ABAP was initially developed as a procedural language (just like earlier procedural programming language like COBOL).

But ABAP has now adapted the principles of Object-Oriented paradigms with the introduction of ABAP Objects.

The Object-Oriented concepts in ABAP such as class, object, inheritance and polymorphism are essentially the same as those of other modern Object-Oriented languages such as JAVA or C++.

History of Programming Languages

- ABAP Objects is a true extension of ABAP

A typical program consists of type definition and data declarations, which describe the blueprint of the data the program uses when it is executed.

To make your program more readable and for better program structure, it is recommended that you work with modularization units (encapsulated units with functions), such as form routines or function modules.

These components can be reused in many different programs.

Object-Oriented Approach: Key Features

- It represents real-time objects in the form of class objects.
- Hence, Objects are as abstraction of the real world.
- Processes can be put into effect realistically.
- Easier to understand, maintain as well as reuse.
- Software structure is improved as well as the consistency is maintained in development process.
- Maintenance expenses are reduced as there will be less requirement.
- Easier to deliver software components on time with good quality.

Object-Oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operation (functions) that can be applied to the data structure.

In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object with another. For example, objects can inherit characteristics from other objects.

One of the principal advantage of Object-Oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented program easier to modify.

As solutions are designed in terms of real-world objects, it becomes much easier for programmer and business analysts to exchange ideas and information about a design that uses a common domain language.

These improvement in communication help to reveal hidden requirements, identify risks and improve the quality of software being developed. The object-oriented approach focuses on objects that represent abstract or concrete things of the real world. These objects are defined by their character and properties that are represented by their internal structure and their attributes (data). The behavior of these objects is described by methods (i.e. functionality).

Procedural and Object-Oriented Programming Comparison

Features	Procedure Oriented Approach	Object Orient Approach
Importance	Tasks are important	It is on who/what does those tasks
Modularization	Programs can be further divided into smaller sections termed as functions.	Programs are arranged into classes and objects. The functionalities are incorporated inside methods of a class.
Data Security	Functions share global data mostly	Data can be made restricted to external sources.
Extensibility	Time consuming to modify/extend the existing functionality.	Data and functions can be added whenever required easily.

How Does it look like in Real World

The left part of the slide shows that, with procedural software systems, data and functions are often:
-> Created Separately
-> Stored Separately
-> Linked with input or output relations

Objects form capsules the data itself and the behaviour of that data. Objects enable you to draft a software solution that is one-to-one reflection of the real-life problem area.

Encapsulation : Encapsulation conceals the functional details of a class from objects that send messages to it.

Polymorphism : Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior.

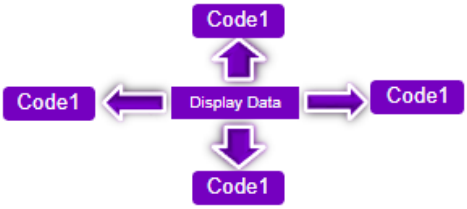
Abstraction : Abstraction is simplifying complex reality by modelling classes appropriate to the problem and working at the most appropriate level of inheritance for a given aspect of the problem.

Inheritance : Subclasses are more specialized version of a class, which inherits attributes and behaviour from their parent classes. And can introduce their own.

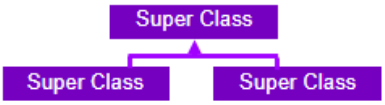
Technical Implementation

- Encapsulation of data and functions

- Polymorphism for support of generic programming



- Inheritance



- Improved structuring and consistency in the software development process

Object behave like client/server systems: When an object calls a method of another object, it automatically becomes the client of the other (server) object. This gives rise to two conditions:

- The client object must adhere to the **protocol** of the server object.
- The protocol must be clearly described so that a potential client can follow it without problem.
- Objects normally adopt both roles. Every object is a potential object and when it is called by a method of another object it becomes a client object too.
- Establishing logical business and software/technical responsibilities between classes results in a true client/server software system in which redundancy is avoided.

Characteristics of a Class

Set of features that have the same structure and performs the same behavior.

Provides a sketch or a blueprint for all instances created using the same class.

Comprises of elements like attributes, constants, types, methods, events and implemented interfaces, defined in the definition part. In the implementation section, you may write your source code.

It is not possible to define a class inside another class. Anyhow, local auxiliary classes for global classes can be defined.

Type of Class: Local Class and Global Class

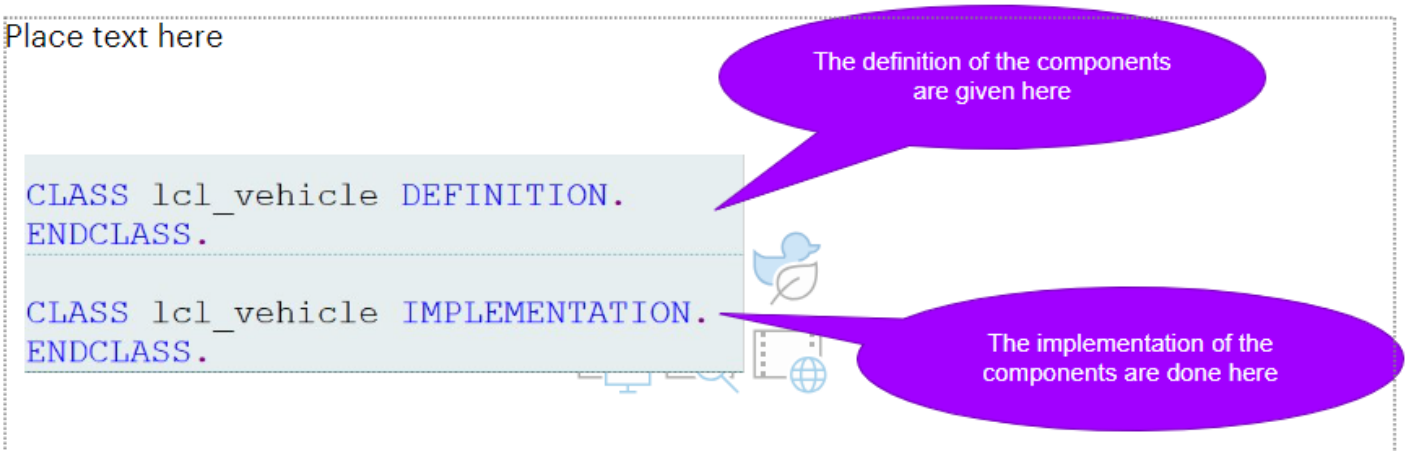
Local classes are implemented inside an ABAP program. They can be used only inside the program where they are defined.

Global classes are implemented in t-code SE24. All the programs in the system may access them.

When you use a class in an ABAP program, the system first searches for a local class with the specified name. If it does not find one, it then looks for a global class.

Local Class Example

Place text here



Attributes in Class

Attributes decide what type of data is stored in objects of particular class.

The content of the attributes determines the behavior of the object.

Defined in the declaration part of the class

Attributes can have any kind of data type:

- ✓ C, N, I, P,....STRING
- ✓ Dictionary types
- ✓ User-defined types
- ✓ TYPE REF TO
  - ➢ Defines a reference to an object

Types of Attributes

Instance Attributes

- One per instance
- Syntax keyword: DATA

Static Attributes

- One class can have one such attribute only
- Syntax keyword: CLASS-DATA
- Also referred as class attributes

Static attributes usually contain information that applies to all instances. Data that is the same in all instances
Administrative information about the instances in that class (for example, counters, ...)
You may come across the expression "class attributes" in documentation, however, the official term in ABAP Objects (as in C++, Java) is "static" attributes.

Attributes and Visibility

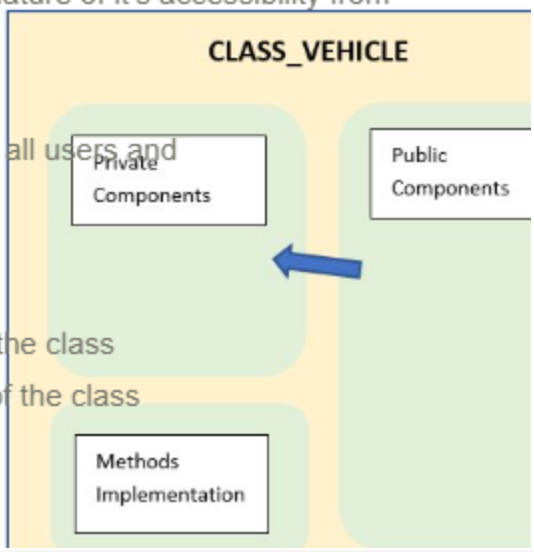
Visibility of attributes decides the nature of it's accessibility from outside class

Public Attributes:

- Can be viewed and changed by all users and in all methods
- Direct Access

Private Attributes:

- Can only be viewed only within the class
- No Direct Access from outside of the class



You can protect attributes against access from outside by characterizing them as private attributes (defined in the PRIVATE SECTION).
Attributes and their values that may be used directly by an external user are public attributes and are defined in the PUBLIC SECTION.
In the above example for the class CLASS_VEHICLE, the attribute make is defined as a public attribute.
Public attributes belong to the **interface** of the class, that is their implementation is publicized. If you want to hide the internal implementation from users, you must define internal and external views of attributes.
As a general rule, you should define as few public attributes as possible.

External Access to Public Attributes

There are different ways of accessing public attributes from outside the class:

- You access static attributes using <classname>**=>**<class_attribute>
- You access instance attributes using <instance>**->**<instance_attribute>
- **=>** and **->** are the component selectors

Methods can access all attributes in their class and can therefore change the state of an object.
Methods have a parameter interface (called **signature**) that enables them to receive values when they are called and pass values back to the calling program.
In ABAP Objects, methods can have IMPORTING, EXPORTING, CHANGING, and RETURNING parameters as well as exception parameters. All parameters can be passed by value or reference. (As of SAP R/3 Basis Release 6.10, you should no longer use the EXCEPTIONS parameter for exceptions but use the RAISING addition instead;

You can define a return code for methods using RETURNING. You can only do this for a single parameter, which additionally must be passed as a value. Also, you cannot then define EXPORTING and CHANGING parameters. You can define functional methods using the RETURNING parameter
All input parameters (IMPORTING, CHANGING parameters) can be defined as optional parameters in the declaration using the OPTIONAL or DEFAULT additions. These parameters then do not necessarily have to be passed when the object is called. If you use the OPTIONAL addition, the parameter remains initialized according to type, whereas the DEFAULT addition allows you to enter a start value.

Types of Methods

Instance Methods

- Can utilize both static and instance components when implemented
- Accessed using an instance
- Keyword to implement is METHODS

Static Methods

- Can utilize only static components during implementation
- Only class can access them
- Keyword to implement is CLASS-METHODS

Static methods are defined at class level. They are similar to instance methods, but with the restriction that they can only use static components (such as static attributes) in the implementation part. This means that static methods do not need instances and can be called from anywhere. They are defined using the CLASS-METHODS statement, and they are bound by the same syntax and parameter rules as instance methods.
The term "class method" is common, but the official term in ABAP Objects (as in C++, Java) is "static method".

Calling Instance Methods: Syntax

```
CALL METHOD ro_veh1->get_count.
```

OR

```
ro_veh1->get_count( ).
```

hl->get_count.

Calling Static Methods: Syntax

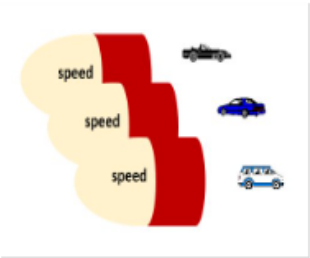
Instance methods are called using CALL METHOD <reference>-><instance_method>.
Since SAP R/3 Basis Release 6.10, the shortened form is also supported; CALL METHOD is omitted.
Special case: When calling an instance method from within another instance method, you can omit the instance name. The method is automatically executed for the current object.
If the method has only one exporting parameter, you can omit the EXPORTING addition in the brackets. In this case, it is sufficient to set the actual parameter of the caller. However, if the method has two or more parameters in its interface, all actual and formal parameters within the brackets must have the EXPORTING addition.
In the case of methods that return parameters to the caller, the IMPORTING addition must always be used and all actual and formal parameters must be listed.
In method calls, multiple parameters are separated by spaces.

Static methods (also referred to as class methods) are called using CALL METHOD <classname>=><class_method>.
Static methods are addressed with their class name, since they do not need instances.
Note:
If you are calling a static method from within the class, you can omit the class name.

In the static method get_count, you can only use the static attribute n_o_vehicles. All other attributes of the class are instance attributes and can only appear in instance methods.

What is an Object?

An object is a special kind of variable that has distinct characteristics and behaviors. The characteristics or attributes of an object are used to describe the state of an object, and behaviors or methods represent the actions performed by an object.



An object has the following characteristics –

Has a state.

Has a unique identity.

May or may not display the behavior.

Multiple instances (objects) of the same blueprint(type of class) are a fundamental attribute of object-oriented languages.

The ability to create multiple instances of a "class", such as a vehicle, is one of the central attributes of object-oriented languages.

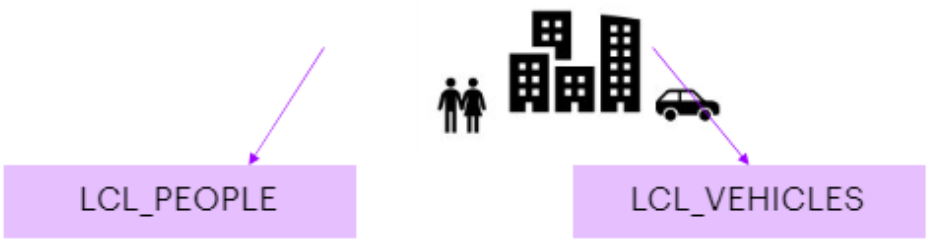
An object is a pattern or instance of a class. It represents a real-world entity such as a person or a programming entity like variables and constants. For example, accounts and students are examples of real-world entities. But hardware and software components of a computer are examples of programming entities.

The state of an object can be described as a set of attributes and their values.
For example, a bank account has a set of attributes such as Account Number, Name, Account Type, Balance, and values of all these attributes.
The behavior of an object refers to the changes that occur in its attributes over a period of time.

Each object has a unique identity that can be used to distinguish it from other objects. Two objects may exhibit the same behavior and they may or may not have the same state, but they never have the same identity. Two persons may have the same name, age, and gender but they are not identical. Similarly, the identity of an object will never change throughout its lifetime.
Objects can interact with one another by sending messages. Objects contain data and code to manipulate the data. An object can also be used as a user-defined data type with the help of a class. Objects are also called variables of the type class. After defining a class, you can create any number of objects belonging to that class. Each object is associated with the data of the type class with which it has been created.

Classification of Objects

A class is a description of a quantity of objects characterized by the same structure and the same behavior. The object is concrete instance of the class.



Class Creation/Modelling

In the real world, there are objects, such as various airplanes , cars, and people. Some of these objects are very similar, that is, they can be described using the same attributes or characteristics and provide the same functions.
Similar objects are grouped together in classes. Each class is described once, and each object is then created in accordance with this blueprint.

A class is therefore a description of a quantity of objects characterized by the same structure and the same behavior.

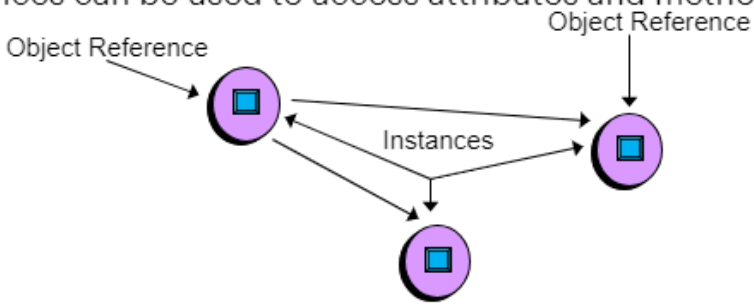
Reference Concept

Object references are used as pointer objects

When an object is created, a reference to the new object is returned to the creator. This can be used for subsequent access to the object. The reference is stored in a reference variable.

Object references (the content of object reference variables) are the only way to access the components of objects in an

ABAP program. References can be used to access attributes and methods, but not events.



When an object is created, a reference to the new object is returned to the creator. This can be used for subsequent access to the object. The reference is stored in a reference variable.
An object is identified by a unique object ID called an object reference. This identity never changes. It distinguishes from other objects that it has the same set of attributes and the same attribute values.
Object attributes can be reference variables.
Different reference variables can point to the same object.
In later versions of ABAP objects, it will be possible to treat objects as values. This allows objects to be embedded in other objects.

Reference Variable ME

You can address the object itself within instance methods using the implicitly available reference variable me.

me=>classmethod()

You can address the object itself within instance methods using the implicitly available reference variable me.
Description of example:
In the constructor, the instance attribute make is covered by the locally defined variable make. In order to still be able to address the instance attribute, you need to use me.
The dummy method demonstrates how to call a class's own method. You can omit the prefix me->.
Other important use:
An object calls another object's method and passes its own address.

A class contains the generic description of an object. It describes all the characteristics that are common to all objects in that class. During the program runtime, the class is used to create specific objects (instances). This process is called instantiation.
Example:
The lcl_vehicle class itself does not exist as an independent runtime object in ABAP Objects.

Implementation:
Objects are instantiated using the statement: CREATE OBJECT.
During instantiation, the runtime environment dynamically requests main memory space and assigns it to the object.

The CREATE OBJECT statement creates an object in the main memory. The attribute values of this object are either initial values or correspond to the VALUE entry.
Reference variables can also be assigned to each other.
For the above example this would mean that ro_veh1 and r_veh2 point to the same object.

Creating Objects (Instantiation)

The object creation generally comprises of following steps –

Create a reference variable in reference to the class. The systax is –

DATA: <object_name> TYPE REF TO <class_name>.

Create an object from the reference variable. The svntax is –

CREATE Object: <object_name>.

```
DATA: ro_veh
      ro_veh
CREATE OBJEC
CREATE OBJEC
```

Functional Method

When defining:

- Only one returning parameter
- Only importing parameters and exceptions are possible

When calling:

- RECEIVING parameters are possible
- Various forms of direct call possible:
  - Using keywords (MOVE, CASE, LOOP)
  - Using logical expressions (IF, ELSEIF, WHILE, CHECK, WAIT)
  - Using arithmetic and bit expressions (COMPUTE),

Example: a = b + c

Methods that have a RETURNING parameter are described as functional methods. These methods cannot have EXPORTING or CHANGING parameters, but has many (or as few) IMPORTING parameters and exceptions as required.

Functional methods can be used directly in various expressions:
Logical expressions (IF, ELSEIF, WHILE, CHECK, WAIT)
The CASE statement (CASE, WHEN)
The LOOP statement
Arithmetic expressions (COMPUTE)
Bit expressions (COMPUTE)
The MOVE statement.

Functional Methods: Example

```
CLASS lcl_test_class DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      func_method
        IMPORTING
          i_param1 TYPE i
          i_param2 TYPE i
        RETURNING
          VALUE(r_return) TYPE char1.
ENDCLASS.

CLASS lcl_test_class IMPLEMENTATION.
  METHOD func_method.
    r_return = i_param1 + i_pa
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION
  DATA: l_retu  TYPE c.
  l_return = lcl_test_class=>func_method(
     i_param1 = 1
     i_param2 = 2
  ).
```

Depending on the number of IMPORTING parameters, the syntax for functional methods is as follows (same for static functional methods):
No IMPORTING parameters: ref->func_method( )
Exactly 1 IMPORTING parameter: ref->func_method(p1) or
ref->func_method(im_1 = p1)
Several IMPORTING parameters: ref->func_method(im_1 = p1 im_2 = p2)
Example of detailed syntax for functional method call:

```
CALL METHOD r_vehicle->get_average_fuel
    EXPORTING im_distance  = 500
              im_fuel      = 50
    RECEIVING re_fuel      = avg_fuel.
```

Here, re_fuel is the formal parameter of the interface and avg_fuel is the actual parameter of the calling program.

Global Class Creation

Classes and interfaces which are maintained globally through Class Builder (Transaction SE24) are known as global classes.

They are saved centrally in class pools (class library of R/3 Repository).

They are accessible by all the ABAP programs in an R/3 System.

Classes also can be created Via SE80

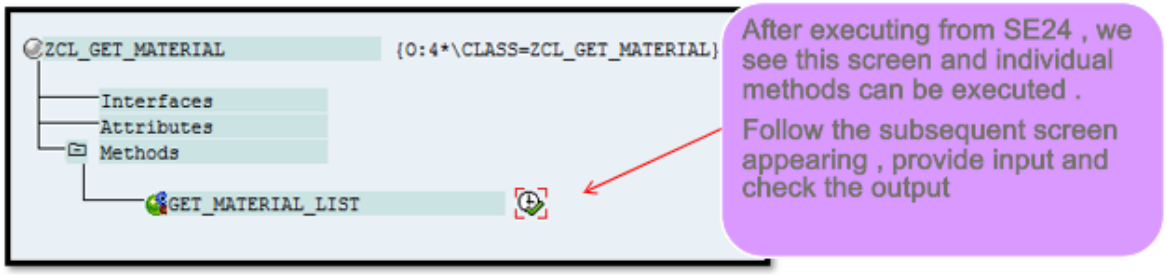
Creation of Global Class - Continued

Use the global class created in a local program

Copy the source code in SE38 and test it.

ZEMP_NO_TEST_CLASS.txt

Also classes can be executed from SE24



Constructor Method

When object (instance) is created in memory, this is the first method that is executed automatically by the runtime environment. This is also known as instance constructor.

It can have only IMPORTING parameters as well as EXCEPTIONS

When CREATE OBJECT statement is executed, this method is called and so for one instance only one constructor call.

Name of the method should be CONSTRUCTOR

CLASS_CONSTRUCTOR is static constructor which is executed once per program

The constructor is a special instance method in a class with the name constructor. The following rules apply:
Each class can have one constructor.
The constructor is automatically called at runtime within the CREATE OBJECT statement.
If you need to implement the constructor, then you must define and implement it in the PUBLIC SECTION.
When exceptions are raised in the constructor, instances are **not** created, so no main memory space is occupied.

You need to implement the constructor when, for example:
You need to allocate (external) resources
You need to initialize attributes that cannot be covered by the VALUE addition to the DATA statement
You need to modify static attributes
You cannot normally call the constructor explicitly.

1. An object is a pattern or instance of a class. It represents a real-world entity.
2. Encapsulation, inheritance, polymorphism, interface
3. A class is a user defined data type with attributes, methods, events, user-defined types, interfaces etc for a particular entity or business application .
4. Attributes, methods and events
5. Instance attribute is available separately for all instances of the class where as static attribute is only one per class irrespective of the number of instances.
6. Method are coding blocks where we write functionality
7. Instance methods can be accessed by all instances of the class, where as static methods are accessible by only class
8. Object references (the content of object reference variables) are the only way to access the components of objects in an ABAP program. References can be used to access attributes and methods, but not events.
9. Special type of method which is called automatically when classes are instantiated.
10. Methods that have a RETURNING parameter are described as functional methods. These methods cannot have EXPORTING or CHANGING parameters, but has many (or as few) IMPORTING parameters and exceptions as required.

11. Inheritance is a relationship, in which one class (the subclass) inherits all the main characteristics of another class (the superclass).
12. Abstract class comprises of both definition and their implementation but instances can't be created
13. Interfaces are independent objects which can be inherited by a class.
Interfaces contains only method definition and a class can implement the method which implements that interface.

Introduction

- When a situation arises during program execution where at one point there is no meaning in continuing the normal program flow, then it is called exception.
- Classical
  - Using SY-SUBRC check
- Class based
  - Using class

SAP NetWeaver Application Server (SAP NetWeaver AS) S 6.10 introduced a new ABAP Objects exception concept that exists parallel to the existing concept based on sy-subrc.
Exceptions and exception handling are now based on classes. This new ABAP Objects exception concept enhanced the classic way of handling exception using sy-subrc.

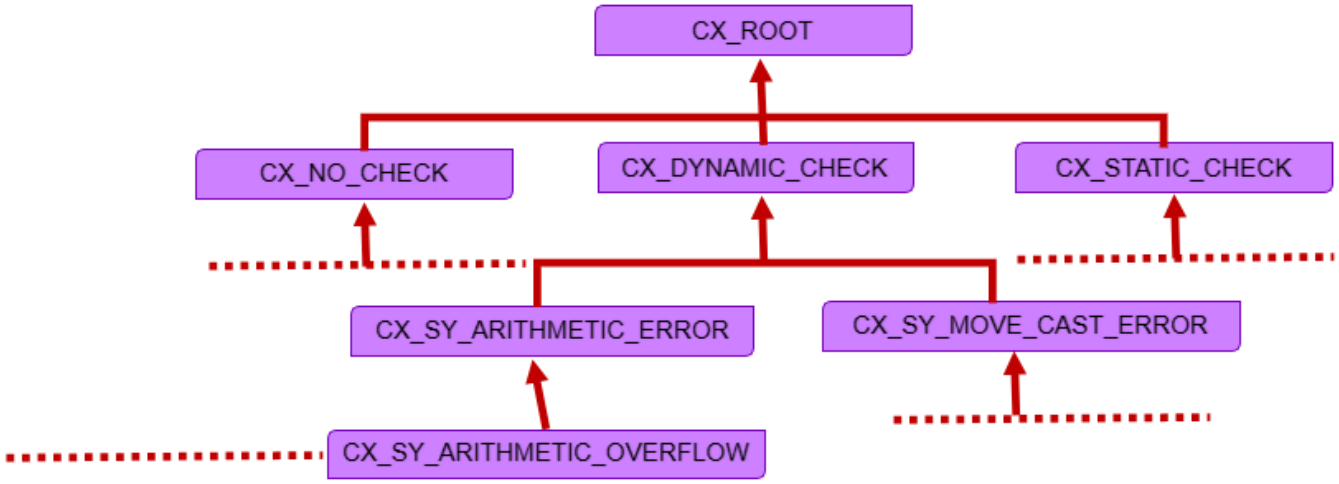
You can handle an exception if the statement that raised it is enclosed inside a TRY-ENDTRY control structure. You handle the exception using the CATCH statement in the TRY-ENDTRY structure.

The TRY block contains the statements for which the exceptions need to be handled. A CATCH block contains the exception handler, which is executed if a specified exception has occurred in the associated TRY block.

Like all control structures in ABAP Objects, you can nest TRY-ENDTRY structures to any depth. In particular, the TRY block, the CATCH block, and the CLEANUP block can contain complete TRY-ENDTRY structures themselves.

Specify any number of exception classes in the CATCH statement. In this way, you define an exception handler for all these exception classes and their subclasses.

Exception Classes: The Inheritance hierarchy



All exception classes are derived from one exception class, CX_ROOT. Therefore, you can generically access any exception object through a reference variable, REF TO CX_ROOT.

However, a new exception class is not allowed to inherit directly from CX_ROOT, derive any new exception class directly or indirectly from one of the subclasses CX_ROOT - CX_NO_CHECK, CX_DYNAMIC_CHECK, or CX_STATIC_CHECK.

Through this, all exception classes are subdivided into three groups. Depending on the group to which a given exception belongs, the exception is treated differently by syntax check and runtime environment. The default group is CX_STATIC_CHECK, which ensures maximum syntax check and program stability. Use the other groups only in special cases.

The GET_SOURCE_POSITION method returns the name of the main program or include program and also the line number in the source code where the exception occurs. The GET_TEXT method returns an exception text in the form of a string. This method is not defined in CX_ROOT directly but in interface IF_MESSAGE, which is implemented by CX_ROOT.

Exception Handling Example

```
PARAMETERS : pa_int1 TYPE i,
             pa_int2 TYPE i.

DATA: gv_result TYPE i,
      gv_text TYPE string,
      gx_exc TYPE REF TO cx_root.

TRY.
    gv_result = pa_int1 * pa_int2.
    WRITE gv_result.
  CATCH cx_sy_arithmetic_overflow INTO gx_exc.
    gv_text = gx_exc->get_text( ).
    MESSAGE gv_text TYPE 'I'.
ENDTRY.
```

If an overflow error occurs, the runtime system raises the CX_SY_ARITHMETIC_OVERFLOW exception

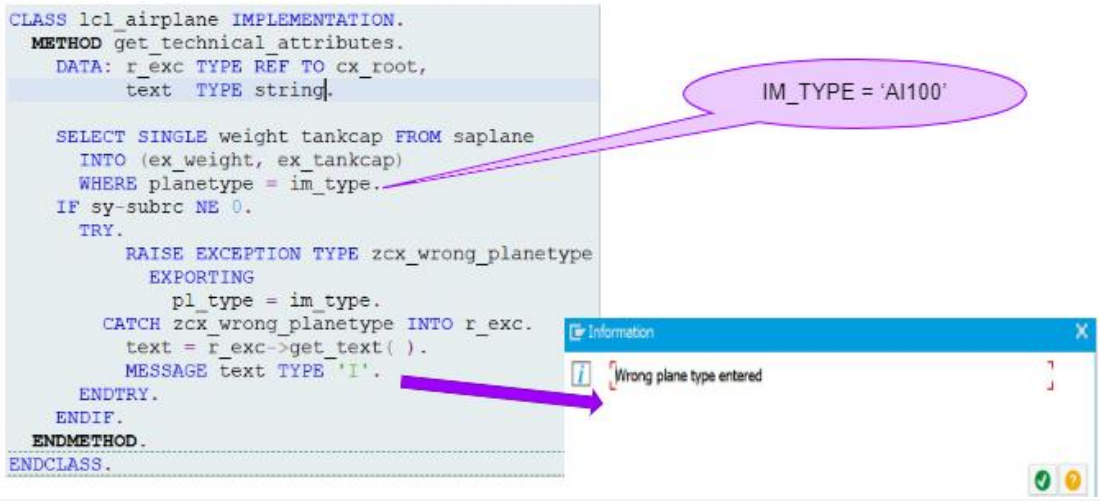
In the above calculation, if the value range for data type i is exceeded, the runtime system raises the exception CX_SY_ARITHMETIC_OVERFLOW. This exception is handled in the implemented CATCH block.
The object reference to the exception object is stored in the reference variable r_exc. Using r_exc and the functional method get_text, the handler accesses the exception text for this exception object and stores in the string variable text.
To display exception texts as messages, the MESSAGE statement has been extended so that you can use any string:
MESSAGE <string> TYPE <type>.
As well as the message <string> that will be displayed, you must display the message type <type>, either as a literal or in a field.
If the value range for data type i is not exceeded, no exception is raised, and the TRY block is processed completely. The program then continues executing after the keyword ENDTRY.
The class CX_SY_ARITHMETIC_OVERFLOW is a subclass of the classes CX_SY_ARITHMETIC_ERROR, CX_DYNAMIC_CHECK, and CX_ROOT. Thus, the exception raised above can also be handled if you enter one of these classes after the CATCH statement.
The keyword documentation for each keyword lists the exception classes whose exceptions may occur when the appropriate ABAP statement is executed.

The above program source code shows the method get_technical_attributes of the class lcl_airplane.. It receives an airplane type as an import parameter and returns its weight and tank capacity as export parameters.
The relevant information is read from the database table *saplane*. If the airplane type passed is not available in this table (that is, if sy-subrc <> 0), the values 100.000 and 10.000 respectively are assigned to the export parameters ex_weight and ex_tankcap. We will now change this behavior: If an airplane type is not entered in the table, an exception that we have defined should be raised and handled appropriately.

Exceptions are represented by objects that are instances of exception classes. Defining an exception is thus synonymous with creating an exception class.
Exception classes are generally defined globally. For special exceptions that will only occur within a single ABAP program however, you can also define local exception classes.
Global exception classes are defined and managed in the Class Builder. When you create a new class, if you use the correct naming convention (prefix ZCX_) and choose the class type *Exception Class*, the system automatically displays the *Exception Builder* instead of the Class Builder.
The *Exception Builder* offers all the functions you need to create exception classes and generates specified components that cannot be changed.
When you create an exception class, you must also specify which category of exception it will be - that is, whether it is derived from CX_STATIC_CHECK, CX_DYNAMIC_CHECK or CX_NO_CHECK.

The methods are all inherited from CX_ROOT. You can also add your own methods. The instance constructor is generated automatically.
You can also define your own attributes, whose contents specify the exception in more detail. The *Exception Builder* ensures that the instance constructor has identically-named IMPORTING parameters for these attributes.
The exception texts of global classes are defined on the *Texts* tab of the Exception Builder. They can contain parameters. To do this, use the elementary attributes of the exception class by enclosing their name in ampersands ('&') in the exception text.

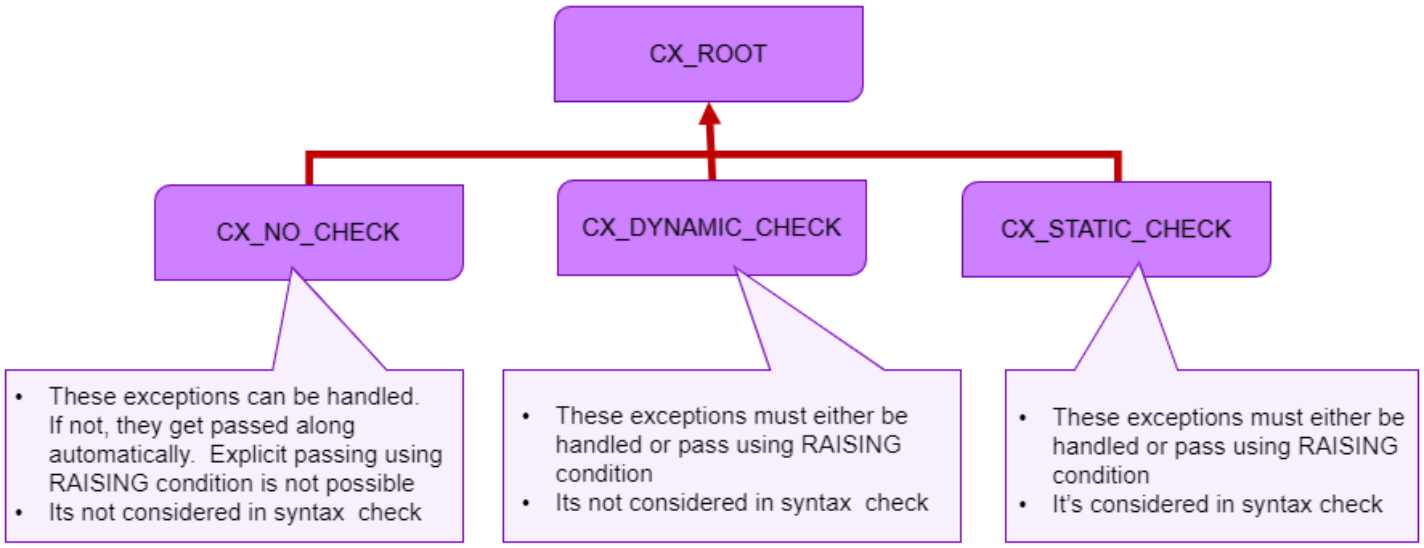
Raising and Handling Exceptions You have Written



If the airplane type passed to the method has not been stored in the table *saplane*, the exception we defined previously, *zcx_wrong_planetype*, is raised. In addition, a TRY-ENTRY control structure is implemented that is only processed if sy-subrc <> 0.
The TRY block contains the application code that is to handle the exceptions. When the exception is raised, the IMPORTING parameter *pl_type* of the instance constructor is filled. (This parameter is automatically generated by the Exception Builder.) Using this parameter, the program then assigns the value of the airplane type to the identically-named attribute.
The exception that has been raised is handled in the CATCH block. The reference to the exception object is stored in the reference variable *r_exc*, which was created as a local data object in the method (TYPE REF TO cx_root).
Since the IMPORTING parameter TEXTID of the instance constructor was not filled when the exception was raised, the default text generated when the exception class was created is addressed using the functional method *get_text*. The method GET_TEXT then exports this text, replaces the text parameter with the contents of the attribute *pl_type*, and returns the text as a character string.
The returned text is stored in the local data object *text*, which has the type *string.* The text is then displayed as an information (type **I**) message.

Exceptions that occur in procedures (methods, function modules, or subroutines) do not necessarily need to be handled there; they can be passed along to the calling program. The calling program can then handle the exception itself or also pass it along to its own caller, and so on.
The highest levels to which an exception can be passed are processing blocks without local data areas - that is, event blocks or dialog modules. The exceptions passed along by the called procedures must be dealt with there, as must any exceptions raised within this processing block itself. Otherwise, runtime error occurs.
To pass along an exception from a procedure, you generally use the RAISING addition when defining the procedure interface.
In methods of local classes and subroutines, specify the RAISING addition directly when defining the procedure (METHODS meth ... RAISING cx_... cx_..., FORM ... RAISING cx_... cx_...). After RAISING, list the exception classes whose objects are to passed along.
In methods of global classes, the exception classes whose objects are to be propagated are entered in the exception table of the method in the Class Builder. Check the *Exception Class* field in this exception table. Similarly, exceptions raised by function modules are passed along by being entered in the Function Builder.

Exceptions that Must be Declared



Subclasses of CX_STATIC_CHECK: The relevant exception must either be handled or passed along explicitly using the RAISING addition. The syntax check ensures that this is the case. At present, only exceptions you define yourself for error situations in the application code are subclasses of CX_STATIC_CHECK.
Subclasses of CX_DYNAMIC_CHECK: The relevant exception does not have to be declared. If such an exception occurs at runtime, just as with subclasses of CX_STATIC_CHECK, it must either be handled or passed along explicitly using a RAISING addition. However, this is not checked in the syntax check. If such an exception occurs at runtime and is not either handled or passed along, a runtime error occurs. Most predefined exceptions with the prefix CX_SY_... for error situations in the runtime environment are subclasses of CX_DYNAMIC_CHECK.
Subclasses of CX_NO_CHECK: These exceptions cannot be declared. These exceptions can be handled. Otherwise they are automatically passed along. The syntax check never finds an error here. All exceptions of the category CX_NO_CHECK that are not handled in the call hierarchy are automatically passed to the top level, If they are not caught there, they cause a runtime error. Some predefined exceptions with the prefix CX_SY_... for error situations in the runtime environment are subclasses of CX_NO_CHECK.

Abstract Class

- Abstract class comprises of both definition and their implementation, but instances can't be created.
- Use the keyword ABSTRACT in the CLASS statement to make an abstract class.
- One of the common usage is to create Super classes, where they are not to be instantiated by themselves, but their subclasses need to.
- Abstract class should at least contain one abstract method.
- Abstract methods are methods without any implementation – only a declaration
- It can also have instance methods (non-abstract method) which should be implemented within the same abstract class

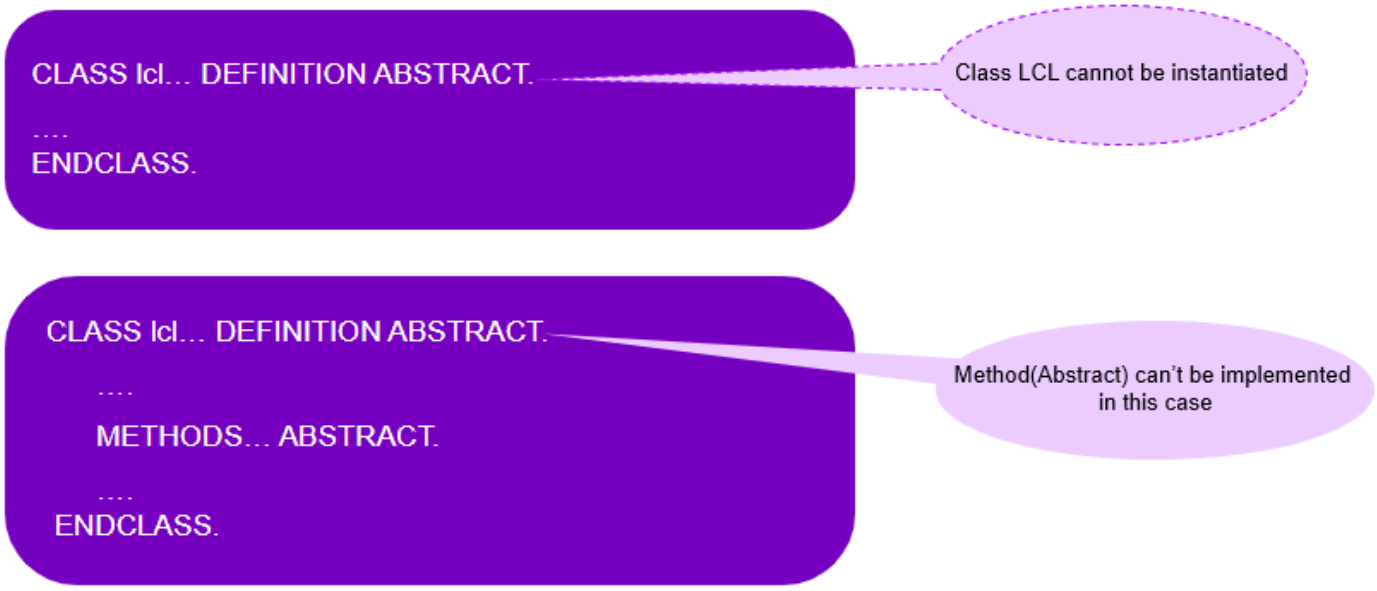
In an abstract class, you can define abstract methods along with other methods. This means that the abstract method cannot be implemented in that class. Instead, it is implemented in a subclass of the class. If the subclass of that class is not abstract, the abstract methods must be redefined and implemented in the subclass for the first time.
The relevant indicator is in the Class Builder on the *Attributes* tab page for that class or method.
References to such abstract classes can be used for polymorphic access to subclass instances. Static methods cannot be abstract because they cannot be redefined.

Abstract Class

- It's not possible to instantiate ABSTRACT classes (however their subclasses can).
- References to abstract classes can point to instances of subclasses.
- ABSTRACT (instance) methods can be defined in the class but cannot not be implemented.
- ABSTRACT Methods must be redefined in the subclasses.
- Instance methods needs to be implemented within the ABSTRACT class

It is not possible to instantiate objects of an abstract class However, this does not mean that references to such a class are not useful. On the contrary, they are very useful, since they can (and must) refer to instances in subclasses of the abstract class at runtime. The CREATE OBJECT statement is extended in this context. You can specify the class of the instance to be created explicitly:
CREATE OBJECT <RefToAbstractClass> TYPE <NonAbstractSubclassName>.
Abstract classes are normally used as an incomplete blueprint for concrete (that is, non-abstract) subclasses, for example to define a uniform interface.
Abstract instance methods are used to specify particular interfaces for subclasses, without having to immediately provide implementation for them.
Abstract methods need to be redefined and thereby implemented in the subclass (here you also need to include the corresponding redefinition statement in the DEFINITION part of the subclass).
Classes with at least one abstract method are themselves abstract.
Static methods and constructors cannot be abstract (they cannot be redefined).

Abstract Class



Inheritance is a relationship, in which one class (the subclass) inherits all the main characteristics of another class (the superclass). The subclass can also add new components (attributes, methods, and so on) and replace inherited methods with its own implementations.
Inheritance is an implementation relationship that emphasizes similarities between classes. In the example above, the similarities between the car, bus, and truck classes are extracted to the vehicle superclass. This means that common components are only defined/implemented in the superclass and are automatically present in the subclasses.
The inheritance relationship is often described as an "is a" relationship: A truck **is a** vehicle.

Relation between Superclass and Subclasses

Common components exists only once in the superclass.

- ✓ New components in the superclass are by default available in subclasses.
- ✓ Here by doing so, the amount of fresh source code is reduced

Subclasses are totally dependent on super classes.

- ✓ "White box reuses": Subclass should know in detail about the superclass implementations.

If inheritance is used properly, it provides a significantly better structure, as **common components only need to be stored once centrally** (in the superclass) and are then automatically available to subclasses. Subclasses also benefit from modifications (however, they can also be invalidated as a result).
Inheritance provides very strong links between the superclass and the subclass. The subclass must possess detailed knowledge of the implementation of the superclass, particularly for redefinition, but also in order to use inherited components. Even if the superclass does not technically know its subclasses, the subclass often makes additional requirements of the superclass, for example, because a subclass needs certain protected components or because implementation details in the superclass need to be changed in the subclass in order to redefine methods. The basic reason is that the developer of a (super)class cannot normally predict all the requirements that subclasses will later need to make of the superclass.

Normally the only other entry required for subclasses is what has changed in relation to the direct superclass. Only **additions** are permitted in ABAP Objects, that is, in a subclass you can "never take something away from a superclass". All components from the superclass are automatically present in the subclass.
The attributes of the superclass lcl_fruits exist in the subclass lcl_apple; the method 'properties' is also available in the subclass.
The subclass defines a method 'variety'. It is not visible in the superclass.

The REDEFINITION statement for the inherited method must be in the same SECTION as the definition of the original method. (It can therefore not be in the PRIVATE SECTION, since a class's private methods are not visible and therefore cannot be redefined in subclasses).
If you redefine a method, you do not need to enter its interface again in the subclass, but only the name of the method. The reason for this is that ABAP Objects does not support overloading.
In the case of redefined methods, changing the interface (**overloading**) is not permitted; exception: Overloading is possible with the constructor.
Within the redefined method, you can access components of the direct superclass using the SUPER reference.
To implement a redefined method in a subclass, you often need to call the method of the same name in the *immediate* superclass. In ABAP Objects you can call the method from the superclass using the pseudo-reference super:
The pseudo-reference super can only be used in redefined methods.

Inheritance should be used to implement generalization and specialization relationships. A superclass is a **generalization** of its subclasses. The subclass in turn is a **specialization** of its superclasses.
The situation in which a class, for example lcl_sub2, inherits from two classes (lcl_super1 and lcl_super2) simultaneously, is known as multiple inheritance. However, this is **not** implemented in ABAP Objects. ABAP Objects only has single inheritance.
You can, however, simulate multiple inheritance in ABAP Objects using interfaces (see the section on interfaces).
Single inheritance does not mean that the inheritance tree only has one level. On the contrary, the direct superclass of one class can in turn be the subclass of a further superclass. In other words: The inheritance tree can have any number of levels, so that a class can inherit from several superclasses, as long as it only has one direct superclass.
Inheritance is a "one-sided relationship": Subclasses know their direct superclasses, but (super)classes do not know their subclasses.

Interface Concepts

- Interfaces are independent objects which can be inherited by a class.
- Interfaces contains only method definition and a class can implement the method which implements that interface.
- It is possible to implement same interface in multiple classes.
- Thus, it is possible to achieve polymorphism of interface methods
- The main purpose of interfaces is to achieve multiple inheritance and re-usability.
- Interface can be defined using t-code SE24 or through Eclipse in Class builder as shown in screenshot.