subtyping is always relative to a contract. We can introduce a superclass `AbstractSet` with very weak contracts (`true` for both pre- and postcondition), and strengthen the subtype contracts via postconditions.
    - Problems might arise in verification. The method add in `Set` has a postcondition `contains(o)` and is implemented as `super.add(o)`, but how can we show that this postcondition is actually established, if the postcondition of the super method does not guarantee us anything?
    - One solution to this problem are *static* contracts, which specify a given method implementation, and are only used for *statically-bound* calls (e.g. super calls). No behavioral subtyping needed.
- Solution 4: Inheritance without subtyping
    - Some languages support inheritance without subtyping (e.g. C++ with private and protected inheritance, or Eiffel with expanded inheritance)

Aggregation	Private Inheritance
- Both allow code reuse without establishing a subtype relation. No subtype polymorphism and no behavioral subtyping requirements.	
- More overhead: two objects at run-time, boilerplate code for delegation, access methods for protected fields	- Private inheritance might lead to unnecessary multiple inheritance

3.2 Dynamic Method Binding

- **Static binding**: At *compile time*, a method declaration is selected for each call based on the *static* type of the receiver expression.
- **Dynamic binding**: At *run-time*, a method declaration is selected for each call based on the *dynamic* type of the receiver object.
- Dynamic method binding enables specialization and subtype polymorphism. However, there are important drawbacks:
    - Performance: Overhead of method look-up at run-time
    - Versioning: Dynamic binding makes it harder to evolve code without breaking subclasses

3.2.1 Fragile Baseclass Problem

- Software is not static (maintenance, bug fixing, reengineering)
- Subclasses can be affected by changes to superclasses
- How should we apply inheritance to make our code robust against revisions of superclasses?
- Example: Selective override
    - Consider again a class `Set` with methods `add` and `addAll`, where `addAll` calls add repeatedly. A subclass might implement counting the number of elements by incrementing a field in `add`. If the method `addAll` class `Set` later is changed to directly add the elements instead of calling `add`, our subclass gets broken.
- Rules for proper subclassing
    - The subclass depends on the implementation of `add`, and not only on its contracts/interface documentation.
    - The superclass should not change calls to dynamically-bound methods, as this potentially breaks subclasses. Do not add, remove or change the order of such calls.
    - The subclass should also override all methods that could break invariants.
    - The subclass should avoid specializing classes that are expected to be changed (often).

3.2.2 Binary methods

- Binary methods take a receiver and one explicit argument (e.g. `equals`). Often, the behavior should be specialized depending on the *dynamic types of both arguments*.
- Example: Consider a class `Shape` with subclasses like `Rectangle` or `Circle`. There might be a method `intersect` that intersects two shapes, where for some special instances (e.g. intersection two rectangles) we would like to use specialized, more efficient code.
- Solution 1: Explicit type tests
    - Type test and conditional for specialization based on dynamic type of explicit argument.
    - Problems: Tedious to write, code is not extensible and requires a type cast.
- Solution 2: Double invocation
    - Additional dynamically-bound call for specialization based on dynamic type of explicit argument.
    - In our example, we would introduce methods like `intersectShape` and `intersectRectangle` that then contain the most efficient code, and `intersect` would simple call the appropriate method on the explicit argument.
    - Note that this is also called *visitor pattern*. `Shape` corresponds to both `Node` and `Visitor`, intersect to `Node.accept` and `intersectX` to `Visitor.visitX`.
    - Problems: Even more tedious to write, and requires a modification of the superclass (which might not always be possible, e.g. for equals).
- Solution 3: Multiple dispatch
    - Some research languages allow method calls to be bound based on the dynamic type of several arguments.

```
class Shape {
  Shape intersect( Shape s ) {
    // general code for all shapes
  }
}
```

```
class Rectangle extends Shape {
  Shape intersect( Shape@Rectangle r ) {
    // efficient code for two rectangles
  }
}
```

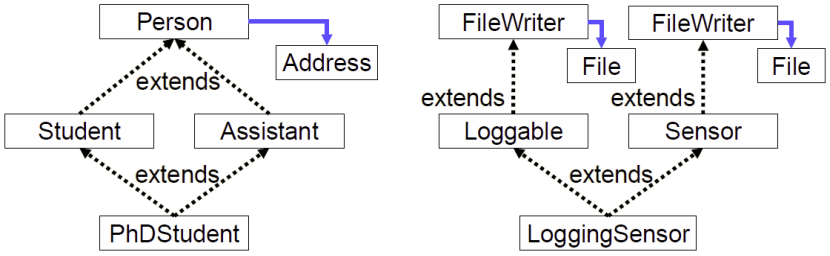
Static type of r

Dispatch on r

- Problems: Performance overhead of method look-up at run-time, and there are extra requirements needed to ensure that there is a *unique best method* for every call.

3.3 Multiple Inheritance

- Problems of multiple inheritance
  - Ambiguities: Superclasses may contain fields and methods with identical names and signatures.
  - Repeated inheritance (diamonds): A class may inherit from a superclass more than once. How many copies of the superclass members are there? How are the fields initialized?
- Ambiguity resolution: merging methods
  - *Related* inherited methods can often be merged into one overriding method, in which case the usual rules for overriding apply (type rules and behavioral subtyping).
  - *Unrelated* methods cannot be merged in a meaningful way, even if the signatures match. The subclass should provide both methods, but with different names.
- Ambiguity resolution: explicit selection
  - The ambiguity can be resolved by the client if he explicitly selects one or another method.
- Ambiguity resolution: renaming
  - Inherited methods can be renamed, and dynamic binding then takes renaming into account.
- How many copies of superclass fields?

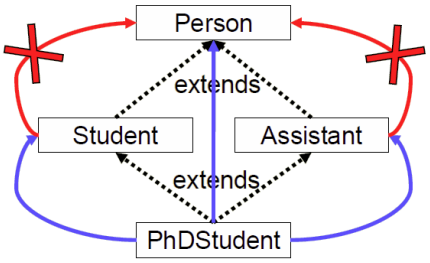


  - One copy (Eiffel with default, C++ with virtual inheritance)
  - Multiple copies (Eiffel with renaming, C++ with non-virtual inheritance)

3.4 Inheritance and Object Initialization

- Superclass fields are initialized before subclass fields. This helps preventing the use of uninitialized fields, e.g. in inherited methods.
- Order is typically implemented via mandatory call of superclass constructor at the beginning of each constructor.
- With *non-virtual inheritance*, there are two copies of the superclass fields, and the superclass constructor is thus called twice to initialize both copies.

- With *virtual inheritance* there is only one copy of the superclass fields. Who gets to call the superclass constructor?
  - C++ Solution
    - Constructor of repeated superclass is called only once by the *smallest subclass*, which need to call the constructor of the virtual superclass directly.



    - Example:

```
class Person {
  Address* address;
  int workdays;
public:
  Person( Address* a, int w ) {
    address = a;
    workdays = w;
  };
};
```

```
class Student : virtual public Person {
public:
  Student( Address* a ) : Person( a, 5 ) { };
};
```

```
class Assistant: virtual public Person {
public:
  Assistant( Address* a ) : Person( a, 6 ) { };
};
```

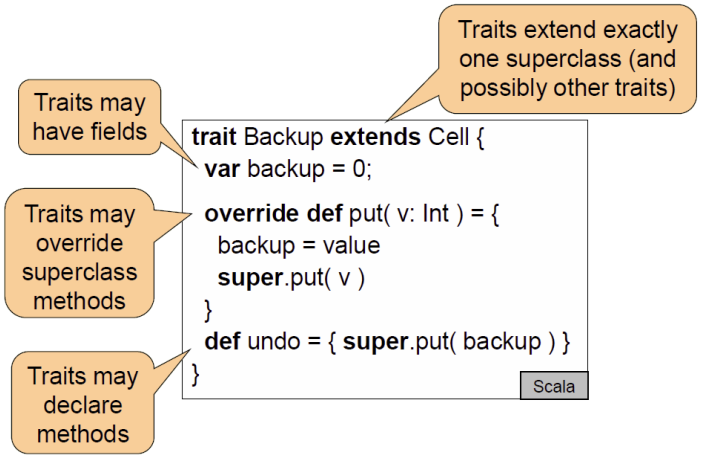
```
class PhDStudent : public Student, public Assistant {
public:
  PhDStudent( Address* a ) : Person( a, 7 ), Student( a ), Assistant( a ) { };
};
```

    - Non-virtual inheritance is default, thus programmer need foresight.
    - Constructors cannot rely on the virtual superclass constructors they call (in the above example, the constructor of `Student` cannot assume that `workdays` equals `5` at the beginning of the constructor, since if it is called in the context of a `PhDStudent`, the Person constructor will be called with a value of 7.
  - Eiffel solution
    - Eiffel does not force constructors to call superclass constructors, and the programmer has full control.
    - No call of superclass constructor: Subclasses have to initialize inherited fields (code duplication and subclasses need to understand superclass implementation)
    - Always call superclass constructor: Constructors of repeated superclasses get called twice (what if they have different arguments?). This is also problematic if the constructor has side-effect.

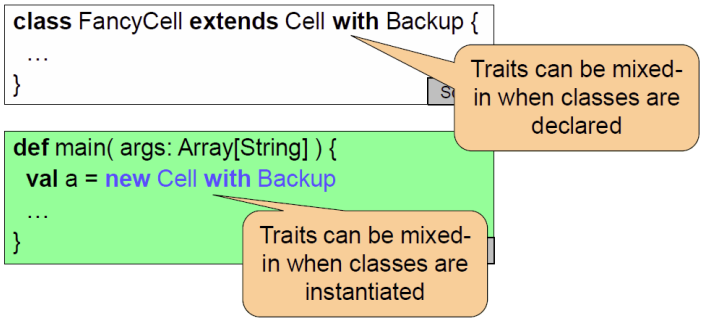
Pros Multiple Inheritance	Cons Multiple Inheritance
- Increases expressiveness - Avoids overhead of using the delegation pattern	- Ambiguity resolution (explicit selection, merging, or renaming) - Repeated inheritance (complex semantics, initialization, renaming) - Complicated!

3.5 Traits

- Mixins and traits provide a form of reuse
  - Methods and state that can be mixed into various classes
  - Example: Functionality to persist an object
- Main applications are making thin interfaces thick, and stackable specializations
- Language that support mixins or traits: Python, Ruby, Scala
- Declaring traits in Scala



- Mixing-in traits



- Class must be a subclass of its traits' superclasses (otherwise we would get multiple inheritance).
- Each trait defines an *abstract type*, similar to interfaces.
- Ambiguities are resolved by *merging*. In particular, there is no scope-operator like in C++ or renaming as in Eiffel.
  - Subclasses override all mixed-in methods with the same method. However, this does not work for mutable fields, in which case the field can be overridden by a method. It is possible to access the fields of supertypes using **`super`**`[Type].fieldName`.

3.5.1 Linearization

- Linearization is the key concept to understanding the semantics of Scala traits. It brings the supertypes of a type in a *linear order*.
- For a definition `C` **`extends`** C_1 **`with`** C_2 … **`with`** C_n the linearization `L(C)` is defined as

$$L(C) = C, L(C_n) + \ldots + L(C_1)$$

    - + is concatenation, where elements of the right operand replace identical elements of the left operand. That is, only the last occurrence of any type in the linearization is included.
- Overriding and super-calls are defined according to this linear order.
- Subclasses only inherit one copy of repeated superclasses (just like Eiffel and virtual inheritance in C++). The initialization order of classes and traits is the *reverse linear order*.
    - Each constructor is called exactly once.
    - Constructors of superclasses of traits must not take arguments.
        - Fields must be initialized in subclasses.
        - Support through abstract constants.
        - Programmers need foresight.

3.5.2 Reasoning about traits

- Stackable specializations: with traits, specializations can be combined in flexible ways.
- Behavioral subtyping with traits
    - Overriding of trait methods depends on order of mixing
    - Behavioral subtyping can only be checked when traits are mixed in.
- Reasoning: traits are very dynamic, which complicates static reasoning.
    - Traits do not know how their superclasses get initialized.
    - Traits do not know which methods they override.
    - Traits do not know where super-calls are bound to.

3.5.3 Summary

- Traits party solve problems of multiple inheritance: Linearization resolves some issues ambiguities and initialization
- Other problems remain
    - Resolving ambiguities between unrelated methods
    - Initializing superclasses
- And new problems arise
    - No specification inheritance between trait methods
    - What to assume about superclass initialization and super-calls

4 Types

4.1 Bytecode Verification

- Mobile code as a motivation
  - Download and execution of code, e.g. as Java applets
  - Upload of code to customize a server
  - Automatic distribution of code and patches in distributed systems
- Class loaders
  - Programs are compiled into platform-independent bytecode that is organized into class files.
  - The bytecode is interpreted on a virtual machine, and the class loader gets code for classes and interfaces on demand.
  - Programs can also contain their own class loaders.
- Mobile code cannot be trusted
  - Code may not be type safe
  - Code may destroy or modify data
  - Code may expose personal information
  - Code may purposefully degrade performance (denial of service)
- How can we guarantee a minimum level of security with an untrusted code producer and untrusted compiler?

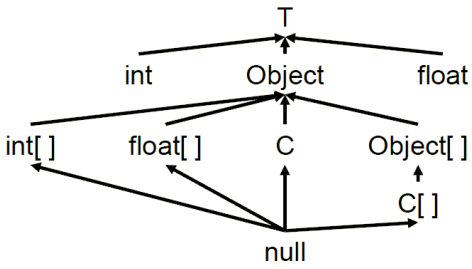
4.1.1 Java Virtual Machine and Java Bytecode

- Java Virtual Machine
  - JVM is stack-based, and most operations pop their operands from a stack and push a result.
  - Registers store method parameters and local variables.
  - Stack and registers are part of the method activation record.
- Java bytecode
  - Instructions are typed.
  - Load and store instructions access registers.
  - Control is handled by intra-method branches (goto, conditional branches).
- A proper execution requires that
  - Each instruction is type correct
  - Only initialized variables are read
  - No stack over- or underflow occurs
  - Etc.
- JVM guarantees these properties by *bytecode verification* when a class is loaded and *dynamic checks* at run-time.

4.1.2 Bytecode Verification via Type Inference

- The bytecode verifier simulates the execution of the program, where operations are performed on types instead of 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* (type mismatch, or stack over- or underflow).
- Types of the inference engine
  - Primitive types, object and array reference types
  - `null` type for the null reference
  - `T` for uninitialized registers



- Rules
  - The maximum stack size `MS` and the maximum number of parameters and local variables `ML` are stored in the class file.
  - Rule for method invocation uses method signature, no jump.

iconst n:
$(S,R) \rightarrow (int.S,R)$, if $|S| < MS$

astore n:
$(t.S,R) \rightarrow (S,R\{ n \leftarrow t \})$,
if $0 \leq n \leq ML \wedge t <: Object$

iload n:
$(S,R) \rightarrow (int.S,R)$,
if $0 \leq n \leq ML \wedge R(n) = int \wedge |S| < MS$

invokevirtual C.m.σ:
$(t'_n \ldots t'_1.t'.S,R) \rightarrow (r.S,R)$, if
$\sigma = r (t_1,\ldots,t_n) \wedge t' <: C \wedge t'_i <: t_i$

  - Branches lead to joins in the control flow, and if an instruction has several predecessors, the smallest common supertype is selected (or `T` if no other common supertype exists).
    - With multiple subtyping, several smallest common supertypes may exist.
    - The JVM solution is to ignore interfaces and treat all interface types as `Object`.
    - `invokeinterface I.m` cannot check whether the target object implements interface `I`, and thus a run-time check is necessary.
- The inference algorithm is a fixpoint iteration

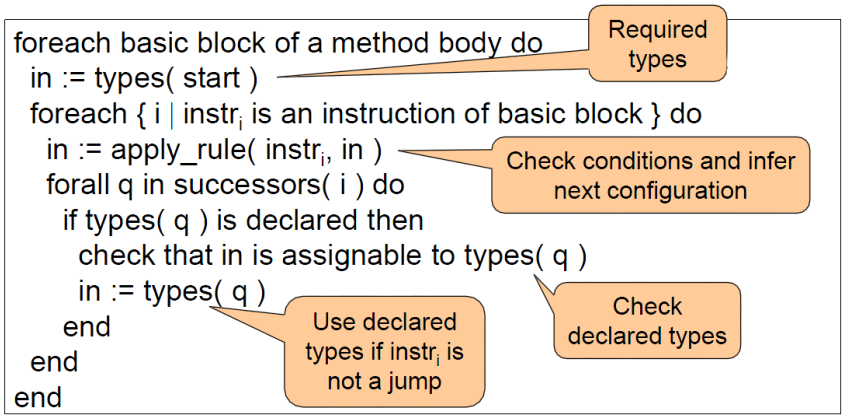
```
in( 0 ) := ( [ ] , [ P0,…,Pn,T,…,T ] )
worklist := { i | instr_i is an instruction of the method }
while worklist ≠ ∅ do
  i := min( worklist )
  remove i from worklist
  out( i ) := apply_rule( instr_i, in( i ) )
  forall q in successors( i ) do
    in( q ) := pointwise_scs( in( q ),out( i ) )
    if in( q ) has changed then worklist := worklist ∪ { q }
  end
end
```

  - `pointwise_scs` computes the smallest common supertype of two vectors, and is undefined for stacks of different sizes.
- Advantages of type inference
  - Determines the most general solution that satisfies the typing rules.
  - Might be more general than what is permitted by the compiler.
  - Very little type information required in class files.
- Disadvantages

    - Fixpoint computation might be slow.
    - Solution for interfaces is imprecise and requires run-time checks.
- The alternative is bytecode verification via type checking which exists since Java 6.

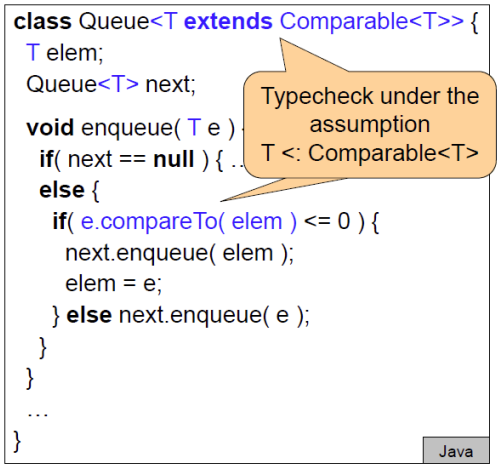
4.1.3 Bytecode Verification via Type Checking

- Class files are extended to store type information, e.g. `([int], [C,int,T])`.
- Type information can be declared for each bytecode instruction, and is required at the beginning of all basic blocks (i.e. at jump targets, and at every entry point of an exception handler). This includes all join points.
- Computation of smallest common supertype no longer needed, which avoids the fixpoint computation and the interface problem.
- Type checking algorithm



4.2 Parametric Polymorphism

- Not all polymorphic code is best expressed using subtype polymorphism. For instance consider writing a class `Queue` to store objects of a certain type.
- Classes and methods can be parameterized with type parameters.
    - Clients provide instantiations for these type parameters.
    - Modularity: The generic code is type-checked once and for all (without knowing the instantiations).
- Type-checking the generic code often requires information about its type argument (e.g. availability of methods). This can be expressed using *upper bounds*. Example:



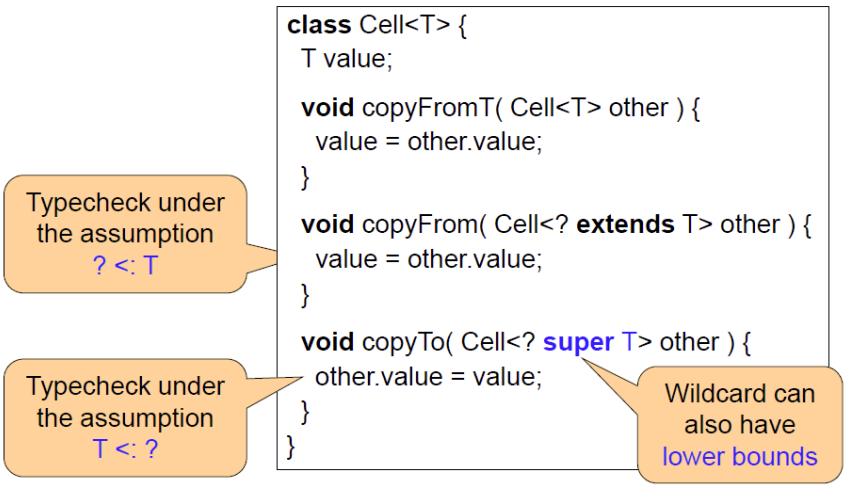
- Subtyping and generics: Generic types are only statically type-safe, if they are non-variant.
  - Covariance is unsafe when the generic type argument is used for variables that are written by clients (i.e. mutable fields and method arguments).
  - Contravariance is unsafe when the generic type argument is used for variables that are read by clients (i.e. fields and method return values).
  - Java/C# solution: Generic types are non-variant.
  - Eiffel solution: Generic types are covariant, which is in line with covariant method parameters and fields, but is not statically type-safe.
  - Scala solution: By default, generic types are non-variant, but the programmer can supply variance annotations. In this case, the type checker imposes restrictions on the positions the type parameters can be used.
    - A covariance annotation (+) is useful if type parameter only occurs in positive positions (result types and types of immutable fields).
    - A contravariant annotation (−) is useful if type parameter only occurs in negative positions (parameter types).
- Working with non-variant generics: There are two ways to write code that works with different instantiations of generic classes.
  - Method type parameters
  - Wildcards

4.2.1 Wildcards

- A wildcard represents an unknown type, and can be interpreted as *existential type*. "There is a type argument `T` such that `c` has type `Collection<T>`". The existential quantifier is automatically instantiated by the type system.

```
static void printAll( Collection<?> c ) {
  for ( Object e : c ) { System.out.println( e ); }
}
```
Java

- Two different wildcards introduce two existential types, which need not to be the same.
- Wildcards can be constraint by lower and upper bounds.

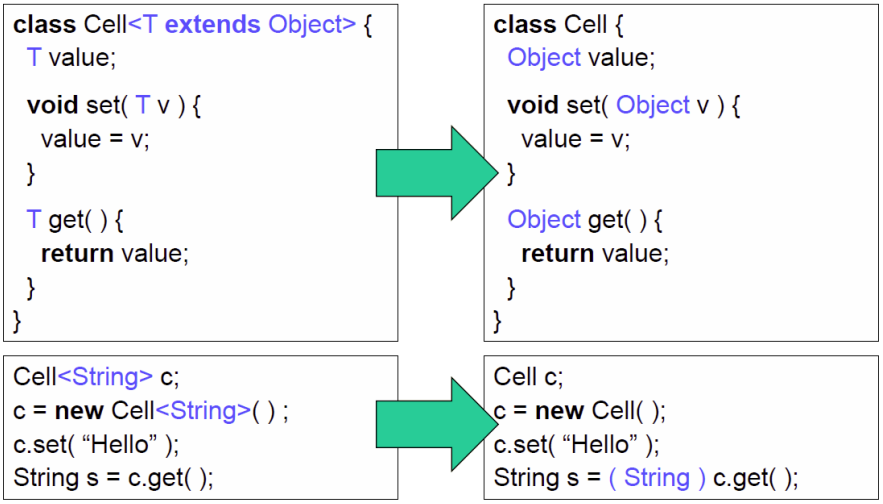


4.2.2 Method Type Parameters

- Often, wildcards can be replaced by additional method or class type parameters (C# does not have wildcards).
- Method type parameters
  - Cannot be used for fields
  - Do not allow upper bounds
  - Allows different types parameters to be equal (e.g. `void <T> foo(T a, T b)`), something which is not possible with wildcards.
- Class type parameters
  - Can be used for fields, but `C<?>` allows different instantiations, where a class type parameter instantiation is fixed over the lifetime of an object.

4.2.3 Type Erasure

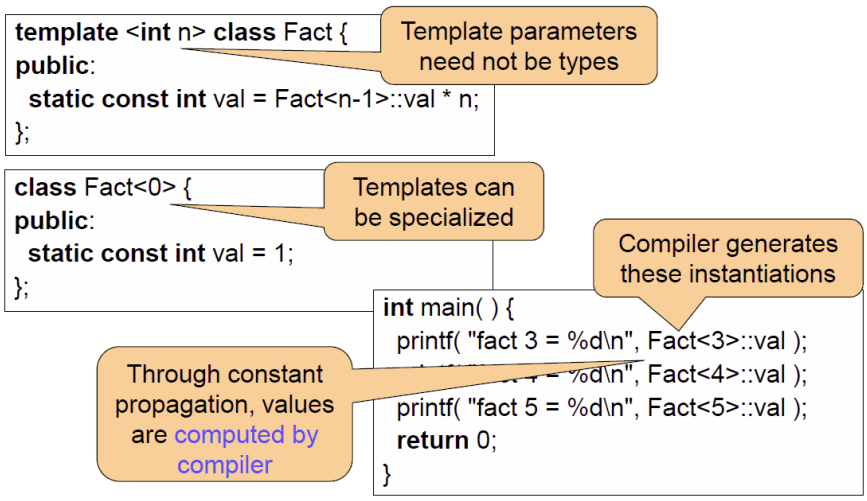
- Java 1.4 introduced generics. For backwards compatibility, Sun did not want to change the virtual machine, and thus generic type information is *erased* by the compiler:
  - `C<T>` is translated to `C`
  - `T` is translated to its upper bound
  - Casts are added where necessary
- Only one class file and only one class object to represent all instantiations of a generic class.
- Example:



- The missing run-time information has various consequences:
  - Generic types are not allowed with **`instanceof`** (as in `c instanceof C<T>`).
  - Class object of generic types is not available (as in `C<T>.class`).
  - Arrays of generic types are not allowed (as in `new C<T>[10]`).
  - Casts to generic types are not checked at runtime (as in `(C<T>)c`)
  - Static fields are shared by *all* instantiations of a generic class.
- As C# does not have type erasure, those limitations do not apply to C#.

4.2.4 C++ templates

- Templates allow classes and methods to be parameterized, where client provide instantiations for template parameters.
- The compiler generates a class for every template instantiation, and the type checking is done for this generated class, not the template. Type errors are only found when the corresponding code is used.
  - There is no need for upper bounds on type parameters, as the availability of methods is not checked anyway (in the template code).
  - Template has to document (informally) what it expects from its type parameters.
- If two files using the same template with the same type instantiation are used, but compiled separately, the executable will contain multiple copies of the same type and corresponding code.
- Templates can be used for meta-programming:



Generic Types	Templates
- Modular type checking of generic classes - Only one copy of code - Run-time support desirable - No meta-programming - Based on sophisticated type theory	- Type checking per instantiation - Code duplication - No need for run-time support, as every instantiation results in its own type - Meta-programming is Turing-complete - “Glorified macros”

5 Information Hiding and Encapsulation

5.1 Information Hiding

- **Information hiding** is a technique for reducing the dependencies between modules:
  - The intended client is provided with all the information to use the module *correctly*, and *with nothing more*.
  - The client uses only the (publicly) available information.
- Objectives
  - Establish strict interfaces
  - Hide implementation details
  - Reduce dependencies between modules
    - Classes can be studies and understood in isolation
    - Classes only interact in simple, well-defined ways
- The client interface of a class contains
  - Class name
  - Type parameters and their bounds
  - Super-class (only for subtyping information)
  - Super-interfaces
  - Signatures of exported methods and fields
  - Client interface of direct superclass
- What about inheritance? Is the name of the superclass part of the client interface or an implementation detail? In Java, inheritance is done by subclassing, and the subtype information must be part of the client interface.
- Various other interfaces exist
  - Subclass interface
    - Efficient access to superclass fields
    - Access to auxiliary superclass methods
  - Friend interface
    - Mutual access to implementations of cooperating classes
    - Hiding auxiliary classes
  - And more
- Expressing information hiding
  - Java: access modifiers
    - **`public`**: client interface
    - **`protected`**: subclass and friend interface
    - default access: friend interface
    - **`private`**: implementation
  - Eiffel: clients clause in the feature declarations
    - **`feature`** `{ANY}`: client interface
    - **`feature`** `{T}`: friend interface for class T

        - **`feature`** `{NONE}`: implementation (only “this”-object)
        - All exports include subclasses
- Safe changes
    - Consistent renaming of hidden elements
    - Modification of hidden implementations as long as the exported functionality is preserved
    - Access modifiers and clients clauses specify what classes might be affected by a change.
- Exchanging implementations
    - The observable behavior must be preserved
    - Exported fields limit modifications severely
        - Use getter and setter instead
        - Uniform access principle in Eiffel
    - Modifications are critical
        - Fragile base class problem
        - Object structures

5.2 Encapsulation

- 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.
- Definition: **Encapsulation** is a technique for structuring the state space of executed programs. Its objective is to guarantee data and structural consistency by establishing capsules with clearly defined interfaces.
    - Encapsulation deals mainly with dynamic aspects.
    - There are different levels of encapsulation. Capsules can be:
        - Individual objects
        - Object structures
        - A class (with all of its objects)
        - All classes of a subtype hierarchy
        - A package (with all of its classes and their objects)
        - Several packages
    - Encapsulation requires a definition of the boundary of a capsule and the interface at the boundary.

5.2.1 Consistency of Objects

- Objects have (external) interfaces and (internal) representation.
- Consistency can include properties of one execution state or relations between execution states.
- The internal representation of an object is encapsulated if it can only be manipulated by using the object’s interface.

- Example 1
    - A class might have an invariant about a public field, but exported fields allow objects to manipulate the state of other objects and thereby breaking such an invariant.
    - Solution: Apply proper information hiding
- Example 2
    - In example 1, one might make the field protected. But now, a subclass might introduce (new or overriding) methods that break the consistency.
    - Solution: Behavioral subtyping

5.2.2 Achieving Consistency of Objects

1. Apply encapsulation: Hide internal representation wherever possible.
2. Make consistency criteria explicit by using contracts and informal documentation (e.g. invariants).
3. Check interfaces: Make sure that all exported operations of an object (including subclass methods) preserve all documented consistency criteria.

5.2.3 Invariants

- Invariants express consistency properties and the textbook solution is that the invariant of an object ○ has to hold in the pre- and poststates of ○'s methods.
- Assuming that all objects ○ are capsules (i.e. only methods executed on ○ can modify ○'s state, and the invariant of an object ○ only refers to the encapsulated fields of ○), we have to show for each invariant:
    - That all exported methods preserve the invariants of *the receiver object*.
    - That all constructors establish the invariants of *the new object*.
- Object consistency in Java
    - Declaring all fields **`private`** does not guarantee encapsulation on the level of individual objects.
    - Objects of the same class can break each other's invariants.
    - Eiffel supports encapsulation on the object level with **`feature`** `{NONE}`.
    - Invariants for Java (simple solution)
        - Assumption: The invariants of object ○ may only refer to private fields of ○.
        - For each invariant we have to show
            - That all exported methods *and constructors of class* `T` preserve the invariants of *all objects of type* `T`.
            - That all constructors *in addition* establish the invariants of the new object.

6 Object Structures and Aliasing

6.1 Object Structures

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

6.2 Aliasing

- Definition (general): An *alias* is a name that has been assumed temporarily.
- Definition (object-oriented programming): An object ○ is *aliased* if two or more variables hold references to ○.
    - Variables can be fields, static fields, local variables (including **`this`**), formal parameters or results of method invocations.
- Definition: An alias is *static* if all involved variables are fields of objects or static fields. An alias is *dynamic*, if it is not static.
    - Static aliasing occurs in the heap, where dynamic aliasing involves stack-allocated variables.

6.2.1 Intended Aliasing

- Efficiency
    - In object-oriented programming, data structures are usually not copied when passed or modified.
    - Aliasing and destructive updates make object-oriented programming efficient.
- Sharing
    - Aliasing is a direct consequence of object identity.
    - Objects have state that can be modified.
    - Objects have to be shared to make modifications of state effective.

6.2.2 Unintended Aliasing

- Capturing
    - Capturing occurs when objects are passed to a data structure and then stored by the data structure.
    - Capturing often occurs in constructors (e.g. streams in Java).
    - Problem: Alias can be used to *bypass the interface* of the data structure.
- Leaking
    - Leaking occurs when data structures pass a reference to an object, which is supposed to be internal to the outside.
    - Leaking often happens by mistake.
    - Problem: Alias can be used to *bypass the interface* of the data structure.

6.3 Problems of Aliasing

- Observation: Many well-established techniques of object-oriented programming work for individual objects, but not for object structures in the presence of aliasing.

    - Information hiding and exchanging implementations (e.g. implementing `addElems` of `ArrayList` via capturing or coping. The observable behavior changes, even though contracts don't).
    - Encapsulation and consistency (invariants can be violated via aliases).
- Other problems
    - Synchronization in concurrent programs: 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.

6.4 Encapsulation of Object Structures

- We need better control over objects in an object structure to avoid the problems with aliasing.
- Approach:
    1. Define *roles* of objects in object structures.
    2. Assign a tag (*alias mode*) to every expression to indicate the role of the referenced object.
    3. Impose *programming rules* to guarantee that objects are only used according to their alias mode.
- Roles in object structures
    - *Interface objects* (*peer* mode) are used to access the structure. They can be used in any way objects are usually used (passed around, changed, etc.).
    - *Internal representation* (*rep* mode) of the object structure. Objects referenced by rep-expressions can be *changed* and must *not be exported* from the object structure.
    - *Arguments* (*arg* mode) of the object structure. Objects must *not be changed* through arg-references, but can be passed around and aliased freely.
- Meaning of alias modes
    - Alias modes describe the role of an object relative to an interface object. Informally, references
        - with peer mode stay in the same context.
        - with rep-mode go from an interface object into its context.
        - with arg-mode may go to any context.

6.4.1 Simplified Programming Discipline

1. No role confusion.
    - Expressions with one alias mode must not be assigned to variables with another mode.
2. No representation exposure.
    - rep-mode must not occur in an object's interface.
    - Methods must not take or return rep-objects.
    - Fields with rep-mode may only be accessed no **`this`**.
3. No argument dependence.
    - Implementations must not depend on the state of argument objects.

- No representation exposure
    - rep-objects can only be *referenced* by their interface objects or by other rep-objects of the same object structure.
    - rep-objects can only be *modified* by methods executed on their interface objects, or by methods execute on rep-objects on the same object structure.
    - rep-objects are *encapsulated* inside the object structure.
- Invariants for Object Structures
    - The invariant of object o may depend on encapsulated fields of ○ and fields of objets that ○ references through rep-references.
    - Interface objects have *full control* over their rep-objects.
- No argument dependence
    - Objects references through arg-references may be freely aliased.
    - Objects structures have *no control* over the state of their argument objects.
    - Invariants must not depend on fields of argument objects, but can depend only on their identity.

```
private /* arg */ T v, w;
// invariant v != w        -- legal
// invariant v.f != w.f    -- illegal
```

- Alias control in modular programs
    - Rules for rep-mode can in general *not be checked modularly*.
    - Subclasses can add new methods or override methods.
    - In Java, rep exposure can be prevented by access modifiers, **`final`** and inner classes. However, this restricts subclassing severely.

7 Ownership Types

7.1 Readonly Types

- Alias prevention (as seen in the last section) is not always wanted, as aliases are helpful to *share side-effect*, and cloning objects is not very efficient.
    - E.g. one often wants to share an address object of a person.
    - In many cases, it suffices to *restrict access* to shared objects, and most commonly: grant read access only.
- Requirements for readonly access
    - Mutable objects
        - Some clients can mutate the object, but others cannot.
        - Access restrictions apply to references, not whole objects.
    - Prevent field updates
    - Prevent calls of mutating methods
    - Transitivity: Access restrictions extend to references to sub-objects.

7.1.1 Readonly Access via Supertypes

- We can introduce readonly interfaces to ensure readonly access. For instance:

```
interface ReadonlyAddress {
  public String getStreet( );
  public String getCity( );
}
```

```
class Address
   implements ReadonlyAddress {
... /* as before */ }
```

```
class Person {
 private Address addr;
 public ReadonlyAddress
    getAddr( )
  { return addr; }
 public void setAddr( Address a )
  { addr = a.clone( ); }
... }
```

- Clients use only the methods in the interface
    - Object remains mutable
    - No field updates
    - No mutating methods in the interface
- Limitations
    - Reused classes might not implement a readonly interface (similar the discussion of structural subtyping).
    - Interfaces do not support arrays, fields and non-public methods.
    - Transitivity has to be encoded explicitly, and sub-objects are required to implement their own readonly interface.
    - This solution is not safe
        - No checks are made that methods in a readonly interface actually are side-effect free.
        - Readwrite aliases can occur, e.g. through capturing.
        - Clients can use casts to get full access.

7.1.2 Readonly access in Eiffel

- Better support for fields

- Readonly supertype can contain getters
- Field updates only on **`this`** object.

- Command-query separation
  - Distinction between mutating and inspector methods
  - But queries are not checked to be side-effect free
- Other problems as before
  - Reused classes, transitivity, arrays, aliasing, downcasts

7.1.3 Readonly access in C++ via `const` pointers

- C++ supports readonly pointers
  - No field updates and no mutator calls.

```
class Address {
  string city;
public:
  string getCity( void )
    { return city; }
  void setCity( string s )
    { city = s; }
};
```
C++

```
class Person {
  Address* addr;
public:
  const Address* getAddr( )
    { return addr; }
  void setAddr( Address a )
    { /* clone */ }
};
```
C++

```
void m( Person* p ) {
  const Address* a = p->getAddr( );
  a->setCity( "Hagen" );
  cout << a->getCity( );
}
```
Call of const function allowed

Compile-time error

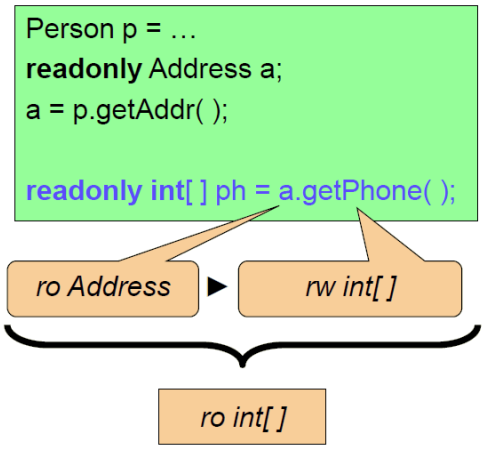
- **`const`** functions must not modify their receiver object.
- Limitations
  - However, **`const`**-ness can be cast away; there are no run-time checks.
  - **`const`** pointers are not transitive, and **`const`**-ness of sub-objects has to be indicated explicitly.

C++ solution: Pros	Cons
- **`const`** pointers provide readonly pointers to mutable objects (prevent fields updates and calls of non-**`const`** functions) - Works for library classes - Support for arrays, fields and non-public methods	- **`const`**-ness is not transitive - **`const`** pointers are unsafe (explicit casts are possible) - readwrite aliases can occur

7.1.4 Readonly Types and Pure Methods

- Pure methods: Tag side-effect free methods as **`pure`**
  - Must not contain field updates
  - Must not invoke non-**`pure`** methods
  - Must not create objects
  - Can only be overridden by **`pure`** methods
- Types
  - Each class or interface introduces *two types*
    - Readwrite type `rw T`, denoted by `T` in programs
    - Readonly type `ro T`, denoted by **`readonly`** `T`
- Subtyping
  - Subtyping among only readwrite or only readonly types is defined as in Java:

        - If S extends or implements T, then `rw S <: rw T` and `ro S <: ro T`
    - Readwrite types are subtypes of the corresponding readonly types:
        - `rw T <: ro T`
    - Subtyping is defined by the transitive closure of these rules.



- Type rules: transitive readonly
    - Accessing a value of a readonly type or through a readonly type should yield a readonly value.
    - We use a type combinatory to determine the type of field or array accesses and method invocations.

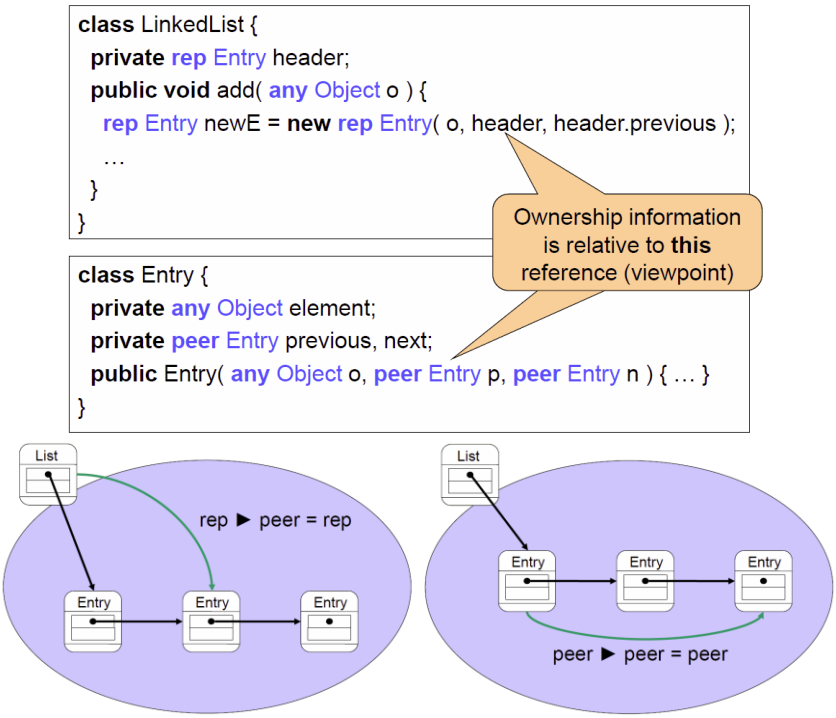
▶	`rw T`	`ro T`
`rw T`	`rw T`	`ro T`
`ro T`	`ro T`	`ro T`

- Expressions of readonly types must not occur as receiver of
    - a field update,
    - an array update, or
    - an invocation of a non-pure method.
- Readonly types must not be cast to readwrite types.
- Discussion
    - Readonly types enable safe sharing of objects
    - Very similar to **`const`** pointers in C++, but
        - transitive, and
        - no casts to readwrite types.
    - All rules for pure methods and readonly types can be checked statically by a compiler.
    - Readwrite aliases can still occur, e.g. by capturing.

7.2 Topological types

- 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 structured into a forest of ownership trees
- Ownership types: 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
- Type safety
    - Run-time information consists of the class of each objects, and the owner of each object.

    - Type invariant: 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 is arbitrary (this can be seen as an existential type).
- The **`lost`** modifier (again, an existential type)
    - Some ownership relations cannot be expressed in the type system.
    - Internal modifier **`lost`** for fixed, but unknown owner is used.
    - Reading locations with **`lost`** ownership is allowed.
    - Updating locations with **`lost`** ownership is unsafe.
- The **`self`** modifier
    - Internal modifier is only used for the **`this`** literal to allow modification of fields on **`this`**.
- Subtyping
    - For types with identical ownership modifier, subtyping is defined as in Java. For `S <: T` we have
        - **`rep`** `S <:` **`rep`** `T`
        - **`peer`** `S <:` **`peer`** `T`
        - **`any`** `S <:` **`any`** `T`
    - **`rep`** types and **`peer`** types are subtypes of corresponding any types
        - **`rep`** `T <:` **`any`** `T`
        - **`peer`** `T <:` **`any`** `T`
    - More rules for **`lost`** modifier
        - **`rep`** `T <:` **`lost`** `T`
        - **`peer`** `T <:` **`lost`** `T`
        - **`lost`** `T <:` **`any`** `T`
    - Rules for **`self`** modifier
        - **`self`** `T <:` **`peer`** `T`
- Viewpoint adaption
    - Again, we have a type combinatory
    - A field read `v=e.f` is correctly typed, if `τ(e) ► τ(f) <: τ(v)`.
    - A field write `e.f=v` is correctly typed if `τ(v) <: τ(e) ► τ(f)`. Furthermore, `τ(e) ► τ(f)`, cannot be **`lost`**.
    - Analogous rules for method invocations, where
        - argument passing is analogous to field writes, and
        - result passing is analogous to field reads.
    - Examples

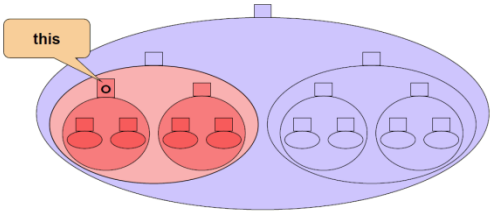


- Type combinatory

▶	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

7.2.1 Owner-as-Modifier Discipline

- Topological type systems can be used to strengthen encapsulation
  - Prevent modifications of internal objects.
  - Treat **`any`** and **`lost`** as readonly types.
  - Treat **`self`**, **`peer`** and **`rep`** as readwrite types.
- Additional rules enforce owner-as-modifier
  - Field write `e.f=v` is valid only if `τ(e)` is **`self`**, **`peer`** or **`rep`**.
  - Method call `e.m(..)` is only valid if `τ(e)` is **`self`**, **`peer`** or **`rep`**, or `m` is a **`pure`** method.

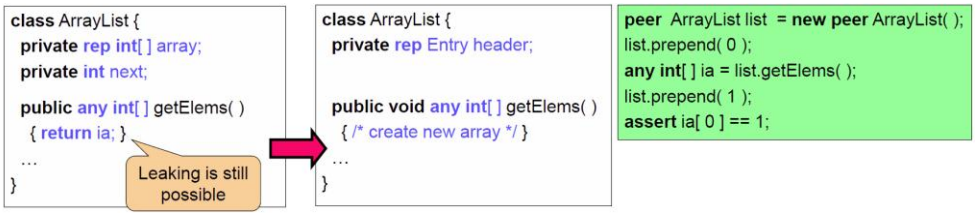


- A method may only modify objects directly or indirectly owned by the owner of the current **`this`** object.

7.2.2 Consequences

- **`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 **`final`** or private inner classes.
- Accidental capturing is prevented, as **`any`** cannot be assigned to **`rep`**.
- However, leaking is still possible. Example (observable behavior changes):



- Consistency of object structures
  - Consistency of object structures depends on fields of several objects.
  - Invariants are usually specified as part of the contract of those objects that represent the interface of the object structure.
  - The invariant of object ○ may depend on
    - encapsulated fields of ○, and
    - fields of objects ○ references through **`rep`** references.
  - Interface objects have *full control* over their **`rep`** objects.
- Summary
  - Ownership types express heap topologies and enforce encapsulation.
  - Owner-as-modifier discipline is helpful to control side effects:
    - Maintain object invariants
    - Prevent unwanted modifications
  - Other applications also need restrictions of read access:
    - Exchange of implementations
    - Thread synchronization

8 Initialization

8.1 Simple Non-Null Types

- Non-null type `T!` consists of references to objects of type `T`.
- Possibly-null type `T?` consists of references to objects of type `T`, or **`null`**.
- 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`**.
- Goal: prevent null-dereferencing statically
  - Require non-null types for the receiver of each field access, array access and method call.
  - Analogous to preventing "message not understood" errors with classical type systems.
- Subtyping and casting
  - The values of type `T!` are a proper subset of `T?`. Therefore, for `S <: T`
    - `S! <: T!`
    - `S? <: T?`
    - `T! <: T?`
  - Downcasts from possibly-null types to non-null types require run-time checks.
- Type rules: expressions whose value gets dereferenced at run-time must have non-null type.
  - Receiver of field access
  - Receiver of array access
  - Receiver of method call
  - Expression of a **`throw`** statement

8.2 Object Initialization

8.2.1 Constructors Establish Type Invariant

- The naïve type invariant requires non-null fields to have non-null values at all times. However, at the beginning of the constructor this invariant cannot possibly hold.
  - Idea: make sure all non-null fields are initialized when the constructor terminates, and weaken type invariant accordingly.
  - Apply definite assignment analysis for fields in constructor (Eiffel's solution for attached types)
- Problem 1: Method calls
  - Method calls are dynamically bound, and (if we want to be modular) we therefore cannot know that they do not access non-initialized fields.
- Problem 2: Callbacks
  - Even if we only call methods on other objects, it may still be the case that the other method calls us back, and our method then possibly accesses non-initialized fields.
- Problem 3: Escaping via method calls

- If we pass out a reference to the **`this`** object in a constructor, other code might directly (callback) or indirectly call a method on our object (or even just read a field), which potentially access non-initialized fields.

- Problem 4: Escaping via field updates
  - Instead of passing out our reference to another method as argument, we could also pass out a reference to **`this`** via a field update on another object.
- Summary of definite assignment of fields
  - Sound and modular checking of definite assignment for fields requires that a partly-initialized object does not escape from its constructor
    - Not passed as receiver or argument to a method call.
    - Not stored in a field or an array.

8.2.2 Construction Types

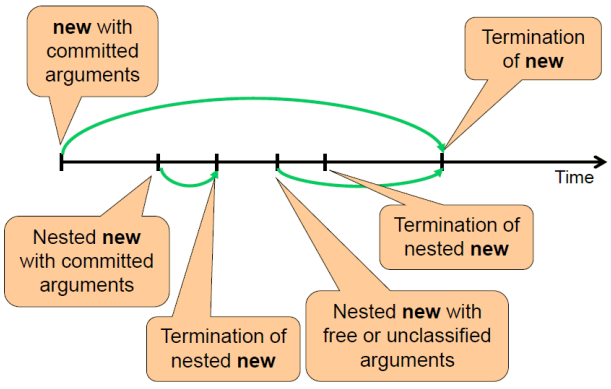
- Idea: design a type system that tracks which objects are under construction. For every class or interface we introduce different types for references to objects under construction, and to such objects whose construction is completed.
- For every class or interface `T`, we introduce six types
  - `T!` and `T?` (committed types)
  - **`free`** `T!` and **`free`** T? (free types)
  - **`unc`** `T!` and **`unc`** `T?` (unclassified types)
- Subtyping
  - `T!` and **`free`** `T!` are subtypes of **`unc`** `T!`
  - `T?` and **`free`** `T?` are subtypes of **`unc`** `T?`
  - No casts from unclassified to free or committed types
- Type rules: field read
  - A field read expression `e.f` is well-typed if
    - `e` is well-typed
    - `e`'s type is a non-null type
  - The type of `e.f` is given by the following table:

Declared type of `f`

Type of `e`	`T!`	`T?`
`committed S!`	`committed T!`	`committed T?`
`free S!`	`unc T?`	`unc T?`
`unc S!`	`unc T?`	`unc T?`

- Type rules: field write
  - A field write `a.f=b` is well-typed if
    - `a` and `b` are well-typed
    - `a`'s type is a non-null type
    - `b`'s class and non-null type conforms to the type of `a.f`
    - `a`'s type is free, or `b`'s type is committed

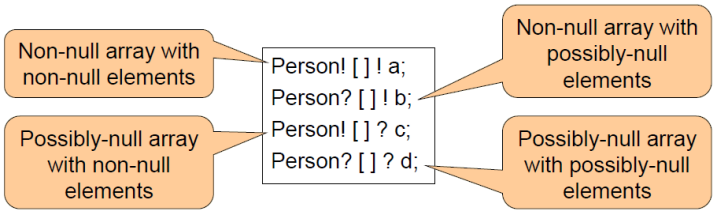
    - The last requirement prevents storing non-committed references in fields of committed receivers.
- Type rules: constructor
    - Constructor signatures include construction types for all parameters
        - Receiver has free, non-null type
    - Constructor bodies must assign non-null values to all non-null fields of the receiver.
        - Definite assignment check
- Type rules: methods and calls
    - Method signatures include constructor types for all parameters
        - Receiver has committed, non-null type, unless declared as free or unclassified.
    - Overriding requires the usual co- and contravariance
    - Calls are typed according to usual rules
- Object construction
    - Is construction finished when the constructor terminates? Not if there are subclass constructors that have not yet executed, which cannot be known modularly in general.
    - Is construction finished when the **`new`**-expression terminates? Not if constructor initializes fields with free references.
- Local and transitive initialization
    - Local initialization: all non-null fields must have non-null values.
    - Transitive initialization: all reachable objects must be locally initialized.
    - Committed references have to satisfy both local and transitive initialization.
- Object creation scenario
    - **`new`**-expression takes committed arguments only, but nested **`new`**-expressions can take arbitrary arguments.
    - After the **`new`**-expression terminates, all new objects are locally initialized, and new objects can only reference committed objects (constructor arguments) or each other, thus all new objects can be typed as committed.
    - Illustrative example:



- Type rules: new expressions
    - An expression **`new`** `C(ei)` is well-typed if
        - All `ei` are well-typed
        - Class `C` contains a constructor with suitable parameter types

- The type of **`new`** `C(ei)` is
  - **`committed`** `C!` if the static type of all `ei` is committed, and
  - **`free`** `C!` otherwise.

- Arrays
  - An array describes two types of references: The reference to the array object, and the references to the array elements. Bot can be non-null or possibly-null.



  - Problems of array initialization
    - Arrays do not have constructors, thus it is unclear at what point the array needs to contain non-null values.
    - Arrays are typically initialized in loops, but static analyses (such as definite assignment check) usually ignore loop conditions. Such an analysis is in general not possible.

- (Partial) solution to array initialization
  - Array initializers, as in `String![]! s = {"a", "b", "c"};`
  - Pre-filled arrays (Eiffel solution). However, it is unclear why a default object is any better than **`null`**.
  - Run-time checks (Spec# solution). In Spec# it is possible to assert that the array is now fully initialized: `NonNullType.AssertInitialized(s);`

- Summary of initialization
  - Object initialization has to establish invariants.
    - Non-nullness is just an example.
  - General guidelines for writing constructors:
    - Avoid calling dynamically-bound methods on **`this`**.
    - Be careful when new objects escape from the constructor.
    - Be aware of subclass constructors that have not run yet.

8.3 Initialization of Global Data

- Most software systems require and maintain global data
  - Factories
  - Caches
  - Flyweights
  - Singletons
- Main issues
  - How do clients *access* the global data?
  - How is the global data *initialized*?
- Design goals of initialization of global data

- Effectiveness
  - Ensure that global data is initialized before first use.
  - Example: non-nullness
- Clarity
  - Initialization has a clear semantics and facilitates reasoning.
- Laziness
  - Global data is initialized lazily to reduce startup time.

- Solution 1: Global variables and init-methods
  - Global variables store references to global data.
  - Initialization is done by explicit calls to init-methods.
  - Dependencies
    - Init-methods are called directly or indirectly from main method.
    - To ensure effective initialization, main needs to know the internal dependencies of the modules.
  - Summary
    - Effectiveness
      - Initialization order needs to be coded manually, which is error-prone.
    - Clarity
      - Dependency information compromised information hiding.
    - Laziness
      - Needs to be coded manually
- Solution 1a: C++ initializers
  - Global variables can have initializers, which are executed before the main method.
    - No explicit calls needed
    - No support for laziness
  - The order of execution is determined by the order of appearance in the source code.
    - Programmers have to manage dependencies manually.

```
class Factory {
  HashMap* flyweights;
  Flyweight* create( Data* d ) { … }
  …
};
Factory* theFactory = new Factory( );
```
C++

- Solution 2: Static fields and initializers
  - Static fields store references to global data.
  - Static initializers are executed by the system immediately before the class is first used (first creation of instance of that class, call to static method, or access to a static field of that class, and before static initializers of sub-classes).
  - Side effects
    - Static initializers can have arbitrary side-effects, and thus reasoning about programs with static initializers is *non-modular*.

    - Summary
        - Effectiveness
            - Static initializer may be interrupted.
            - Reading un-initialized fields is possible.
        - Clarity
            - Reasoning requires keeping track of which initializers have already run.
            - Side effects through implicit executions of static initializers can be surprising.
        - Laziness
            - Static initializers are not called upfront, but also not as late as possible.
    - Static fields and procedural style
        - Procedural style: make all fields and operations of the global data *static*, i.e. use the class object as global object.
        - Disadvantages
            - No specialization via sub-typing and overriding.
            - No dynamic exchange of data structures.
            - Not object oriented.
- Solution 2a: Scala's singleton objects
    - Scala provides language support for singletons
        - Singleton objects may extend classes or traits, but cannot be further specialized.
    - Not every global object is a singleton.
    - Initialization is defined by a translation to Java, which means that the solution inherits all pros and cons of static initializers.
- Solution 3: Eiffel's **`once`** methods
    - **`once`** methods are executed only once.
    - Result of first execution is cached and returned for subsequent calls.
    - Mutual dependencies lead to recursive calls. Such calls return the current value of **`Result`**, which is typically not meaningful.
    - Arguments are only used for the first execution of the method, arguments to subsequent calls are *ignored*.
    - Summary
        - Effectiveness
            - Mutual dependencies lead to recursive calls.
            - Reading uninitialized fields is possible.
        - Clarity
            - Reasoning requires keeping track of which once methods have run already (use of arguments, side effects).
        - Laziness
            - **`once`** methods are executed only when the result is needed (as late as possible).
- Summary of initialization of global data

- No solution ensures that global data is initialized before it is accessed.
  - How to establish invariants over global data?
  - For instance, all solutions are not suitable to ensure that global non-null variables have non-null values.
- No solution handles mutual dependencies.
  - Maybe the programmer should determine the initialization order with appropriate restrictions.

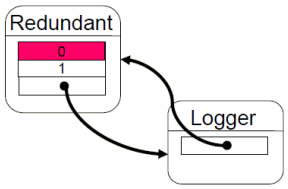
9 Object Invariants

9.1 Call-backs

- Consider the following example

```
class Redundant {
  private int a, b;
  private Logger l;
  // invariant a == b
  public void set( int v ) {
    a = v;
    l.log( "Inside set" );
    b = v;
  }
  public int div( int v ) {
    return v / ( a – b + 1 );
  }
}
```

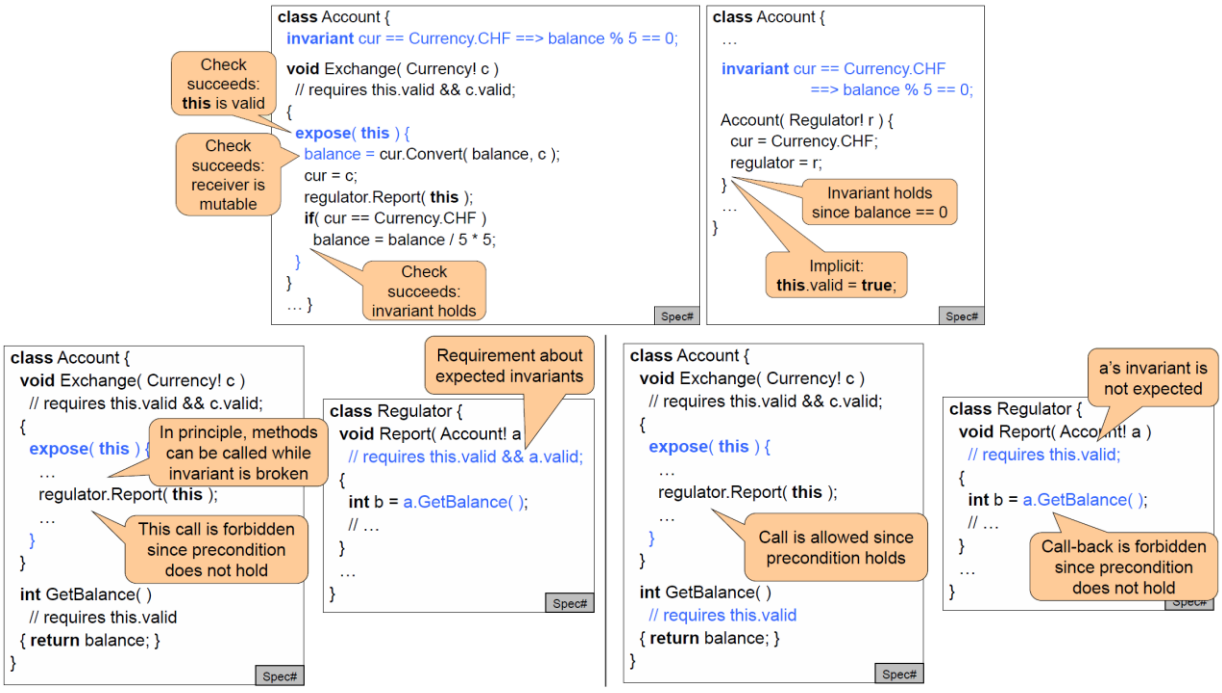
```
class Logger {
  private Redundant r;
  public void log( String m ) {
    System.out.println( m + r.div( 5 ) );
  }
}
```



- Other variants of the example exist
  - Self-calls where the `div` method is called directly from `set`.
  - Re-entrant locks and monitor invariants.
- Solution 1: Re-establishing invariants
  - Check invariant before every method call.
  - However, this is overly restrictive, as most methods do not call back.
  - Too expensive for run-time checking.
- Solution 2: Static analysis
  - Statically analyze the code of the callee to detect callbacks, and check invariant only if call-back is actually possible.
  - Not modular: for dynamically-bound methods, all overrides need to be known.
- Solution 3: Explicit requirements
  - Specify in each precondition which invariants the method actually requires.
  - Check required invariants before every method call.
  - Problems
    - Writing the concrete invariant in the precondition violates information hiding.
    - Some methods require a large number of invariants (e.g. tree traversal).
- Solution 4: Dented invariants
  - Use a boolean field `valid` to indicate whether the object is valid or not. This can be used to turn invariants off an on.
  - Every invariant `inv` is replaced by `valid => inv`.
  - Explicit requirements can be stated using the valid field, e.g. `requires this.valid && arg.valid`.
  - Problems
    - Programmers might forget to set valid field again.
    - Invariant still need to be checked before every method call.
    - A method can break many invariants through direct field updates.

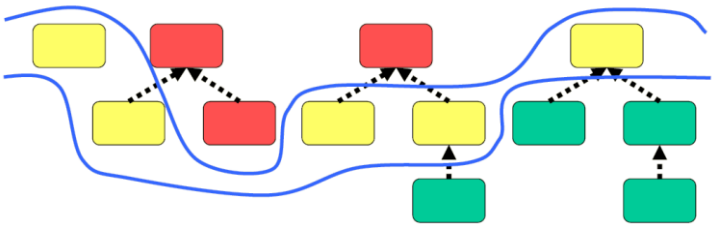
9.1.1 Spec# Solution

- Basic methodology
  - Every object has an implicit `valid` field. Objects can be *valid* or *mutable*.
  - Each invariant is *implicitly dented*.
  - Admissible invariants: The invariant of an object `o` may depend on field of `o` (and constants). (will be relaxed later on)
  - Enforce that invariants hold in *all* execution states, not just visible states.
    - Un-dented invariants hold whenever the object is valid.
  - Valid objects must not be modified. That is, for every field update `o.f=x`, check that `o` is mutable.
- Maintaining object validity
  - Setting the valid field to `true` might break the dented invariant, which is why the valid field can only be modified through a special **`expose`** block.
    - Exposed object must initially be valid.
    - Similar to non-reentrant lock block.
- Establishing object validity
  - New objects are initially mutable, i.e. the valid field is initialized to false.
  - After initialization, the un-dented invariant is checked and the valid field set to true (we ignore inheritance here).
- Proof obligations
  - Invariant of `o` holds after `o` has been initialized.
  - Invariant of `o` holds at the end of each **`expose`**`(o)` block.
  - Every expose operation is done on a valid object.
  - Every field update is done on a mutable receiver.
- Examples



9.2 Invariants of Object Structures

- Invariants often depend on more than the fields of the current object.
  - Example: A person-object might have a field `savings` and an invariant that says the person has a positive amount of money saved: `savings.balance >= 0`.
- Ownership-based invariants
  - Admissible invariants: The invariant of an object ○ may depend on field of ○ and objects (transitively) owned by ○ (and constants).
  - Requirement: When an object ○ is mutable, so are ○'s (transitive) owners, because an update of ○ might break the owners' invariants.
- Enforcing mutable owners
  - The owner is exposed before the owned object, and un-exposing works in the reverse order.
  - Additional checks for **`expose`**`(○)`
    - Before **`expose`**, ○ must be valid and ○'s owner must be mutable.
    - At the end of **`expose`**, all objects owned by ○ must be valid.
- Heap snapshot
  - Red (mutable), yellow (valid, mutable owner) and green (valid, valid owner) objects.

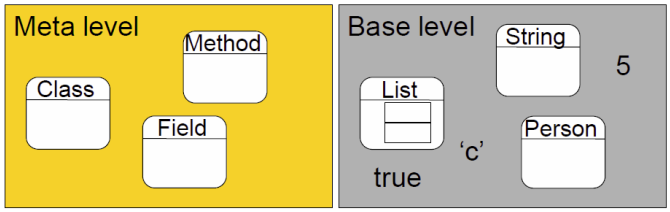


- Proof obligations
  - Owner of newly created object is mutable.
  - Invariant of ○ holds after ○ has been initialized.
  - Invariant of ○ holds at the end of each **`expose`**`(○)` block and all objects owned by ○ are valid.
  - Every expose operation is done on a valid object with a mutable (or no) owner.
  - Every field update is done on a mutable receiver.
- Observations of Spec# methodology
  - Methodology relies on encapsulation of object structure. There is no strict enforcement of owner-as-modifier discipline, but the owner must be exposed before owned object.
  - Responsibility for invariant checking is divided
    - A method *implementation* is responsible for the objects in the context of the receiver.
    - A *caller* is responsible for the objects in its context.
  - Ownership-based invariants are *too restrictive* for many useful examples.
- Invariants and immutability
  - Immutable objects can be freely shared.
  - Invariants may depend on the state of shared immutable objects.

    - Immutability often leads to more reliable programs, especially for concurrency.
- Invariants and monotonicity
    - Many properties of objects evolve monotonically, e.g.
        - Numbers grow or shrink monotonically.
        - References go from null to non-null.
    - Invariants may depend on properties of shared objects guaranteed by their history constraint.
- Invariants and visibility
    - Invariants may depend on fields of shared objects if a modular static analysis can determine all necessary checks.
    - Invariants and fields are declared in the same module (common example: recursive data structures).
- Summary
    - Sound, modular checking of object invariants is surprisingly difficult (call-backs, multi-object invariants, inheritance).

10 Reflection

- In Java it is possible to do dynamic type checking with the **`instanceof`** operator.
- Reflection: A program can observe and modify its own structure and behavior.
  - Simplest form: Type information is available at run-time.
  - Most elaborate form: All compile-time information can be observed and changed at run-time.



10.1 Introspection

- Class objects
  - The class object for a class can be obtained by the predefined class-field, e.g. for class `String` via `String.class`.
- Example: `Field.get(Object obj)`, returns `Object`
  - We can get the value of a field of an object `obj` at run-time.
  - Safety-checks have to be done at run-time:
    - Type checking: does `obj` have that field?
    - Accessibility: is the client allowed to access that field? It is possible to suppress Java's access check by calling `setAccessible(true)` on the field.
- Example `Class.newInstance()`
  - Again, safety checks have to be done at run-time:
    - Type checking
      - Does the class-object represent a concrete class?
      - Does the class have a parameter-less constructor?
    - Accessibility
      - Are the class and the parameter-less constructor accessible?
- Java generics
  - Due to Java's erasure semantics, no generic type information is represented in the class file and at run-time. This can lead to surprising results:

```java
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", String.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

Java

Method not found

```java
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", Object.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

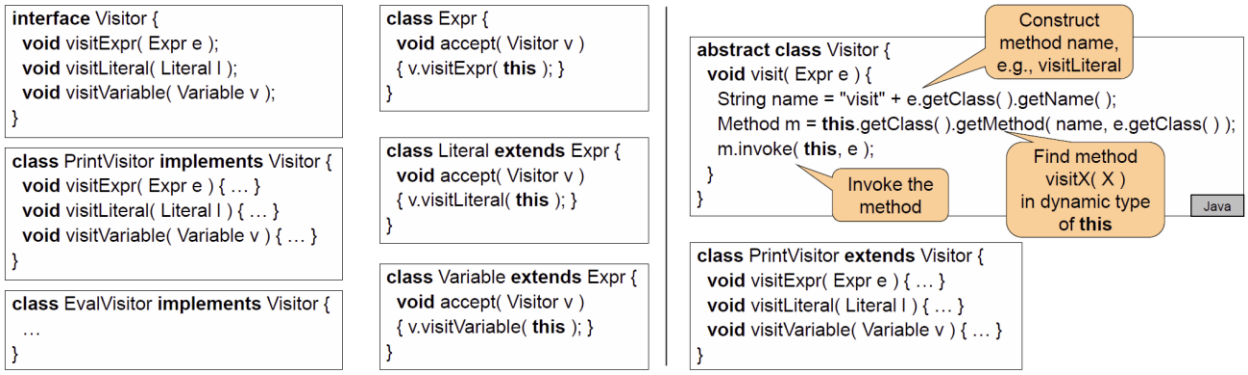
Java

// no exception

10.1.1 Visitor-Pattern via Reflection

- The visitor pattern uses an additional dynamically-bound call for specialization on the dynamic type of the explicit argument.

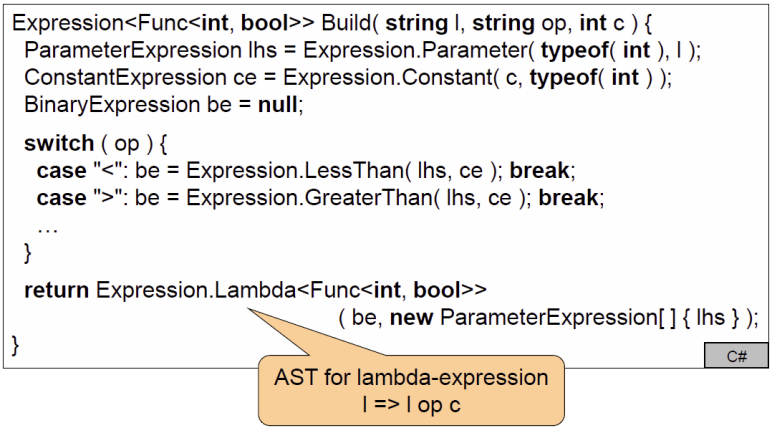
- We can implement a visitor also via reflection and some naming conventions. Example of the standard visitor pattern, compared with an implementation using reflection (error handling omitted):

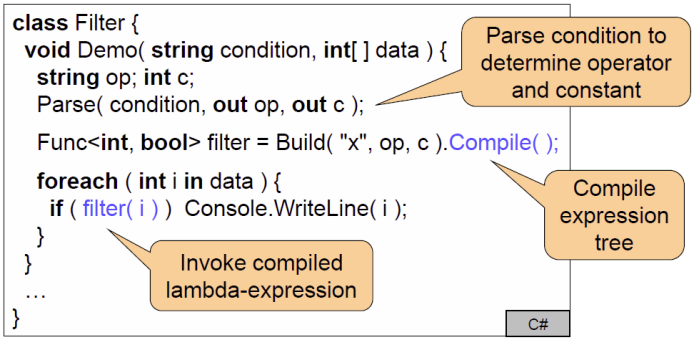


Reflection visitor: Pros	Cons
- Much simpler code (second dynamic dispatch is implemented via reflection, no need for accept-methods - Flexible look-up mechanism (e.g. visit could look for the most specific method)	- Not statically safe (missing method detection must be done at run-time) - Slower: many run-time checks are involved.

10.2 Reflective Code Generation

- If code is represented as data, we can as well allow programs to create code from data.
- Generate code dynamically according to user input and execution environment.
- Examples include class loading in Java, or expression trees in C#
- C# expression trees
  - Expression trees represent the abstract syntax tree of C# expressions and can be created like any other data structure
  - Class expression provides a compile-method, which compiles expression trees to executable code (compilation happens during run-time).
  - Main application is the generation of SQL queries.
- Example:





10.3 Summary

Applications	Drawbacks
- Flexible architectures (plug-ins) - Object serialization - Persistence - Design patterns - Dynamic code generation	- Not statically safe - May compromise information hiding - Code is difficult to understand and debug - Performance

11 Language features

11.1 C++

- Weakly-typed language
- Static method binding is default in C++ (and C#), only using the virtual keyword in front of methods allows dynamic binding.
- Private (and protected) inheritance
    - Inheritance without subtyping
- Virtual inheritance (using keyword virtual in front of superclass)
    - One copy of inherited fields
- Non-virtual inheritance (default)
    - Multiple copies of inherited fields

11.2 Eiffel

- Strongly-typed language
- Dynamic method binding is default
- Expanded inheritance
    - Inheritance without subtyping
- Default inheritance gives one copy per inherited field
- Renaming during inheritance results in multiple copies of inherited fields
- Method arguments, fields and generic types are covariant, even though this is not statically type-safe.

11.3 Java

- Strongly-typed language
- Dynamic method binding is default
- Subtyping for access modifiers:

```
public <: protected <: default <: private
```