2017-2 Concepts of Object-Oriented Programming

46053 characters in 6947 words on 1022 lines

Florian Moser

February 20, 2018

1 Introduction

1.1 requirements

1.1.1 reuse

quality
documented interfaces
extendability and availability

1.1.2 computation as simulation

modeling entities of the real world
describing dynamic system behavior
running simulations

1.1.3 GUI

adaptable standard functionality
concurrency

1.1.4 distributed programming

communication
concurrency
distribution of data and code

1.1.5 core requirements

Cooperating Program Parts with Well-Defined Interfaces
classification and specialization
higly dynamic exection
correctness

1.1.6 from requirements to concepts

cooperating programm parts with well defined interfaces
Objects (data+code), interfaces, encapsulation

classification and specialization
classification and polymophism, substitution principle, leads to classes, inheritance, subtyping, dynamic binding, etc

dynamic adaptation of programm behavior
object model with active objects, message passing

correct programms
interfaces, encapsulation

1.2 core concepts

1.2.1 philosophy

use concepts as close to the real world as possible
easier to use as programmer as he is trained to the real world

1.2.2 object model

software system consists of cooperating objects
objects have state and processing ability
objects exchange messages
objects have state, identity, lifecycle, location, behavior

1.2.3 leads to

different programm structure
different execution model

1.2.4 interfaces

objects have public fields, methods
describe the behaviour of the object

1.2.5 encapsulation

implementation is hidden, information hiding

1.2.6 classification

hierachical structuring of objects
objects belong to different classes simultaniously

1.2.7 substitution principle

subtype objects can be used wherever supertype are expected

1.2.8 specialization

adding specific properties to an object or refining concepts
behaviour of specialised must be compliant to more general type
inherit fields and methods from superclass
override methods in subclass

1.2.9 polymophism

"the quality of being able to assume different forms"
"program part is polymorphic if it can be used for objects of several classes"

subtype polymophism
program parts working with supertype work with subtype

parametric polymophism
generic types

ad-hoc polymorphism
method overloading

1.3 language concepts

enable and facilitate the application of core concepts

dynamic method binding
enables classification and polymorphism
method implementation is selected at runtime

why not just use language concepts as guidelines
inhertiance has been replaced by code duplication
subtyping needed casts, same memory layout of super & subclasses needed

1.4 language design

1.4.1 design goals

simplicity
syntax and semantics can be easely understood by users & implementors of language
simple are BASIC, Pascal, C
conflicting with expressiveness

expressiveness
easely be able to express complex processes and structures
expressive are C#, Scala, Phyton
conflicting with simplicity

(static) safety
language discourages errors and discovers/reports them ideally at compile time
safe are C#, Java, Scala
conflicting with expressiveness, performance

modularity
allows to compile modules separately
modular are C#, Java, Scala
conflicting with expressiveness, performance

performance
programms can be executed efficiently
performant are C, C++, Fortran
conflicting with safety, productivity, simplicity

productivity
language leads to low costs of writing programs
productive are VB, Python
conflicting with static safety

backwards compatability
newer language versions interface with older versions
compatible are Java, C
conflicting with simplicity, performance, expressiveness

2 Types and Subtyping

2.1 type systems

2.1.1 definition

type system is a tractable syntactic method for proving absense of certain program behaviours by classifying phrases according to the kinds of values they compute

2.1.2 properties

syntactic
rules are based on form, not behaviour

phrases
expressions, methods of program

kinds of values
types

2.1.3 weak vs strong

untyped languages
values not classified into types (assembly)

weakly-typed
values classified into types, but additional restrictions not enforced (C, C++)

strongly-typed
all operations must be applied to arguments of appropiate types (c#, eiffel, java, python, scala, smalltalk)

2.2 types

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

2.2.1 subtyping concepts

answer what properties are shared by values of type T
nominal when based on type names (C++, Eiffel, Java, ...)
structural or duck-typing when based on availability of methods and fields (PHP, Python, Ruby, Smalltalk, Go, O'Caml)

2.2.2 type checking

static type checking
every expression has a type
types of variables & methods are declared explicitly or inferred
type rules are use at compile time to check whether program is correctly typed
conservative checks as it needs to approximate run-time behaviour
can bypass checks with dynamic keywork, introduces new run-time checks to preserve type safetly
needs certain run-time checks, examples include type conversions using casts, or array stuff

dynamic type checking
variables, methods, expressions are typically not typed
objects & variables have a type
run-time system checks that operations are applied to expected arguments
support on the fly code generation and class loading

static type checking advantages
guarantees that value held by variable v is subtype of T
static safety finds more errors at compile time
readability because types serve as documentation
efficiency because type information allow for optimization
tool support because types enable auto-complete, support refactoring, ...

dynamic type checker advantages
expressiveness as no correct program is rejected
low overhead because no need to annotate type
simpler compared to static type systems

2.2.3 combinations

static & nominal
sweetspot, maximal static safety, used by C#, C++, Eiffel, Java, Scala

static & structural
inconvenient to declare many types, weird subtypes semantic, used by Go, O'Caml

dynamic & nominal
why declare all type infos but not check it, used by none

dynamic & structural
sweetspot, maximum flexibility, used by JavaScript, Python, Ruby, Smalltalk

2.3 subtyping

2.3.1 substitution principle

objects of subtypes can be used whenever supertypes are expected

2.3.2 syntactic classification

subtypes understand at least the messages of its supertype

2.3.3 semantic classification

supertypes provide at least the behaviour as their supertypes

2.3.4 covariant

more specialized, S <; T implies S[] <; T[]

2.3.5 contravariant

more general, S <; T implies T[] <; S[]

2.3.6 invariant

same

2.3.7 nominal subtyping

determine type membership based on type names
determine subtype relationship based on explicit declarations
wider interfaces (existance & accessibility of methods and fields, types of methods and fields)

rules
sub <; super
sub can add but not remove methods
sub can make methods more accessible
sub must make parameters contravariant (more general, because set)
sub must make result types covariant (more specific, because get)
sub must make fields invariant (same, because get & set)
sub can make final fields covariant (more specific, because get)

problems with reuse
cannot combine similar objects from different namespaces
use adapter pattern (ExployeeAdapter implements Person) but needs boilerplate code
allow generalization (interface Person generalizes Employee) but weird with inheritance, may cause conflicts when changing superclasses

problems with generality
if we only use one specific method in function call we still need to declare whole interface which declares this method
can make separate interfaces for separate functionality but many useful subsets of operations, ReadonlyCollection, WriteonlyCollection, ...
can make methods optional to implement but static safety is lost

2.3.8 structural subtyping

determine type membership based on availability of methods
determine subtype relationship based on availability of methods

reuse
does not have same problems as static

generality
for static checking additional supertypes must be declared, but not the subtype relation
for dynamic checking possible run-time errors due to MethodNotFound

2.4 behavioural subtyping

properties of types should also include behaviours
can be expressed as contracts
concept often implemented via specification inheritance

2.4.1 behaviour definition

all definitions must hold before/after method execution
invariants / history constrains over inherited fields can be violated by all that have access to said field

preconditions
hold in the state before the method is executed

postconditions
hold in the state after the method body has terminated
can use old() expressions
make sure to specify all relevant aspects including data which has not changed

history constrains
how object evolves over time

can use reflexive & transtive old() expression
can strenghten

invariants
describe consistency criteria for objects
can strengthen

2.4.2 contract checking

static checking
programm verification
static safety; more errors are found at compile time
complexity; static contract checking is difficult
large overhead; requires extensive contracts
used by Spec#, .NET

dynamic checking
runtime assertion checking
incomplete, because not all properties can be checked efficiently at runtime (p==p∗p ⇒ (p==0||p==1))
efficient for bug-finding complements testing
low overhead; partial contracts still useful
used by Eiffel, .NET

2.4.3 specification inheritance

simple inheritance
must have identical preconditions than parent, postconditions are conjuncted
problems with multiple subtyping (implement two interfaces, with n>0 && n<0)

improved inheritance
caller may only assume the postcondition which follow from the fulfilled preconditions
need to satisfy each postcondition for which the corresponding precondition (in prestate, old()) holds
effective precondition is the disjunction of all declared preconditions, pre_super ⇒ pre_sub (like parameter)
effective postcondition is the conjunction of all (old(Pre_super) ⇒ Post_super) && (old(Pre_super2) ⇒ Post_super2)
other invariants are all conjuncted

simplify constrains
old(parameter) = parameter as they are immutable
true ⇒ my statement = statement
can use helper methods such as generated() in AbstractClasses if need to reuse implementation for non-behavioural subtypes

2.4.4 structural subtyping

dynamic type checking
callers have no knowledge of contracts, cannot establish precondition, cannot assume postcondition

static type checking
callers could state which signature and behaviour are required (specify reqires P, ensure Q)

2.4.5 type concepts

types as contracts
types can be seen as form of contract, but static checking is decidable
weakper precondition implies contravariance
stronger postcondition implies covariance

immutable types
do not change state after construction
immutable types should be subtype because they specialize behaviour (sematic), but does not work because the interace of mutable is wider (syntax)
mutable types should be subtype because they have wider interface (syntax), but they do not specialize behaviour
clean solution requires no subtype relations between the two concepts
java does it with optional, mutating method (which throws out static safety)

3 Inheritance

3.1 inheritance and subtyping

code reuse can be archived by inheritance (one object, relation fixed at runtime) and aggregation (two objects at runtime)

3.1.1 inheritance

means of code reuse (usually coupled with subtyping)
not a core concept as can be simulated by delegation (delegate calls to IPerson interface to person field in IStudent implementation)

3.1.2 subtyping

expresses classification (substitution principle, subtype polymophism)
use of "extends"/"implements" class
dominates inheritance
implies behavioural subtyping!

3.1.3 subclassing

subtyping + inheritance

3.1.4 aggregation

"has-a", hold object of different type as field

3.1.5 delegation

inside method, call method of other object to take care of things

3.1.6 specialization

override/extend existing method call

3.1.7 example Circle / Ellipse

subtyping makes Circle <; Ellipse (because circle is ellipse)
inheritance makes Ellipse <; Circle (because ellipse has more features)
subclassing chooses Circle <; Ellipse (we must have a is-a relation)

3.1.8 example BoundedSet <; Set

BoundedSet specializes Set.add method (implying behavioural subtype)
BoundedSet strenghtens precondition of Set.add (implying behavioural supertype)
choosing BoundedSet <; Set by returning boolean in add() method
choosing Set <; BoundedSet possible by assigning very high capacity

3.1.9 solutions for not behavioural subtypes but code reuse

aggregation
BoundedSet uses set, delegates method calls
no subtype relation, runtime overhead (two objects), boilerplate code

creating new objects
creating and returning supertype if necessary
problem for clients of subtype because they always need to check

weak superclass contract
create a AbstractSet with weak (public) contracts
but now method is useless (as it does not establish useful postcondition)

weak superclass contract with static contract
have "require false", "static require true", "ensure true", "static ensure contains(o)"
can be used by statically bound calls

inheritance w/o subtyping (private inheritance)
no polymorphism, import code and choose what to make public from superclass
can reuse code, but may leads to unnecessary multiple inheritance

3.2 method binding

3.2.1 static binding

at compile time, method is selected based on static type
default at C#, C++

3.2.2 dynamic binding

selected based on static type of receiver, but dynamic type of argument
drawbacks are performance (method lookup at runtime) and versioning (not breaking everything when code evolves)
default at Eiffel, Java, Scala, dynamically-typed languages

3.2.3 fragile baseclass scenarios

problems
selective overriding (not override all methods which may break invariant)
unjustified assumptions (only can assume explicit postconditions)
mutual recursion (method call themselves)
additional methods (accidentially override methods, accidentially make method more specific in base class), java chooses always most specific, C# chooses most specific in static class

superclass
do not change calls to dynamically-bound methods

subclass
override all methods can could break invariants (only add instead of addAll too)

relay on interface documentation not on implementation (result ˆ2 = something can be + and -!)
avoid extending classes that change often (calls other method, which is overridden to call callee)
check if a new method is overriding existing ones (cleanUp, delete very similar names, can happen by accident)

java vs C#
java has dynamic binding per default, so if a baseclass calls another method this method may be overridden in a subclass, possibly changing behaviour
c# does static method binding per default, so the szenario above does not happen unless specifying virtual

3.2.4 rules summary

use only subclassing for is-a relations (syntactic and behavioral subclassing)
do not rely on implementation, use precise documentation such as contracts
do not mess around with dynamically-bound methods when evolving subclass (do not add/remove/reoder any as it possibly changes behaviour)
do not specialize superclasses that change often (as risk of mistakes increases)

3.3 binary methods

take receiver and one explicit argument
behaviour should be specialized depending on dynamic type of both arguments
how to implement this behaviour specialization?

3.3.1 explicit type tests (instanceof)

how
check for dynamic type of passed argument and cast to more specific type to apply operations

bad
tedious to write, not extensible, required type cast

3.3.2 double invocation

how
in each child class override generic method (like intersect) to call a specialized method (like intersect rectangle), also called visitor pattern

bad
tedious to write, required modification of superclass

3.3.3 overloading plus dynamic

how
c# allows overloading of inherited method; then casting both calle and caller to dynamic forcing the resolution at runtime → most specfic method is called

bad
overhead at runtime, not type safe

example
Shape { Shape intersect(Shape s) } Rectangle { Rectangle intersect(Rectangle r) } DoIntersect(Shape s1, Shape s2) { return (s1 as dynamic).intersect(s2 as dynamic)}

3.3.4 multiple dispatch

how
allow method to be bound based on dynamic type of argument, write Shape intersect(Shape@Ractangle r)
statically type safe!

bad
performance overhead at runtime for method lookup
extra requirements to ensure best method for all calls

3.4 multiple inheritance

often useful to reuse code from several superclasses
good because it increases expressiveness and avoids delegation pattern
but needs ambiguity resolution, may causes repeated inheritance, is complicated

3.4.1 example

Person, Assistent, Student, PhDStudent

3.4.2 simulation in non-multiple inheritance

PhDStudent extends Assistant implements StudentInterface, aggregation + delegation with Student

3.4.3 problems

ambiguities
superclasses may contain fields & method with same name → choose which one in subclass

repeated inhertance (diamonds)
subclass may inherit from superclass multiple times → how many copies of fields, how to initialize fields

3.4.4 ambiguity resolution

explicit selection
phdStudent.Assistant;;workLoad()
client resolves ambuguities → but has to know implementation details!

merging methods
merge related inherited methods
usual rules for overriding apply → behavioural subtyping, type rules

renaming
rename inherited methods if needed (for example not behavioural subtypes)

3.4.5 repeated inheritance resolution

diamond style
superclass fields are initialized before subclass fields → implemented by mandatory supercall in constructor
with virtual inheritance we need to decide who calls the parent constructor

3.5 linearization

mixins and traits
methods and state can be mixed into various classes
making thin interfaces thick
stackable specializations

scala specials
trait Backup extends Cell{}
new Cell with Backup
traits can have fields, (overriding) methods
traits extends one superclass, and 0 or many traits
can be mixed in when declared or when initialized
traits are abstract types

thin and thick interfaces
traits can extend thin interfaces with additional methods
allows very specific type with little syntactic overhead
callme(p: ThinInterface with MyTrait)

ambiguity resolution
ambiguity is resolved by merging
subclass can still call superclass methods with super[Student].workLoad
merging is not required due to linearization

linearization
bring supertypes in linear order
1. linearize superclasses, 2. linearize traits
last trait specified is on top, so assistants workload overrides students'
intialization order in the reverse linear order ("common sense"), arguments to supperclass are provided by preceeding class
supercalls & overriding concerns the next method in linearization order

stackable specializations
with traits (with linearization) specializations can be combined in different orders
with multiple inherited methods of specialized superclasses (diamond methods) are called twice
can reoder mixins so override order changes, change behaviour with complete code reuse

traits summary
very dynamic therefore static reasoning harded
don't know how superclasses are initialized
dont' know which method they override, and which method is called with super.()
order of mixed in traits same for type system; can assign objects with different trait order to each other

linearization summary
fixed some ambiguities, intialization issues fixed
problems with resolving ambiguities between unrelated methods (because override depends on client)
behavioural subtyping cannot be checked modularly (because can mixing traits in different orders),
superclass initialization & call ambiguous

4 Types

4.1 introduction

4.1.1 motivation

mobile devices which only need to implement JVM, bytecode itself runs everywhere

4.1.2 class loaders

load additional classes at runtime, programm can implement their own class loaders

4.1.3 security for java

sandboxing
applets get access to systems resources through API, can implement access control

security preconditions
type safety, code stays inside sandbox

4.2 bytcode verifcation

using java JVM as an example

4.2.1 code may

not be typesafe, modify / destroy data, expose personal information, crash VM, try to DoS
guarantee minimum level of security (untrusted code, untrusted compiler with JVM)

4.2.2 basics

stack based (most operations push/pop stuff from stack)
contains register (to store method parameters & local variables)
size / content fixed on calling method

4.2.3 java bytecode

typed instructions (istore for integer, astore for references)
load / store access registers
control handled by conditional branch, goto

4.2.4 proper execution

instructions must be type correct, only initialized variables can be read, no stack-overflow
guaranteed by bytecode verification and dynamic checks at runtime

4.2.5 bytecode verification BCV

simulates execution of the program, operations are performed on types rather than values
each possible instruction has type rules how stack/registers are modified
if no transition can be found → failure!
example iadd (int.int.S,R) → (int.S, R)

4.2.6 types of inference engine

T (top), then primitive types such as float, int, Object, then the subtypes of objects, null at the bottom
uninitialized is T

4.2.7 branches in BCV

lead to joins in control flow
smallest common supertype SCS is selected
ignores interfaces because this could lead to multiple SCS (checked at runtime, interface types treated as object type)
like static analyzer

4.2.8 BCV with inference (algorithm)

create table; x = # of instruction, y = in(i), out(i); create worklist which contains all instructions numbers #
advance by always choosing lowest entry in worklist, then removing it from it
in(0) is ([], [paramters]) and out(0) is after applying the instruction 0
then in(1) = out(0), and out(1) is in(1) after applying instruction 1

do this for all q
successors(i)

if in(q) has changed add q back to worklist

join out(i) with in(j) by choosing SCS for each stack / register entry

advantages;
determine most general solution
might be more general than compiler
very little type info required in file

disadvantages
fixpoint computation is slow
interface handling is imprecise and needs additional runtime checks

4.2.9 BCV with type checking

since java 6 compiler places more info in .class file to speed up BCV
type info required at beginning of basic blocks (between jump handlers, exception handlers)
so algorithm gets simpler & faster O(n), only has to check if variables are assignable to declared onces by compiler
if not declared do type inference like before

4.2.10 summary

enables secure mobile code
can be done by type inference or type inference
some runtime checks necessary for interfaces

4.2.11 manual bytecode verification

start with the tuple (S, R) where S is stack and R is register, ([], [(this type), ...])
iconst loads to S, istore does S → R, iload does R → S, ifeq jumps if true & removes boolean from stack
merge when loops collide
at each block one may add static information from the compiler to avoid fixpoint computations
interfaces are cast to object with runtime check
bytecode is rejected if stack size is not same for all entry points
type of variable can change in register, stack, but commands must be correctly type (iload vs aload)

4.2.12 type inference

infert type to reduce annotation overhead
determines static type automatically then performs static checking

4.3 parametric polymorphism

parametrize classes with types
can be checked once at compile time without knowing the concrete instantiations (modularity)
specify upper bounds on type contraints
covariant type parameters unsafe because of write
contravariant type parameters unsafe because of read

non-variant generics
statically type safe, but not so expressive

covariant generics
can assign P<A> p = new P<B>() for B more specifc than A

contravariant generics
can assign P<B> p = new P<A>() for B more specifc than A
to allow; need to check return types, and field reads at runtime

additional type parameters
introduce additional type parameters (like PersonComparator example)
bad because exposes implementation details, inconvenient for client
cannot be changed at runtime

wildcards
represents unknown type
interpretation as existential type
subtyping is possible if the set o possible instantiations of one type is stricly inside the set for the other type T2

5 Information Hiding and Encapsulation

5.1 Information hiding

definition
technique for reducing the dependencies between modules
the client is provided with all the information needed to use the module correctly
the client uses only the (publicly) available information

objectives
establish strict interfaces
hide implementation details
reduce dependencies between modules (understand classes in isolation, only simple interactions)

client interface
class name
type parameters with bounds
super-class and super-interfaces

signatures of exported methods and fields
interface of direct super class

subclass interface
access to superclass fields
access to auxiliary superclass methods (protected)

friend interface
mutual access to implementations of cooperating classes
hiding auxiliary classes

safe changes
renaming of hidden elements
modification of hidden functionality as long as exported is preserved
use access modifiers to know which clients are affected
observable behaviour cannot change (don't expose fields, fragile baseclass problem)

5.2 Encapsulation

definition
technique for structuring the state space of executed programs.
guarantee data and structural consistency by establishing capsules with clearly defined interfaces
behaves according to specification in any context it is reused, and disallows reuse

levels of encapsulation
individual objects, object structures, class (with all objects), classes in subtype hierarchy, package
encapsulation requires definition of boundary of capsule and interface of said capsule

consistency of objects
objects have external interface and internal representaion
representation can only be manipulated by using the interface
break consistency with bad information hiding, or breaking behavioural subtyping (overriding fragile methods)

how to archieve consistency
apply information hiding
use contracts or informal documentation to express invariants
check interfaces that all invariants are preserved

check for invariants
for each method check that all exported methods preserve invariants of receiver object (may have to generalize to "for all objects of type T" because private is not "this" private)
for all constructors that they establish the invariants

6 object structures

6.1 definitions

object structures
set of objects that are connected via references

6.2 aliasing

name that has been assumed temporarely (several variables refer to same memory, can lead to unexpected side effects)

6.2.1 object is aliased

if two or more variabes hold reference to object

6.2.2 static aliasing

if all involved variables are fields of objects or static fields (in heap memory)

6.2.3 dynamic aliasing

if not static (includes the stack variables)

6.2.4 intended aliasing

for efficiency
makes OO efficient, data structures not copied

for sharing
to make changes to state effective

6.2.5 unintended aliasing

capturing
when passing object to data structure and it stores it
→ can be used to by-pass interface

leaking
when data structure passes reference of object which is supposed to be internal
→ can be used to by-pass interface

6.3 problems of aliasing

observations
object structures do not provide encapsulation
aliases can be used to bypass interface

consistency of object structures
depends in fields of several objects
making all fields private is not enough (array aliasing for example)

more problems
synchronization in concurrent programs (each individual objects has to be locked)
distributed programming (parameter passing for remote method invocation)
optimizations (cannot inline objects if aliasing is used)

alias control example
LinkedList of java
Entry contains the value, is private inner class, references are not passed out, subclasses cannot leak or manipulate Entries
ListItr allows to iterate, is private inner class, passed out, but no one can manipulate or leak ListItr
subclassing restricted

6.4 readonly types

aliasing useful to share side-effect, but restrict access often useful → make readonly

requirements
mutable objects (some can modify, some cannot)
prevent field updates, calls of mutating methods
transitivity (references to restricted objects are restricted too)

using supertype
"implement" ReadonlyInterface, then pass around object as this ReadonlyInterface
limtied because subclasses can use non-readonly stuff, interfaces do not support arrays & fields & non-public methods
not safe because client can simply cast to not readonly, no checks that methods are side-effect free

pure methods
add keyword pure to side-effect free method
must not contain field update, not invoke non-pure methods, not create new objects, must be overridden by pure methods

types
each class T introduces readwrite T (denoted by T) and readonly T (denoted by readonly T)
rw is default
subtyping as usual, and rw T <; ro T
transtivite readonly; if something is returned by ro T it must be ro too, generally as restrictive as possible
ro types must not be accessor of field updates, array updates, non-pure method calls
ro types cannot be cast to rw types
→ can be checked statically from compiler

6.5 ownership types

6.5.1 roles in object structure

interface objects which are used to access the object structure
internal representation of the object structure which must not be learked
arguments of the object structure which must not be modified

6.5.2 ownership model

object has zero or one owner objects
context is the set of objects which have the same owner
owner relation is ascyclic
heap structured in those ownership trees

6.5.3 types

peer
objects in same context as this object (LinkedListEntry peer nextEntry, objects owner is owner of this)

rep
representation objects in the context owned by this (LinkedList rep header,

objects owner is this)

any
argument objects in any context (LinkedListEntry any content, objects owner is arbitrary)
can not safely write

type safety
runtime info consist of type T and of ownership type

type invariant
static ownership information reflects the run-time owner of said object

subtyping
identical ownership information same subclassing as always, else there is rep T <; any T and peer T <; any T

viewport adapation
denoted with a |> b = c, means a combined with b equal type c
v = e.f (field read, result passing) expressed as T_e |> T_f <; T_v
e.f = v (field write, argument passing) expressed as T_e |> T_f ;> T_v

lost
internal modifier
fixed but unknown ownership bc some ownership information cannot be expressed
more specific than any, less specific than peer, rep
for example when accessing rep properties of peer ojects
peerObject.repProperty → we can't safely change this property
can not assign to lost, as it means it is peer or rep → can not safely assign, therefore already prohibited by type system

self
internal modifier
only valid when using this.myProperty → compiler figues this out
does not modify the viewpoint
more specific than peer

keywords
on all fields of objects (so not structs)
not on constructors, but all other methods which do not return void
on all method arguments which are not structs
when constructing objects, can put rep (to be owner), peer (because we know owner)

6.5.4 viewport adaptation

draw a picture with bubble stuff

resolving types
go from left to right
? combined with any → always gets any
? combined with rep → always gets lost
peer combined with peer → peer
rep combined with peer → rep
any combined with peer → lost
lost combined with peer → lost

6.5.5 owner as modifier principle

based on ownership, define r/w or readonly
any, lost are readonly
self, peer, rep are read/write
can only write to field if target is self, rep, peer
can only call non-pure method if to-be-called object is self, rep, peer
methods may only modify objects directly, indirectly owned by this object

archievements
enables encapsulation of whole object structures
cannot violate encapsulation by casting
fully supports subclassing
accidential capturing prevented
controls side-effects (maintain invariants)

non-archievements
leaking still possible
no restrictions of read access!

6.5.6 consistency of object structures

depends on fields of several objects
invariants specified on objects which represent interface of structure

6.5.7 invariants of object structure

depends on encpasulated and owned fields
interfaces have full control over rep objects

7 Initialization

7.1 simple non-null types

main usages of null
initialization to null
indicate absence of data

non-null types
write type! for non-null
write type? for possible null
goal is to prevent null-dereferencing at compile time
type! <; type?

data-flow analysis
know the possible set of values at any point. Use the resulting graph to know where an assigned value might propagate.
does not track heap locations (does not follow function calls; can set stuff to null in method call)
may not work in concurrent programs

7.2 object initialization

7.2.1 idea

make sure non-null fields are initialized after constructor finishes
apply definitive assignment rule to fields (local variable must be assigned before used)

7.2.2 problems

method calls (dynamically bound methods can cause null exception)
callbacks (implicit call to parent constructor)
escaping via method / field calls (race conditions)

7.2.3 only safe if

partly initialized objects do not escape
not passed as receiver or argument to method call
not stored in a field or array

7.2.4 general guidelines

don't call dynamically bound methods
don't let new object escape constructor
be aware of sub-class constructors

7.2.5 construction types

type system tracks which types are under construction
guarantees that if static type of expression is non-null type, then at runtime its value is non-null
introduce different type of references for obejcts under construction and those which finished construction

types
T! and T? for commited types
free T! and free T? for free types
unc T! and unc T? for unclassified types

subtyping
unc T ;> T
unc T ;> free T
can not cast from T! to T?
can not cast from unc to free or commited

requirements
local initialization (all non-null fields have non-null values)
transitive initialization (if all reachable objects are locally initialized)
cyclic structures must be possible (assign free type of fields in constructor)
if type of expression is commited then value at run time is transitively initialized

field write (e1.f = e2)
e1 type is free or e2 is commited
can not assign free, unc to unc

field read (e.f)
e must be non null
if e is free or unc then read value is of type unc
only if e is commited, read value is also commited

constructors
this is of type free in whole constructor
this is of type commited after termination of constructor
can access methods with keyword free
can declare constructor argument to be unc (to be as permissive as possible)
when passing free references resulting object will be free (for cyclic structures)

when passing commited references resulting objects will be commited

lazy intialization
still works by keeping T? inside class, but returning T!

arrays
syntax is T![]? (possibly null array with non-null elements T)
compiler can not check definite assignment
can solve with default initializers, or run-time checks (NotNullType.Assert(myArray))

7.3 initialization of global data

design goals
effectivness; initialized before access
clarity; clean semantics
lazyness; only initialize what is needed

global vars & init methods
variabes are globally accessible
initialized by explicit call to init method
but main() needs to know structure of application to do this
order needs to be coded manually, compromises information hiding

static fields & initializers
static initializers are executed immediately before sytem uses it
may have arbitrary side effects
reasoning about programs is non-modular
can read unitialized fields in cyclic structures, side effects possible, more or less lazy

once methods
executed only once, result is cached
recursive calls simply use current value of Result as return
only arguments of first call evaluated
very lazy, needs to keep track of already executed methods

8 Reflection

8.1 reflection

a program can observe and modify its own structure and behaviour

introspection
class objects contains methods, class hierchy, fields, ...
field read has run-time checks which does type checking and accessiblity checks
new instance has run-time checks which looks for class, parameterless constructor, and checks accessibility
can do double invocation with flexible lookup, not statically safe, slower, but simpler code

reflective code generation
generate code dynamically, typecheck it at tun-time
can build expression trees in c# for SQL queries

dynamic code manipulation
modify existing code
replace methods at runtime

applications
flexible architectures
object serialization
persistance
design patterns
dynamic code generation

drawbacks
not statically safe
may compromise information hiding
code gets more complex
performance issues

9 Language Specifics

9.1 C

inheritance simulation
use structs; copy fields & methods from base class
need casts for code reuse
all fields / methods must be in the same order (same memory layout)
→ but then can reuse code from base struct

9.2 java

OO approach uses
Inheritance (to avoid code duplication)
Subtyping (to express classification)
Overriding (to specialize methods)
Dynamic binding (to adapt reused algorithms)

OO specialities
allows covariant return types, but not contravariant parameters (this always results in overloading, which is resolved statically)
methods are virtual by default (dynamic binding, contrary to c#)
can declare methods as final so they can't be overridden (static binding)
covariant arrays (more specific) → runtime check on update needed
marks some interface methods as optional, allows to throw notimplemented exception → static safety lost
calls most specific in WHOLE class hierarchy (contrary to C#)
can cast any object to any interface → only at runtime its checked if it holds
interfaces can contain default method implementations (multiple inheritance light, as no issues with field initializations arise)
allows for covariant result types when overridding methods
allows for covariant arrays (S <; T implies S[] <; T[]) therefore each array update requires run-time check

security model
applets get access to system resources only through java API
access control implemented by such API
code is prevented from bypassing API

bytecode principals
each method gets own stack frame (with stack which gets executed, and paramater store), share memory
three times verification (compile, load, runtime)
the type of variable in stack at load verification can change (x=1, x=true would be valid in bytecode verification)
the stack size must be equal if multiple possible sources are (for example when using gotos)
start with ([],[E,T,T,T,..]) at first step; where E is type of class ("this") and T are all local variables

why java runtime checks
bytecode cannot check interface methods (but casts to type object)
casts
co-variant arrays

static vs dynamic stuff
overloading is resolved at compile time statically (like which method signature to call)
overriding is resolved at run time dynamically (like which method with the determined signature should be called)

access modifiers
public; every class can access
protected; only subclasses & classes in same package can access
default; only classes in same package can access
private; only this class can access the element

parametric polymorphism
invariant type parameters
arrays covariant, need runtime checks
java generics are undecidable, and turing complete

get-put principle
extends when only get from structure
super when only write to structure
no wildcard when write and read from structure

wildcards
can define wildcard as type parameter ?
upper bounds with <? extends Person>
lower bounds with <? super Student>
can decide at client side! Cell<? extends B> cell = new Cell<C>() or Cell<? super B> cell = new Cell<A>()
cannot specify lower bounds for type arguments (<S super P> invalid, but <? super P> valid)
instantiation of wildcards can change at runtime
type stored in wildcard can change over time, in contrary as if specified fixed type T
typechecker figures out instantiation of type which satisfies all contraints

generics examples
<S extends Person> int noWildCard(Cell<S>[] type)
int wildCard(Cell<? extends Person>[] type)
<T> addToList(T obj, List<? super T> list)
concat(List<? extends T> from, List<? super T> to)

type erasure
java introduced generics late; did not want to break backwards compatability
generic type infos is erased by the compiler
C<T> replaced by C, T replaced by upper bound (specified by extends clause, or object), casts added where necessary
missing runtime information (instanceof generics fail, class object of generic not available, array of generic types not allowed)
missing runtime checks (can cast Cell<Int> passed into Cell<?> to Cell<String>, will fail at runtime when accessing cell.value)
static fields are shared by all generic class instantiations

information hiding
public (client interface), protected (subclass + friend interface), default (friend interface), private (implementation)

method selection
determine static declaration
check accessibility
determine invocation mode (virtual vs non-virtual)

bugs with method selection
JLS1 has bug where default is overridden by public but not in same package
JLS2 has bug with protected overriden by protected from different package

object consistency
when declaring a field as private other instances of same class can access it

initialization
can have static {} initializers,
executed immediately before use, only once (like scala linerization)

9.3 scala

objects are subtypes from all their parents and their traits
traits must extend an object or another trait (write "trait T extends Person")
you can only mixin traits to classes which are suptype of this supertype → because of linearization

linearlization
linerarize superclasses
linearize traits (from left to right)
determines who is called when calling super class!

generics
variance annotations (+ for only out as return types and immutable fields, - for only in as parameters)
+ is covariant (more specific), - is contravariant
can write P1[T<;A], P2[+T]
can asign P[A] = new P[B] for class P[+T]

example resolving
class B extends A {}
class Q[-X] (can assign to Q[T] when LOWER in the hierarchy)
Q[B] = new Q[A] is valid
Q[A] = new Q[B] is invalid

initialization
provides language support for singletons with the object keyword
uses java static initializers

9.4 C#

OO specialities
methods are statically bound by default (all method calls are resolved at compile time, contrary to java)
cam mark methods as virtual (dynamic binding)
covariant arrays (more specific) → runtime check on update needed
selects most specific method FROM STATIC OBJECT, not base or dynamic (contrary to java)
has anonymous types { prop = 108 }, and var similar to auto keyword in C++
has NO covariant return types
allows for covariant arrays (S <; T implies S[] <; T[]) therefore each array update requires run-time check

correctness
does some static, some dynamic checking

var
type inference
can replace var with object except when using anonymous types

dynamic
is a special kind of type
allows the resolving of methods at runtime (cast variables to dynamic)
can replace dynamic with object except when applying special kind of functions (for example +)

virtual
mark methods with virtual to make them overriable in subclasses

override
mark methods as override if they override a as virtual marked method
this is the "fragile baseclass problem", if the parent calls this method he does not know if he will get his won implementation or the one of a subtype

new
use the new keyword with method declaration to "override" methods not marked as virtual
if the parent now call this method he will get his OWN implementation, not the "overridden" one

generics
invariant type parameters
arrays covariant, need runtime checks

parameteric geenrics
out when only get/read
in when only write

9.5 Eiffel

inheritance
narrowing interfaces permitted (specializing field types, changing existence of methods, override with covariant parameter types)
covariant overriding needs nullable type, at runtime passes null
allows more specific parameters which leads to run-time exception, or setting to null
does not enforce call to superclass constructor

correctness
does dynamic checking

multiple inheritance
class PhDStudent inherit Student rename test as takeExam Assistant
single copy of fields of superclasses per default (can be changed with renaming)
need to explicitly select diamond methods
only one copy of diamond fields by default → multiple if they were renamed
if subclasses superclass contructor this constructor is called multiple times (different arguments, side-effect, ...)

generics
covariant type parameters, need runtime checks

information hiding
client clause at feature
feature { ANY } (client interface), feature { T } (friend interface), feature { NONE } (implementation only, this object only)
all exports to subclasses

readonly
readonly fields can contain only getter, will only be modifyable from this object
command-query seperation (but queries are not actually checked to be side-effect free)

9.6 C++

OO specialities
chooses most specific method in static type; includes base classes in search (like Java does)
can choose specific implementation by calling B;;get
has private inheritance (no subtyping but code import)
default is non-virtual inheritance (so all fields duplicated) because its faster

multiple inheritance
diamond fields are duplicated by default, can use virtual (then only once)
smallest (highest) subclass needs to call constructor of virtual superclass, constructors of diamond edges not performed
"public virtual X" implies that if other classes have "virtual X" in hierachy then X is created only once

templates
allow class & methods to be parametrized
template<class T> class Queue { T elem; }
compiler does not type statically check code with generic argument (like int in new Queue<int>); but breaks at runtime if used methods are not available (structural!)
no need for upper bounds as no checking occurrs
run-time types correspond to generated classes

can do template turing complete meta-programming, compiler which calculates specific values, specialize template with Queue<2> for base case

const
you can mark pointers as const (with const ∗) or variables (const myVal)
all methods to be called from const stuff must also be marked as const, those methods can only modify fields declared as mutable
use const pointers (no field updates, no mutator calls, actually checks for no side effects)
one can cast away const simply by casting to non-const pointer
const not transitive

initialization
can initialize globals with new() statement, order undefined

10 Comments

todo
why not any in last Task
can I assign unc to unc?

gotcha's from exercises
if parameter type is "static" it is generally unsafe because at runtime we don't know the dynamic type of the parameter is subtype of the object (Task 1, Ex3)
C++ marry with himself example works, because B ∗b is different address than C ∗c