1 objects and classes

1.1 object

Is a software machine that allows other elements to access (query) and modify (command) a collection of data. F.e an object can represent a city or a button.

Each object belongs to a class that defines the features or operations one can call on this class. It allows other software to access data on modify data on this object (read/write).
An object is an instance of a class.
Objects only exist during the runtime of a program.

1.1.1 object oriented programming

In object oriented programming you model objects of the real world into physical/abstract or software object.

1.2 class

A class is a sort of category for an object. All objects of a class share the same properties (a class is like a category of things, type).
A program is a model.
A class is the generating class of an object.
Classes only exist in the software text.

1.3 architecture

Architecture is the design of a program into it's classes and the relations between them.

1.4 implementation

Writing the instructions (algorithms) and data structures of the classes.

2 interfaces and information hiding

2.1 clients and suppliers

A client of a software mechanism is any sort of system that uses it. f.e person or other software(electronic system).

The software mechanism is the supplier to the system. f.e. class or group of classes.
Arrow always goes from client to suppliers.

2.2 interface

An interface is the description of the techniques enabling clients to use it. A program interface is an interface where the client is other software. API: Abstract Program Interface.

2.3 API's

Objects are characterised by the operations clients may apply on them.

queries   Operations that return the state or information of an object.

commands   Operations that change the state of an object.

The interface is how we (the client) are going to see the object. Most of the time we do not wan't to see the implementation behind an object and we only wan't to see the interface = Information hiding.

information hiding   The designer of the class must decide which properties are accessible to the clients and which are internal(only for the purposes of the class itself (public and secret)).

3 Commands and queries, uniform access principal

3.1 Features: commands and queries

Every feature call, will always be called on a object (your_object.your_feature). A feature is an operation available on a certain class of objects. Their are three kinds of features:
Commands
Queries
Creation procedures

Queries calling a query should not change the answer the program gives back! The goal of a query is to obtain information about the object without modifying it.

Commands change the properties of one or multiple objects.

3.2 Routines, procedures, functions, attributes, and the Uniform Access Principle

3.2.1 Routine (=method)

Removing details specifics and capturing the essence of the information.
In programming there are two sorts of abstractions:
Data abstraction, done with a class.
Computational abstraction (algorithms) done with routines.
Example of a routine in Eiffel:
r(arg: type)
require
      preconditions...
do
      instructions...
ensure
      postconditions...
end

3.2.2 Uses of routines

There are two different uses for Routines:

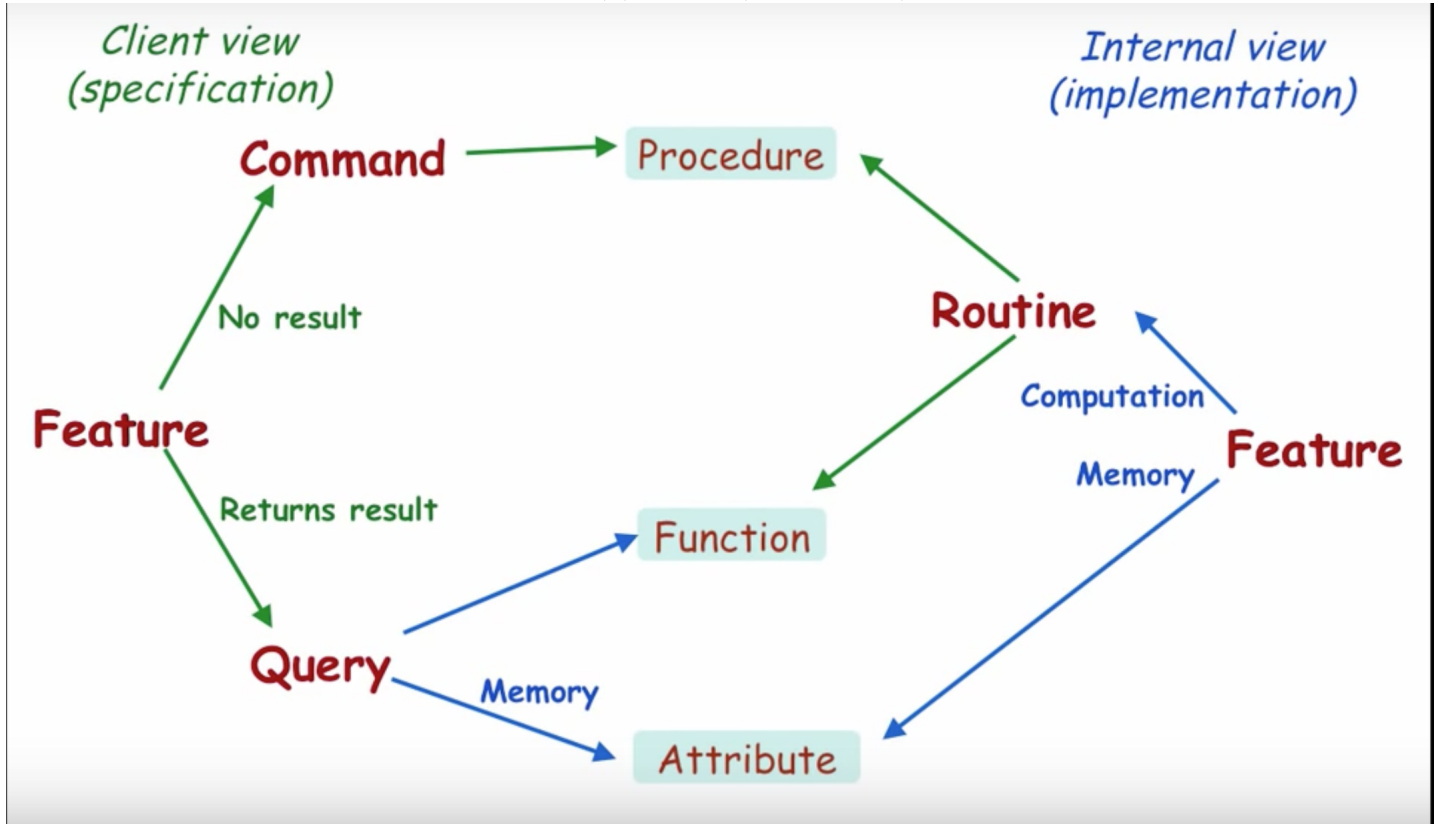**bottom-up:** Encapsulating a computation for reuse.

Top-down: Calling a routine before having written it. Allows to write the bigger code at the beginning and go to the little details later.

3.2.3 Kinds of routines

Their are two kinds of routines:

Procedure: doesn't return a result.

Function: returns a result(f(arg: type):result type).

3.2.4 Attributes

Each class has an attribute field where there is every object of this class in the program.

3.2.5 The uniform access principal

Features should be accessible to the client in the same way wether implemented by storage or by computation(it does not matter if what you are looking for already exists in the memory or if you have to compute it from some other pieces of memory(in the image function and attribute are therefore equal)).

4 Object creation

4.1 basics

4.1.1 Identifier

An identifier is a name chosen to describe certain elements of the program such a class, feature or object.
An identifier that denotes a value at runtime is called an <u>entity</u>.
If the value of an entity can change during runtime it is called a <u>variable</u>.
An entity detonating an object is <u>attached</u> to this object.

4.1.2 Object creation

First you create an object in memory and then attach an entity to it. First you have to declare the entity with a certain type (leg1: LEG). Then you create leg1(create leg1). Leg1 will then be created during runtime and initialised with the default values.
If you want to create an object with other than the default values, you can create <u>creation procedures</u> for the class.

class point create
      default_create, make_cartesian, make_polar
feature
      ...
end

You then have to declare these creation procedures in the feature. Then you can declare the object as(create point.make_polar(r,t)) (create point == create point.default_create).

4.2 Void references

4.2.1 Initial state of an object

Initially, before the creation of an object, it's reference is void. During execution a reference is either <u>void</u> or <u>attached</u>(to find out you can write x=void or x/=void).

4.2.2　Utility of void references

For example in a linked list, to know where the list ends the next next reference of the last element of the list is void.
Void references do cause trouble however (you cannot call a feature on a reference wich is void). Eiffel is void safe, therefore the code will not compile if the code might cause a void call.

4.3　Object creation and the system root

4.3.1　How it all starts

Executing a system is the creation of a root object, this is an instance of the root class, using a special creation procedure of this class called a root procedure. In every program you have to choose a root class and a root procedure(main class in java).

4.3.2　Current object

Is the latest object on which an operation was started. There is always a current object during execution, initially it is the root object(this in java). When you call x.f(a), x becomes the new current object, after it has finished executing the current object before it was x becomes the current object again.

5　References, Assignment, and Object Structure

5.1　Objects, values and references

5.1.1　Object structure

An object is made out of fields, each field has a value which is either a basic value or a reference.

5.1.2　Types

There are also two kinds of types, reference where the entity has a reference to it's type, or expanded where the entity is it's type (an expanded type can not be void since it corresponds to an object). Many objects can have a reference to the same type, but expanded type cannot share.
In order to get expanded objects, the classes have to be declared as expanded:

expanded class E_STATION

All classes such as integer, boolean, character etc.. are all expanded classes.

5.1.3 Initialisation

Their are automatic initialisation rules:
0 for numbers
null for characters
false for booleans
void for reference

5.1.4 Strings

Strings are actually and object in eiffel and java. Therefore the field containing a String in an object will be a reference field.

5.2 Kinds of feature calls and the client relation

5.2.1 Attributes

The fields of a class reflect it's attributes.
To set a field: x:=new_x(= in java (instruction))(= is equal to == in java(expression)). The value of the field is then changed or the reference.

5.2.2 Feature calls

There are two types of feature calls: qualified and unqualified.

Unqualified call   when the feature is being called on the current object (set(new_x, new_y) or Current.set(new_x, new_y)).

Qualified call   the feature is being called to a different object and this object becomes the new current object (position.set(new_x, new_y)).

5.2.3 The client relation

If a class has a field that is a reference to another class and features calls on this field with feature from the other class. The first class is a client of the second.

5.3   Assignment

5.3.1   Entities

An entity is a name in the program that denotes run-time values.
Some entities are <u>constant</u> and others are <u>variables</u>: attributes or local variables.
changing a variable value: target:=source
The target may be an attribute, a result of a function or a local variable. A source a call to a query or an arithmetic expression.

5.3.2   Unreachable objects

When a field in the memory has no reference to it.
There are two approaches to this problem: Manual(c++,Pascal) the programmer has to take of this, Automatic garbage collection (eiffel, java).

5.3.3   Assignments

Two types of assignments: reference(replaces the reference with another one (or void)) and expanded(copies the value).

5.3.4   Local variables

Local variables are entities that only exist inside a routine. Declaration:

```
require
       ...
local
       x: Real
do
       ...
ensure
       ...
end
```

In this case x is a local variable.

6 Logic

6.1 Boolean logic

6.1.1 boolean expressions

In a boolean expression there are: boolean variable (denoting boolean values) and boolean operators (not, or, and, =, implies).
Truth assignment for a set of variables is a choice of true or false for every variable.

6.1.2 Tautologies

A tautology is a boolean expression that has true for every possible truth assignment.

6.1.3 Contradiction

A contradiction is the opposite of Tautology.

6.1.4 Satisfiable

An expression is satisfiable if for one truth assignment it is true.

6.1.5 De Morgan's laws

You can use these laws to simplify boolean expressions.
and and or are associative: a and (b and c) = (a and b) and c. The same for or.

6.1.6 Implies

Implies is only false if a implies b if a is true and b is false.
a implies b = not b implies not a.

6.1.7 semi-strict operators and quantifiers

6.1.8 semi-strict boolean operators

If you divide by x you get an undefined result if x = 0. Therefore you want to check if x = 0 before you do the division and so you need a non-commutative version of and and or: semi-strict boolean operator.
and then = semi-strict version of and.

or else = semi-strict version of or.
semi-strict boolean operators allow you to use an order in boolean expressions.
a implies b = not a or else b (implies is always semi-strict).

6.1.9   Universal quantifiers

Universal quantifiers are there exists and for all.
On a empty set: there exists is always false and for all is always true.

7   Control Structures

7.1   Basics and compounds

7.1.1   Algorithms

An algorithm is a process to be carried out by a computer.
The difference between an algorithm and a program is that an algorithm is more abstract independent of a platform and a programming language. A program normally contains multiple algorithms. Programs are the combination of data structures and algorithms.

7.1.2   Control structures

Definition: program construct that describes the scheduling of basic actions.

There are three fundamental control structures:
      sequence
      conditional
      loop
they are the control structures of structured programming.

Sequence   to achieve c form a, I first achieve b from a, and then c from b.

Conditional   Solve the problem separately on two or more subsets of the input set.

Loop   solve the problem on successive approximations of its input set.

7.1.3 Correctness of a compound

Are done with post and pre condition.
The postcondition an instruction must imply the precondition of the next instruction.

7.2 Conditionals

7.2.1 Structure

if
        condition
then (or if condition then)
        instructions
else (elseif)
        other instructions
end
You can combine conditional instructions in other ones: <u>nesting</u> but it is better to just use elseif.

7.2.2 Multi-branch statement(case)

<u>Structure:</u>

inspect expr(character or integer)
        when value1 then
                instruction1
        when value2,value3 then
                instruction2
        when value3..value5(anything between value4 and value5) then
                instruction3
        else
                other_instruction
end

The inspect table stops as soon it reaches one that is true.
You have to write the else on the end of multi-branch or it will cause an exception if it goes to the end.

7.3 Loops

7.3.1 Structure

from
      initialisation
invariant
      invariant expression
variant
      variant expression
until
      exit condition
loop
      body
ensure
end

The loop will keep on going until the exit condition becomes true. invariant and variant are optional.

7.3.2 Operations on a list

When we have a list there is a cursor which gives us the current position in the list.

Commands   on lists
      start
      back
      forth
      count
      before
      after
start = position 1, before = position 0 and after = position count + 1.
there for in a loop you can write:
from
      line.start
until
      line.after
loop
      line.forth

end

7.3.3 Across loops

An across loop is a more compact way to write a loop that traverses a structure:

```
across structure as c loop
      do something with c.item
end
```

7.3.4 Other loop types

these are not available in eiffel

```
while condition(while this is true (inverse to normal loop)) do
      body
end

repeat
      body
until
      condition
end

for i:a..b (i form a to b) do
      body
end
```

7.4 Loop invariants and variants

7.4.1 Invariants

The invariant helps to check that if the loop terminates it fulfils it's goal and is a <u>boolean expression</u>. The variant helps to check that the loop will terminate. The loop body will always be very tightly tied to the invariant. The invariant can be seen as the set in which the loop operates and the loop can never exit this set. Therefore the body must preserve the invariant.
The conjunction of the invariant and the exit condition gives us the effect of the loop.

7.4.2 Variants

Invariants are used to ensure that loops will terminate and is an <u>integer expression</u>.

The integer expression must be:

 non negative after initialisation

 decrease while remaining non negative for every iteration of the body executed with exit condition not satisfied.

For example in a loop that traverse a list the invariant would be:

 list.count - list.index + 1

The <u>Entscheidungsproblem</u>: the fact that no compiler will be able to tell if a loop will terminate(Alan Turing).

8 Single inheritance

8.1 Single inheritance

8.1.1 Classes

Classes can be viewed in two different ways, as a <u>module</u>(groups a set of services), or as a type. As a module it has a set of features and as a type it has a series of runtime instances.

8.1.2 Inheritance basics

Describing a new class as a extension or specialisation of an existing class(or several with multiple inheritance).

If B inherits from A,:

as modules :<u>all features of A are available in B</u>.

As types: <u>whenever a type A is needed a type B will also be acceptable</u>.

In this case B is a <u>heir</u> of A and A is a <u>parent</u> of B.

The <u>descendants</u> of A are A and the descendants of A's heirs.

8.1.3 Declaring a heir class

class

 class name

inherit

 this class's parent

feature

end

8.1.4 Features

An immediate feature is a feature declared in a class itself or it can be a declared feature.

8.2 Polymorphism

Related to the type view of inheritance(polymorphic = multiple types).

8.2.1 Polymorphic assignement

If you have a reference object initialised as an object of type T where, A and B inherit from T. You can change the reference of T to either A or B(however you cannot call a feature defined in A or B on T even after you have changed it's type because of the compiler).
For example if you have a class transport, a class taxi and a class car. If T is of type transport, you can then change T to be of type car or taxi, according to the choice of the client for example.
The type of source has to be the descendant of the type of the target when target := source.

8.2.2 Static and Dynamic types

The static type of an entity is the type used in it's declaration.
The dynamic type of an entity is the type of the object that this entity is attached to, during runtime.
An entity can only have one static type but can have multiple dynamic types.
The dynamic type will always conform(is a descendant of) it's static type.
A expanded type conforms only to itself.

8.2.3 Static typing

When a language is static typed the compiler checks that there will be no type faults. If you call x.f, x will always have a feature f.

8.3 Dynamic binding

8.3.1 Redefinition

When a class B inherits from a class A, you can redefine a feature that was already declared in the class A to function differently in B.

```
class B inherit A
        redefine
                feature
        end
create
feature
invariant
end
```

You can not have two features with the same name in A and B.

8.3.2   Dynamic binding

With polymorphism we have an object T that may be either it's static type A or it's dynamic type B. Both A and B have a feature T.r where r has been redefined in B. The effect of the call T.r will depend on the dynamic type of T.(Vs dynamic binding which will call the static type feature (c++)).
Static typing will guarantee that there is <u>at least one</u> version of the feature that will be able to be called.
Dynamic binding guarantees that every call will use the <u>most appropriate version</u> of the feature.

8.3.3   Single choice principle

If a system supports several variants of a notion, knowledge of the set of variants should be limited to one module(principle behind dynamic binding).

9   Design by contract

9.1   Preconditions

Expresses what a supplier routine requires to function properly when invoked by the client. The responsibility for a correct invocation lies on the client. They are boolean expressions checked every time the routine is invoked.

9.1.1   Structure

```
function()
        require
                precondition
        do
```

```
        ...
    ensure
    end
```

A precondition is generally preceded by a text label to make it more readable, example of a precondition: value_positive: a ¿ 0 (do not need to write this during the exam waste of time).

Precondition can be linked together using and, or, not, = implies. Writing two expressions on two consecutive lines is considered as an implicit and. You can also invoke separate boolean expressions that are too complicated or too long to put in the precondition.

If a client invokes a routine and doesn't satisfy the precondition, an exception will be invoked, the line and the label of the precondition will come in handy.

9.1.2   Design by contract vs defensive programming

Design by contract assumes that one should trust clients to invoke a routine correctly. The checks for correct routine invocation are encoded in the supplier using preconditions. The checks can be disabled at runtime if we are sure of the code. Preconditions are intended for scenarios that should never happen(a fail points to an error in the code). Preconditions should be as strong as possible, so that if they do not fail, the program should run as expected.

9.2   Postconditions

Expresses what a supplier routine guarantees if invoked properly by the client. Responsibility for a correct result lies on the supplier. Can be disabled if sure that the code is correct.

They are boolean expressions checked after the execution of the body.

Postconditions are located after the ensure in the routine. They are the same thing as preconditions otherwise.

9.2.1   Old keyword

Old is used to see how a value has changed after the routine body execution. For example: value_added: value = old value + a

In postconditions we specify what has changed during the execution and how it has changed.

9.3 Class invariants

Class invariants express properties that all objects of the class must satisfy. They are boolean expressions checked every time a supplier's status is observable by clients, as soon as an object is created, before and after a feature is available to clients.

9.3.1 Structure

class classname
invariant
    boolean expressions

they are written the same as preconditions.
Class invariant are aloud to be broken in these two scenarios: before the invocation of a creation feature and in the body of any routine. The invariant has to always hold when it matter: when the values of the object are available to clients via queries

9.4 Contracts and inheritance

This section explains what happens to contracts in inheritance, polymorphism and dynamic binding.

9.4.1 All you need to know

We take the point of view of a client C accessing a feature f of class A and redefined in classB that inherits form A.
The precondition in the redefined f can only stay the same or be weakened the postcondition in the redefined f can only stay the same or be strengthened
the class invariant in the descendant class can only stay the same or be strengthened.

If i want to redefine a precondition of a redefined function f then I have to write:
f
    require else precondition

This new precondition has to be weaker than the old one.

The similar goes for postcondition but, fi you want to change the postcondition you have to write:

ensure <u>then</u> postcondition

This postcondition has to be stronger than the old one.
For invariants, it is similar for the postconditions and you just write new class invariants when writing the class and <u>the new invariant will be added to those of the parent class</u>, they also have to be stronger. If you are happy with any of the old post, pre or invariant just dont write require, ensure or invariant.

9.5   Putting it all together

For performance, you can enable or disable(also partially) contracts at runtime(disable after program tested).
Desing by contract allows tests to be generated automatically.
Contracts should never be used to validate user input(or external sources).
Too weak == sound == incomplete, too strong == unsound == complete.