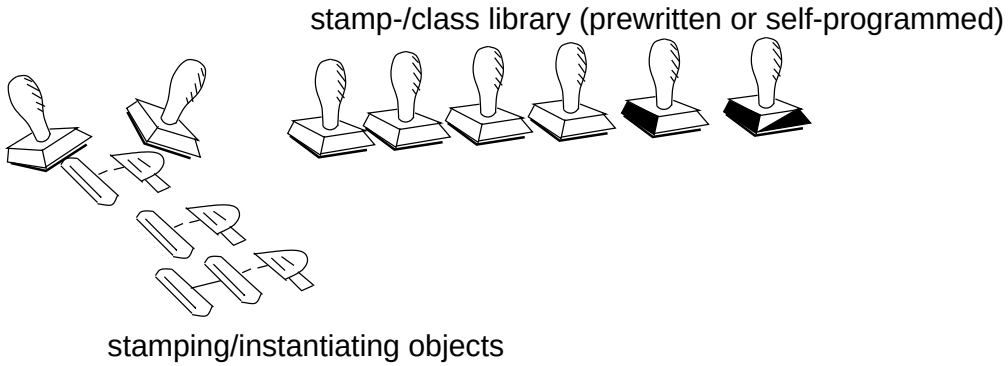
Structuring Code and Data: Object-Orientation

Structuring Code and Data: Object-Orientation — Methods and, if those methods properly encapsulate their functionality, the associated Application Programming Interfaces (APIs) are useful for structuring source-code. If properly designed, even large pieces of source-code, which have been structured with the means of proceduralization, do work and can be maintained. Nevertheless there are many (real-life) programming problems that are open to another way of structuring: Object-Oriented Structuring of source-code.

Historically the object/class-model emerged from trying to simulate physical real-life objects and their interactions[45]. There, classes characterize sets of objects with common properties; this approach can be seen as the more general one.

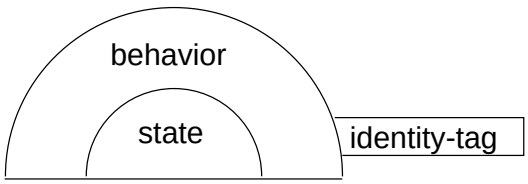
Another approach starts by developing programmer-defined types: A number may be thought as being characterized by its data AND the methods for its manipulation, like addition and subtraction, multiplication and division.

Since real-life objects often can be seen as displaying a considerable complexity, their relevant properties have to be isolated. That process of model-building is also called abstraction. Object-Orientation has a considerable conceptual overhead, but a readily approachable aspect can be shown with the stamp-analogy: Classes within object-oriented structuring are like stamps (which the programmer can carve or take from a library). With these classes the programmer can instantiate objects, much like the stamps can be used to produce several prints, into which it is possible to write data by hand. There even may be little stamp collections, which appropriately combined ("extended") produce an new type of printed matter.



Concept: Object/Class-Model

Concept: Object/Class-Model — This section just introduces the central notion of the Object/Class-Model: An object is defined as an entity having a state, a behavior and an identity[46].



[45] Nygaard, Kristen and O. J. Dahl. *The Development of the Simula Languages* in History of Programming Languages. Academic Press. New York, NY. 1981.

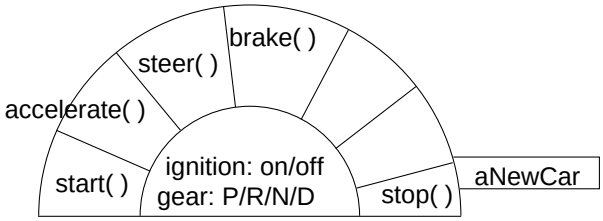
[46] In object-oriented programming-languages the "state" of an object usually is given by variables and the "behavior" is given by methods. Though the identity often may be thought as being given by a variable-name, this may not be true when variable-names are pointers, that may change their pointing-direction! This is the case in Java; there, an identity-tag has to be retrieved by a special `hashCode()`-method, as introduced on page 161. CAUTION: Variable-names in Java do not fix the identity of an object!

In the graphical representation the identity-tag often is omitted, because of self-evidence. But the identity gives the means to discern two objects with the same state and behavior. Two objects with the same state and behavior are called equal (but they are not the same due to their different identities).

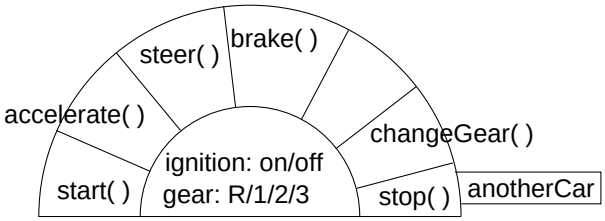
Later, the object-model is supplemented by an equally generally defined class-model, see page 140.

Example: Object, Class and Class-Hierarchy

As an example of such a general object-notion may be taken a new car coming from a production line: Although the cars look alike, the car under consideration is a unique one, it has an identity. It has also a state describable by the gauges for gasoline, motor oil, braking fluid, gear box fluid, cooling water, battery voltage, engine on/off, lights on/off, blinker right on/off, blinker left on/off, warning blinker on/off and other dashboard indicator positions. The behavior of that car, starting the engine, driving forward, driving backward, changing the direction of driving is described by more or less complex sequences of actions as found in the handbook of the car (or to be learned by driving it). The graphic below presents two objects which can be seen as simplified representations (so-called abstractions) of a real car:



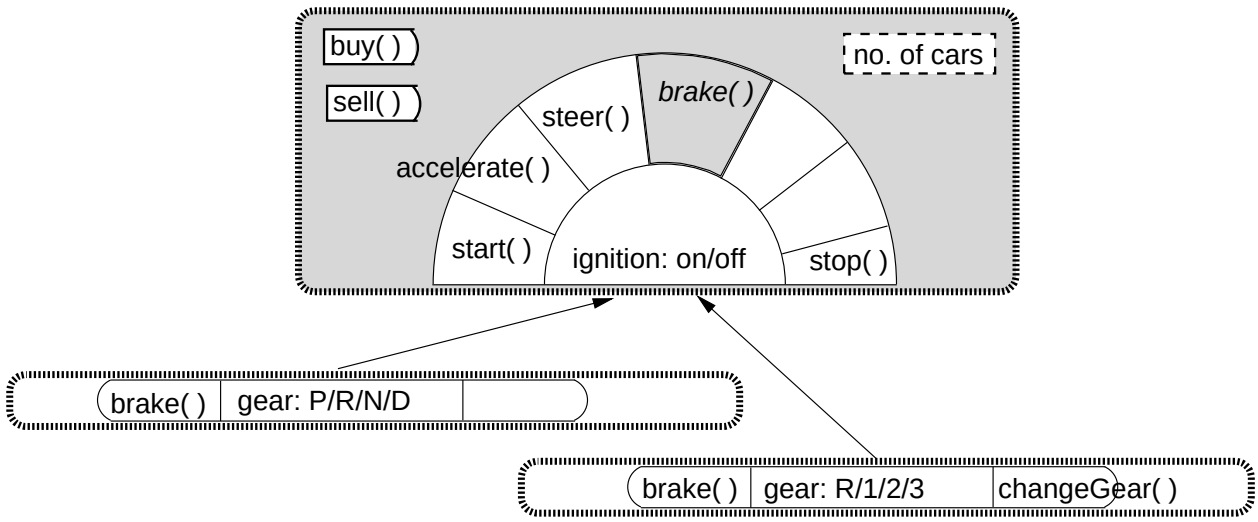
car with automatic transmission



car with mechanical gearbox

One car may have an automatic transmission. Another car may have a mechanical gearbox, this changes the possible state of the car (possible gear-positions are 1/2/3 instead of P/N/D) and this even extends the behavior of the car by an explicit complex changeGear()-procedure. So far, the structure of the above objects can be overseen easily. This view may change in cases where ten or hundred cars have to be considered, abstracted in the way above. If more cars are to be collected, then the different structure of the objects would become clearer by isolating common states and behaviors to get the pure structure of the collection of objects:

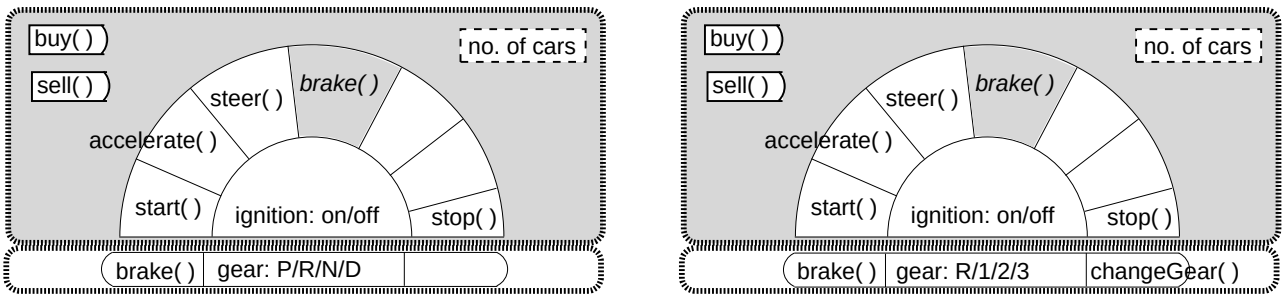
Car- Class Structure



(The various features, like rounded frames and grayed backgrounds, of the graphic above will be described successively in the following paragraphs.) Of course, the pure structure above does not relate directly to any individual object, but any given individual object can be classified fast and easily by using the structure above. The structure just collects all the different features used to describe the two kinds of cars above. The iglu-shaped part of that analysis describes a sort of basic car structure giving no information about what gearbox-system such a car contains. The extension of that structure produces two new structures: One of those extensions describes a car with an automatic transmission, and the

 ISBN 3-00-009429-6

other describes a car with a mechanical gearbox. The rounded frame with the iglu-like figure inside represents a structure, called a **class**. That **class can be extended** by adding one of the little frames: class, class-extension



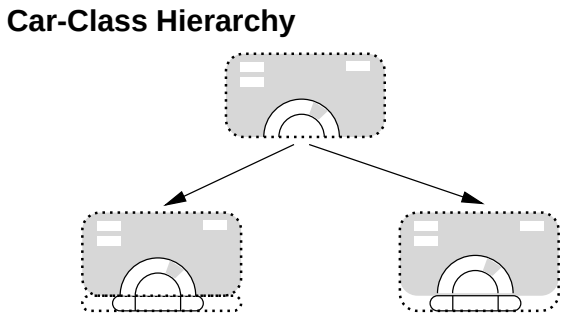
The resulting new classes above can be used as templates for producing objects (Car-objects[47] with automatic transmission and Car-objects with a mechanical gearbox). The frame around the iglu-shape, or around the burger-like shape indicates that it is a class or part of a class-structure. Producing objects from a class is also called instantiating the class or **generating an instance of the class** or **deriving the object from the class**. This instantiating process in a way resembles the stamping-act: The class is used as a stamp for printing out one or more objects. instantiating a class

The structural analysis of the Car-objects produced a **class-hierarchy** given by the abstract Car-class, without any gear-specification, and the two extended classes, above: A class for cars with a mechanical gear-box and a class for cars with an automatic gearbox. class-hierarchy

Buying a car can be seen as instantiating the car in your living area. Selling removes it. Buying, selling or scrapping the same car more than once in the time of your ownership would be unusual. So buying, selling, scrapping can be seen as actions that are part of the behavior of the Car-class rather than reformulating them for individual Car-objects (The sales-man or the car-crunching machine are not integral parts of the car's state or behavior!). These behaviors can be assigned to the Car-*class* and then are called **class-behavior**. class-behavior

The number of Car-objects, which are described with this class-model, neither characterizes the individual Car-object; but that number describes the *set of Car-objects* which is considered. So, the "number of Car-objects" can be seen as a **class-state**, to be associated with the Car-*class*. In the frame that fixes the class-structure, the class-states and class-behavior are collected as squares or square-like boxes in the upper corners. class-state

The graphics above allow another interesting observation: The base-class, that can be extended, cannot be sensibly instantiated; the Car-object would have an undefined gearbox and therefore couldn't describe an average real-world car. – Such classes which cannot be sensibly instantiated are also called **abstract classes**. But, as can be seen above, abstract classes can be used for devising clearer structures and abstract classes can have a defined behavior and a definable state which they transfer onto the extended classes (the ignition on/off state and the other behaviors) and instantiated objects thereof. Abstract classes here are written with a gray-underlaid frame, to make them discernable from non-abstract, instantiable classes. (The Java-implementation of the notion "abstract class" is given on page 183 and an elementary example in Java can be found on page 181.) abstract class



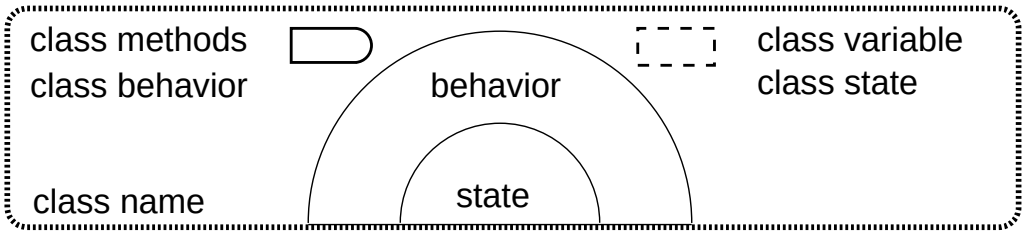
As could be seen above, the very general notion of "behavior" usually is realized as a set of actions or procedures also called **methods**. Whereas the notion of a "state" usually is realized as a set of **variables**. There can be discerned so-called **instance-variables** (ignition with the values on/off, gear position with the values P/R/N/D) used for describing the state of an object and so-called **class-variables** (number of Car-instances) describing the state of the class. methods and variables, instance-variables and class-variables

[47] The upper-case letter "C" indicates that "Car-object" denotes a mental image — the model — of a car, not the real thing!

instance-**methods** and class-methods

The same classification can be applied to the *behavior* of classes and objects: There are **instance-*methods*** (start(), accelerate(), ...) that describe the behavior of the individual object. These instance-methods can be invoked for each object individually (car1.accelerate(slow) and car2.accelerate(fast)). – But there are also **class-*methods*** which describe behavior of the entire class: Like Car.buy(car1) and Car.buy(car2). The act of buying is taken as a class-method, because the buying-process can be described independently from the individual car, whereas for example the price of the car describes an aspect of the car's individual state. Later, in the context of programming-languages, the equivalent to producing and buying a car can be seen as the instantiation of an object by using a class-specific constructor-method.

These remarks make it possible to identify the various abstracted notions in the graphical representation of the class-model:



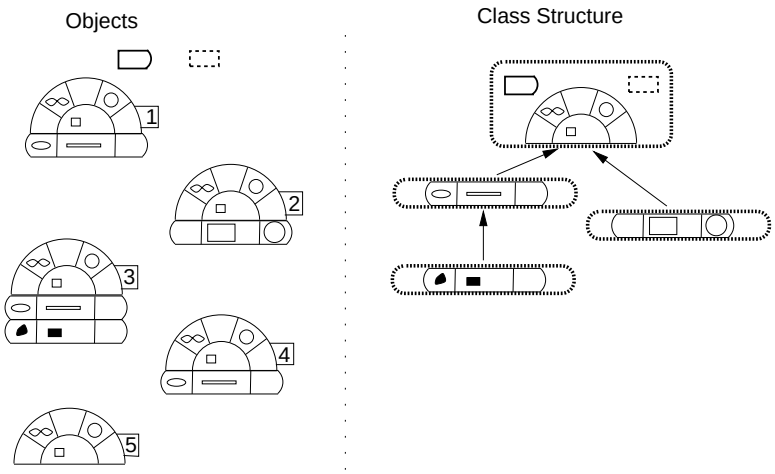
abstract method

Finally, consider a reason why even unimplemented methods, so-called **abstract methods** may be a sensible idea: The brake()-method is a method of the abstract Car-class. But the braking-process differs in a car with a mechanical gearbox from that in a car with an automatic gearbox (The clutch should be released during a heavy braking-process.). So, specifying the braking process for the entire Car-class becomes difficult because the brake()-method has to be implemented differently for the two derived classes. But still it may be useful to have at least the idea of a brake()-method within the Car-class, because braking is what every car should be able to do. This necessity can be indicated with a non-implemented (abstract) brake()-method in the Car-class. (The Car-class itself couldn't be instantiated, because there exists no conventional car without a transmission of any kind, and therefore the entire Car-class has to be called abstract.) Of course, a class with at least one abstract (unimplemented) method automatically has to be declared abstract, because objects with unimplemented methods violate the idea of a defined behavior.

abstract class

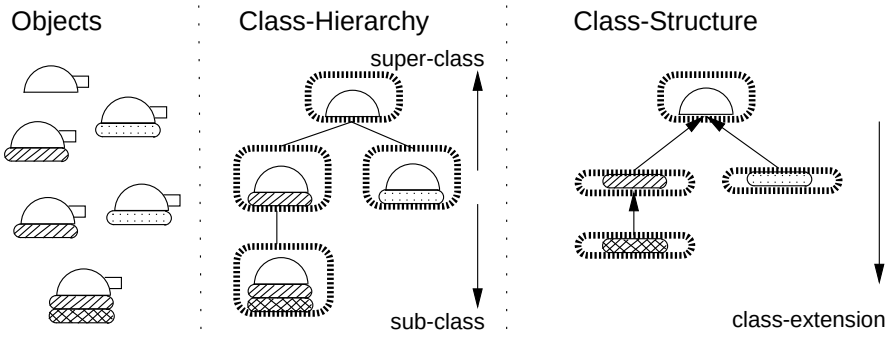
What has done with the brake()-method could have been done also with the accelerate()-method, assuming that acceleration is thought of involving a changing of gears. Why the difference? There exists no particular reason for that difference: Class/object-structures are models chosen by an onlooker for describing an area of interest or a problem according to the onlooker's necessities. THERE IS NO INHERENT ULTIMATE TRUE STRUCTURE TO BE FOUND, the choice above was taken just to illustrate the class/object-model. Of course this freedom, or seemingly arbitrary approach, can be used to properly design classes to depict real-world-problems; with the intention of letting this class/object-model be re-used as much as possible. — Making class-design, problem-oriented as well as anticipating future needs, can turn out to be more difficult than expected!

The relation between a collection of objects, class-methods or class-variables and a given class-structure is restated in the following graphic:



 ISBN 3-00-009429-6

Observe that all objects share the class-variables and the class-methods. Objects Number 1 and Number 4 are equal but not the same! (Their state and behavior are the same but they have been given different identities.)



extension-hierarchy

A given class-structure, like that of the Car-class, makes it easy to construct sub-classes by adding class-extensions. In the (resulting) class-hierarchy, also called extension-hierarchy, those classes that have been extended often are called **super-classes**, their extensions consequently are also called **sub-classes**. Observe also that THE MOST COMPACT DESCRIPTION OF THE CLASSES CONSIDERED GIVES THE ORGANIZATION OF THE CLASS-STRUCTURE. Therefore in programming and documentation, classes are economically declared or described by using the more fragmented but least redundant representation given by the class-structure (and avoiding the organization as a class-hierarchy or worse as a collection of objects).

super-classes and sub-classes

class-structure: most compact description, thus widely used!

The notions “object” and “class” thus can be seen from two sides:

(Class as Template for Creating Objects) A class can be seen as a collection variables and methods that can be of the instance- or class-kind. A class-hierarchy is given by a structured collection of these variables and methods. The class then serves as a template for objects; and the hierarchy of classes represents a tree-structure of successively refined templates. Class-methods and class-variables can be used independently from the existence of an object.

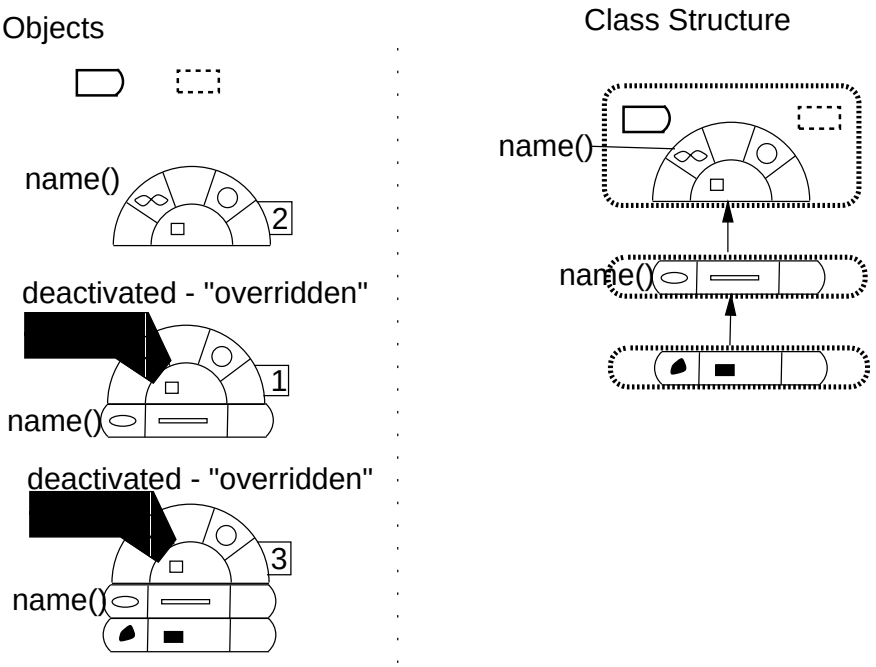
(Object Collection is abstracted to Class) Consider several similar objects *together* with methods that describe how to construct and how to destruct an object: The methods that exist independently from the existence of any object are called class-methods. There may be a variable that has the same value for all objects under consideration, that variable is likely to become a class-variable. Another variable that occurs in some of the objects and contributes to the object’s individuality by having different values from object to object needs to be an object’s state- or instance-variable. Depending on the variety of the given objects, there are compilable different collections of variables and methods. These collections constitute the classes that can be arranged into a class-hierarchy.

So both approaches, the one starting with the class-notion, or the one starting from the object-notion are part of - or result in the same object/class-model.

OUTLOOK: The class-structure is what will be used to define class-hierarchies in Java on page 163. An example of a larger class-hierarchy is given by the graphic on page 564 in the context of arranging a Graphical User Interface; details of the emerging class-structure of the three base-classes (`Component`, `Container`, `JComponent`) are given successively in the lists on pages 553, 556 and on page 559.

Overriding Methods and Shadowing Variables

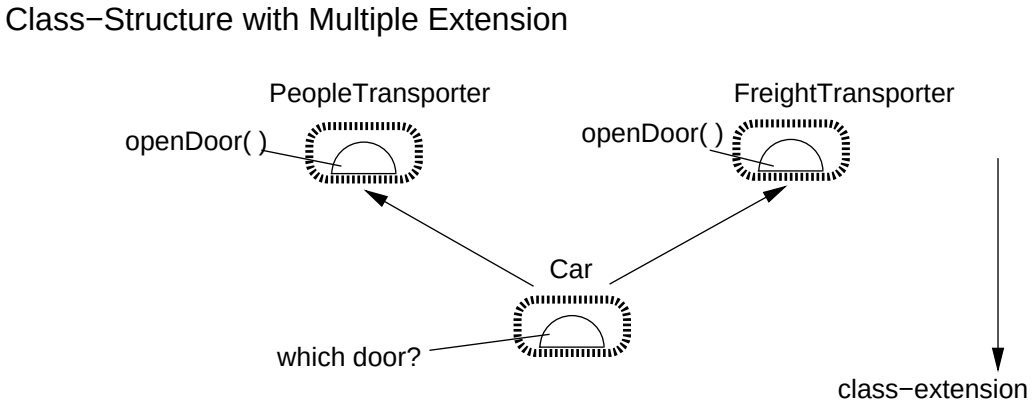
Overriding Methods and Shadowing Variables — Consider the brake()-method in the graphics around page 132; there the abstract method has been redefined differently in both sub-classes. What about redefining non-abstract methods of super-classes? Or formulated conversely, what if a method or a variable in a class-extension has the name of a method or variable in the super-class? First, objects derived from (one of) the super-classes use the original method, BUT all objects derived from the class-extension or its sub-classes use the newly defined (instance-)method or (instance-)variable. The overriding of a method has been visualized in the graphic below:



Remember the shadowing of variables by local variables of Java methods as introduced on page 123, shadowing of variables by class-extension can be imagined likewise. Overriding class-methods and shadowing class-variables would produce *class*-specific variants. For an example consider the Java source-code on page 167 and the corresponding footnote following that source-code.

Class-Extension — Inheritance with multiple Super-Classes

Class-Extension — Inheritance with multiple Super-Classes — Another approach to the process of extending classes may be thought of by imagining a class having *several* super-classes. Example: Below the Car-class extends the class PeopleTransporter and the class FreightTransporter. Each instance of a Car may be used to transport people as well as freight:



Example I: The Car-class may extend the class PeopleTransporter, but at the same time the Car-class may extend the class FreightTransporter. This makes it possible to construct an ambiguity: Both classes may use a method openDoor(), but as can be seen with the example of a standard car, such a method usually means different doors (doors to the area with seats or door of the car's trunk). Though, here, the ambiguity may seem constructed, a general method for structuring contexts should be designed to spare the programmer or the user unnecessary complexities. (Sketch the class-structure for this example, the graphic above may serve as an orientation.)

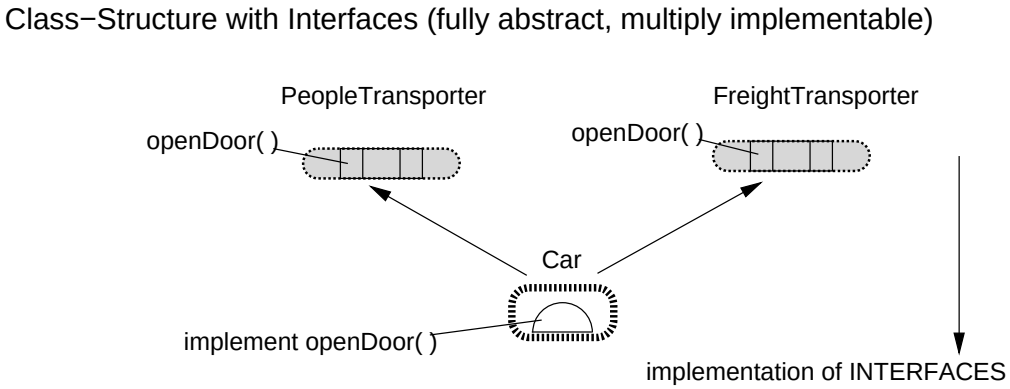
From above emerges a BASIC PROBLEM: MULTIPLE INHERITANCE BECOMES AMBIGUOUS IF TWO BASE-CLASSES HAVE METHODS OR VARIABLES OF THE SAME NAME. The question may arise about which member takes precedence when they both have the same name. Therefore, using classes that extend multiple super-classes should be considered rather careful.

again abstract methods

A tricky workaround concerning this predicament can be formulated as follows: Consider a class having only named, but unimplemented methods, also called **abstract methods**. This makes such a class automatically abstract, because it obviously cannot be sensibly instantiated. This kind of class, with no

apparent functionality, furnishes a kind of template and this seems to be the best one can get, to formulate multiple inheritance without an inherent source of contradictions or unnecessary considerations about precedence.

Example I (continued): So, using the example above, PeopleTransporter's openDoor() and FreightTransporter's openDoor() may be unimplemented. When both are extended to the Car-class, the openDoor()-method has to be implemented considering the two types of doors for loading people and for loading freight.



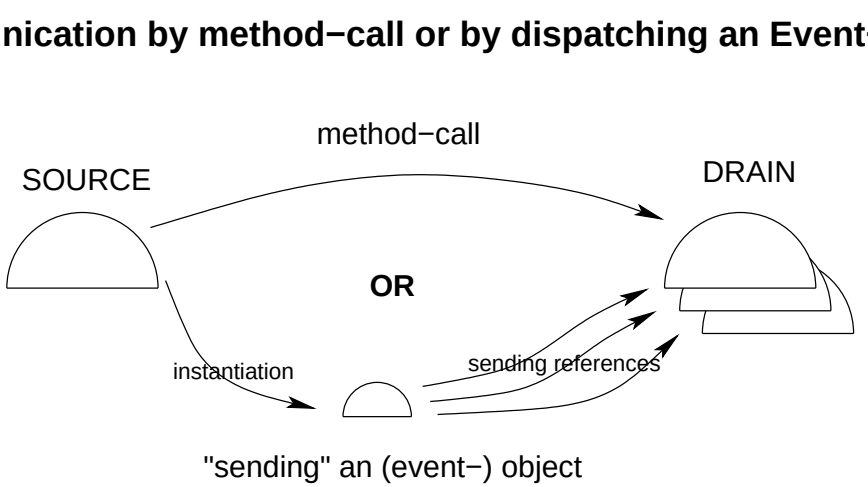
"Classes" which contain only unimplemented, so-called abstract, methods AND allow other inheriting "classes" of the same kind at their side are called "interfaces" in Java. The name "interface" signals the option to realize a kind of multiple inheritance as described above. Java[48] classes can implement *multiple* "interfaces" and thus furnish a version of multiple inheritance that proves relatively free of inherent complexities. For more see page 185.

Java's "interfaces"

Communication-Mechanisms

Communication-Mechanisms — The structuralizing of objects by classes and class-hierarchies may be considered as useful, but coexisting objects should be able to communicate to simulate real-life interaction or just to make the object-model more efficient:

Method-Activation or method-calling represents the natural means of communication between objects. This requires the objects to know each other well, but this knowledge of each other may also involve an unwanted specialization of the objects.



Message-Objects exchanged between communicating objects generally requires the objects to know very little about each other: The only knowledge, that remains necessary, is to know how to interpret the message-objects, how to send them and how to listen for them. The communication is thus realized by exchanging messages between objects, this "uncouples" objects. Then, the methods that have to

[48] The programming-language C++ allows multiple inheritance which has to cope with making inherited behavior non-ambiguous.

be called, are only listening- and dispatching methods, which, by following conventions, generally simplifies matters! For an object in a message-exchange context, a carefully-designed set of message-objects furnishes a relatively lean interface to its outside world. The rest of the implementation of the object could be changed, if necessary, without changing the behavior of the object. This isolation of the inner workings of an object exemplifies the concept called **encapsulation**. Like in proceduralization, encapsulation of object-functionalities means hiding the details of the implementation.

For more Java-related information on general messaging see pages after page 271. If the interaction between objects is realized by exchanging message-objects rather than by issuing object-specific method-calls, then this is one first step towards what is often called **component-model**. There, objects usually send and filter highly normed event-objects. For more information on the area of software-components, see page 311.

Object/Class-Model Keywords

Object/Class-Model Keywords — The following paragraph gives a small glossary on the main notions of the object/class-model:

abstract- or virtual class A CLASS THAT CANNOT BE INSTANTIATED, this means that no instances of that class can be created. Abstract- or virtual classes are collections of common states and behaviors; but these become only part of an object that is an instance of some sub-class of the abstract- or virtual class. Some of the methods may be implemented and others may be declared abstract. Example: A Car-class which lacks the specification of the transmission type is an abstract class, the brake()-method may be abstract but the accelerate()-method may already be implemented.

abstract method A named method with no functionality defined. A class which contains at least one such unimplemented method automatically cannot be instantiated; that means a class with a single abstract method becomes implicitly abstract itself! A standard use of abstract methods can be found in multiple inheritance, where these method-declarations (in Java "interfaces") are used as templates. But as could be seen with the example of the Car-class, when considering the brake()-method non-abstract, classes can be considered to be abstract without having abstract methods. A car has to have either an automatic transmission or a mechanical gearbox! (Unless it is an electrical car; this option has been ignored as the Car-model was introduced.)

class A collection of possible states and behaviors, either gathered from a set of similar objects or established from a given problem or area of interest **(class as collection of states and behavior), (class as an abstraction of a collection of objects)**. In programming, a class can ALSO be taken as a programmable type-concept **(class as a customizable type)**. A class can be seen as a template that furnishes the fields (state) and methods (behavior) of an object **(class as a template)**. A class gives the behavior and the possible states; an object of that class has the same behavior but only one fixed state and an identity. Due to the given identity (a registration-number or some other tagging-instrument) of an object, there can be several objects with the same state, instantiated from the same class.

class-extension Defining a new (sub-)class by adding other kinds of states and behaviors, respectively kinds of variables and methods to an existent class.

class-hierarchy A collection of classes, arranged according to their dependencies in form of sub- and super-classes. These kind of dependencies usually form a tree-like structure.

class-methods Methods which are associated with the class, not with an individual object. For example, methods (also called constructors) for creating instances of that class. In Java, beside constructors, so-called **static methods** are the class-methods.

class-variables Variables that are associated with the class, not with an individual object. For example the number of objects that have been instantiated during the use of the class can be counted by using a class-variable. In Java, class-variables are also called **static variables**.

container-object An object which contains a list of references pointing to other objects. Container-objects typically implement buffers, stacks or little non-persistent databases. Java supplies container-objects like instances of the class `java.util.Vector`. CAUTION: Though Java array-types can index objects, array-types are something different than class-types (See the syntax-diagram on page 143, and the introduction on page 234.). Only from Java class-types can be instantiated objects!

encapsulation Hiding, often complex, self-contained solutions of problems and giving them a much easier programmer- or user-interface. This allows a relative ease of use without having to grasp inner complexities. Proceduralization offers a means of encapsulation by needing only to know the method's signature and return-value to use it. Object-Orientation can be made to work similarly on the level of the entire object. Encapsulated code may document its functionality in a so-called (Application) Programming Interface (API). Especially in the context of networking, the effects of simplifying things, by encapsulation of code, are circumscribed by the notion "transparency".

inheritance All sub-classes (or class-extensions) receive the methods and the variables furnished by their super-classes (sometimes also called parent-classes). INHERITANCE DESCRIBES THE SAME AS CLASS-EXTENSION; but inheritance focuses on the sub-classes whereas extension stresses its starting-point being the super- or parent-classes.

instance of a class An object derived or instantiated from that class. The class represents a template for state and behavior; an object bears a concrete state, a concrete behavior and and identity(-tag). Also refer to the item "object" below.

instance-method A method associated with an individual object (in contrast to a class-method which is associated with the entire class). The nature of that association, for example, makes the method change one of the object's variables. Instance-methods are taken to represent the behavior of their object.

instance-variable Instance-variables are associated with an individual object (in contrast to a class-variable which is associated with the entire class). Instance-variables describe the state of their object.

method A method can be called (for example by other objects) to alter the object's state or to send messages to other objects. In programming practice, a method (also called function, procedure, subroutine) usually represents a kind of named piece of code associated with an object or class. Usually methods are the means by which another object can change the object's variables (alter the object's state).

message A signal from a source-object to a target-object, usually requesting a method-invocation of the target-object. Sometimes, simply calling the method of another object and transferring information within the method's arguments is considered as sending a message! But, often messages are objects themselves, then they bear a standardized structure, which is known to all communicating instances. Then, the method-calls used for communication of objects become highly standardized. Usually a message-object contains the name of the sending object, the name of the receiving object, the name of the method to invoke and some variables that are used as the invoked method's arguments. This concept has been developed further beginning with page 271 and on page 571.

multiple inheritance The case where a class inherits state and behavior of multiple super-classes (In Java, multiple super-interfaces). To avoid ambiguities of the super-classes with same variable- or method-names, only abstract methods and unassigned variables should be used. The Java `interface`-construct furnishes this restriction.

object An entity with a **state**, a **behavior** and an **identity**. In programming, an object is given as an entity of variables and methods together with an identity. This generality often reduces to an object being seen as a variable that has been instantiated from a class-definition, where the class is taken to be a sort of generalized type-definition.

overloaded methods Methods are identified by their object's name, their own name and their parameter-list. So even in their own class, different methods can have the same name but a differing parameter-list to remain valid. Such methods are called overloaded. Remember the section beginning with page 124.

overridden methods Overridden methods always occur in the context of a class-extension: A class-extension may define a method with the same name and the same list of parameters as a method given by the super-class. The newly defined method becomes the default method of any objects instantiated from this extension or any further extensions. An analogon on the level of variables presents itself: Two variables with the same name but different overlapping scopes produce the following effect: The variable with the limited scope overshadows the other. (Why are there no overridden variables? Because overriding variables would amount to redefining the type of the variable-name, nothing more.)

Pragmatics: Constructing class-hierarchies, to describe a problem efficiently, usually turns out to be demanding and even may need redesign efforts. But there are some simple rules and conventions that may ease the task: When describing the real-life problem, using the everyday language,

---

nouns indicate an object or a class to be used in the object-oriented analysis,

verbs indicates a method's functionality and the formulation

is a *name* indicates something to be an object or a sub-class.

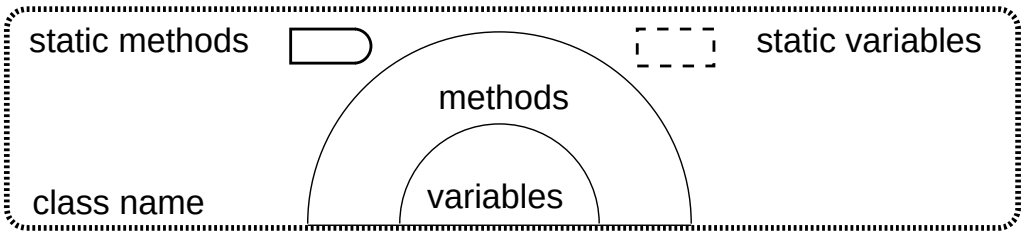
Object-oriented analysis of problems can happen without using a programming-language. Object analysis represents a problem-oriented method of program-design and therefore usually happens *before* constructing source-code. But object-oriented design usually addresses itself to EXTENSIVE real-life problems with less dominant algorithmic complexities. The wealth of approaches and methodologies of object-oriented-analysis and -design is described in books or related Internet-sites.

Concept: Object/Class-Model within Programming-Languages

Concept: Object/Class-Model within Programming-Languages — In many programming-languages, aggregates of variables are called `record`s or `struct`s. Aggregates of command-sequences and variables are described by functions, methods or subroutines. Here aggregates of variables AND methods are called **class**es.

```
class name
{
  variables;
  methods;   // usually for manipulation of the variables
}
```

In that context, remember the object-model on page 131 and the graphic of the general class-model on page 134. The general class-model, introducing state and behavior, changes in the context of a programming language to a class-model given by an aggregate of methods and variables:



Classes in programming-languages can be given another variant of definition: Classes just furnish a way for the user to aggregate data (variables) and methods (functions, subroutines). Object-oriented programming-languages also furnish ways to introduce static class-members and ways to derive individual objects from a given class.

For practical programming purposes the notion of a class can be used

- to introduce programmer-defined types,
- to use a class as a self-contained program and
- to use a class-structure as an index for reference when re-using software:

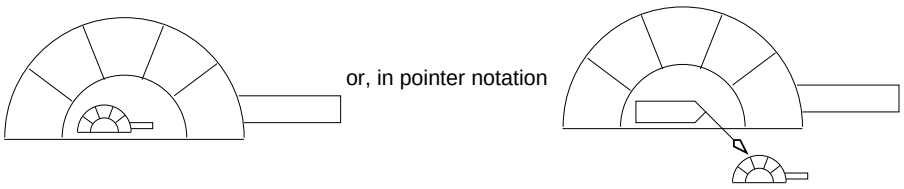
Classes as Custom-Programmed Types **Classes can be used to define programmer-specified types** within the programming-language. These programmer-defined types give templates for the data and furnish the methods to manipulate the data. The central idea aims at presenting the data and methods for the data-manipulation as an entity. Thus, in other contexts, methods for data-manipulation do not have to be re-written. To the programmer, both, the data and the methods, are related intrinsically; just like any integer type furnishes mathematical functions like addition, subtraction, multiplication and division for its manipulation:

```
class Integer // this is a concept-class and does not work in Java
{
  int i;
  addTo(int j){return i+j;}; multiplyWith(int j){return i*j;}; // etc...
}
```

If the standard types of the given programming-language can be supplemented by types defined by classes, then variables of class-type even can be declared inside classes:

```
class ClassName1
{ ClassName2 identifierOfObject; // declaration of variable of class-type
}
```

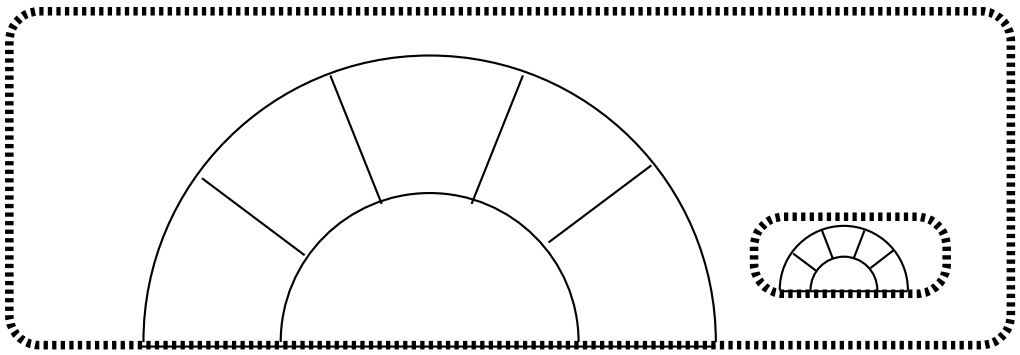
Thus a variable of class-type — an object itself — can be seen as part of the state of an object:



So the state of an object is given by objects (pre-programmed or programmed by the application-programmer) or by variables of primitive-type. Objects of standard classes given by the programming-language (Java: `String`) or variables of primitive-types (Java: `int`) finish the recursion indicated by the diagram above (objects in objects in objects ...). For example, in the concept-class above, named Integer, the `int`-variable of primitive-type, given by the programming-language, defines the state. A practical example of this idea is given on page 142, where a newly-defined type (VarInteger) is used for declaring and instantiating a variable (there the variable can be seen as an equivalent to an object).

Instead of containing only a variable-declaration, a class even may contain a class-definition, a so-called inner class, giving a kind of local type:

```
class ClassName1
{ class ClassName2 // class-definition
  { ...
  }
  ...
  ClassName2 name; // declaration of a variable of that local type (inner class)
}
```



Inner classes are described further beginning with page 157.

Class as a Program A class can be a flexible type, or **if a class is furnished with a `main()`-method, then a class can be a program**:

```
class Program // this is a concept program and does not work in Java
{
  main(){ statements }
}
```

If the source-code inside the `main()`-method references other classes or creates and references other objects, then the class with the `main()`-method does represent the central piece of a program! Conventionally, classes with a `main()`-method enable an end-user to issue some command on the level of the operating-system, eventually triggering a call of the `main()`-method (the `java`-command entered into a terminal-window of the native operating-system triggers a call of the `main()`-method). This peculiarity gives classes with a `main()`-method all the characteristics of a regular program. Thus, in most object-oriented programming-languages, the class-notion becomes more than simply a flexible way to define new types.

Class-Structure as Index to Software-Library But the perspective of the class/object-model gives another important functionality to object-oriented programming-languages: As can be seen in the graphic on page 135, class-extension lets the code be positioned in a sort of programmer created hierarchical index-structure for easier identification. Successive class-extensions and the ensuing class-hierarchy can be used to **realize a structure of the involved objects**. Since the super-classes may be used multiply (remember the car example above), that kind of structuring also makes it possible to easier re-use code.

Java: Classes as Programmer-Definable Types and Programs

Java: Classes as Programmer-Definable Types and Programs — Java classes can be more than templates for objects. A Java class with a `main()`-method constitutes the center of a program. The fully-functional example below defines two Java-classes: The first class introduces a programmer-defined type and the second class acts as a program due to its `main()`-method. (The use of the modifiers `public`, `static` and `void` is described elsewhere: `public` on page 175, `static` on page 147 and `void` beginning with page 117)

```
// a Java class: a programmer-defined type or a program
class VarInteger                                  // programmer-defined type
{ int i;                                             // state: instance-variable
  VarInteger( int x ){ i = x; };                     // class-method: constructor
  public void set( int x ){ i = x; }                 // behavior: instance-methods
  public int get(){ return i; }
  public void increment(){ i = ++i; }
  public void print(){ System.out.println( i ); };
}
public class Program                               // program or application
{ public static void main( String unused[] )       // class-method: static main()
  { VarInteger theObject = new VarInteger( 0 );
    String theOtherObject;
    theOtherObject = new String( "Hello" );
    theObject.print();
    System.out.println( theOtherObject );
} }
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% javac Program.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% java Program
0
Hello
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro%
*/
```

Another example of a programmer-defined type is shown in the source-code on page 76, where the Pair-class has been introduced. These two approaches for using Java classes (program and custom-defined type) are the most typical and elementary ones. – A class may also be used to just furnish a collection of methods; as does the `java.lang.Math`-class which just offers a range of mathematical functions to the Java programmer (for more see pages 94 and 257).

A class with a `main()`-method remains instantiable!

An instantiation of a class with a `main()`-method has no special meaning with respect to the `main()`-method and its program-character: The `main()`-method just remains one of the static methods, except that it can be activated by an external event (the issuing of the "`java`"-command). A first example of such a case is given on page 159.

Java: Reference-Types and their Instantiation with the new-Keyword

Java: Reference-Types and their Instantiation with the new-Keyword — Remember the introduction of primitive-types on page 83. Variables of primitive-type have their identifier fixed to an address in the computer's Random Access Memory (RAM). These types, with identifiers being an unchangeable reference,

 ISBN 3-00-009429-6

thus being glued to the RAM-address and exhibiting a fixed behavior (addition, division with number types), are called atomic- or primitive-types. Primitive-types constitute an integral part of the Java language and prescribe the data-format (int: 4bytes) and the possible manipulations (Addition, Subtraction, Multiplication, Division). With variables of primitive-type assignment happens by-value, this means that an assignment copies the value to the variable-name's part of the RAM. Usually a computer processes variables of primitive-types faster than Java objects of class-type.

remember: assignment by-value

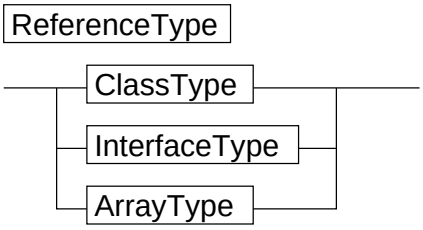
Java offers two kinds of basic non-primitive-types, the so-called reference-types :

class-type Currently discussed. A variable of that type is instantiated by prefixing a constructor-call with a new-keyword; for more see page 145.

array-type Realizes a simple form of aggregation of values or object-references. Variables of class-type as well as variables of primitive-type always can be aggregated as an array. Variables of array-type are recognizable by the brackets [ ] used in declaration and referencing, for more see page 234.

An example of introducing a class-type has been given in the preceding program on page 142. There has been defined a class(-type) "VarInteger" and from this class has been instantiated an object (referenced by an identifier "theObject").

The interface-type gives a variant of the class-type for cases of multiple-inheritance, for more see page 185.



ReferenceType := *ClassOrInterfaceType* | *ArrayType*

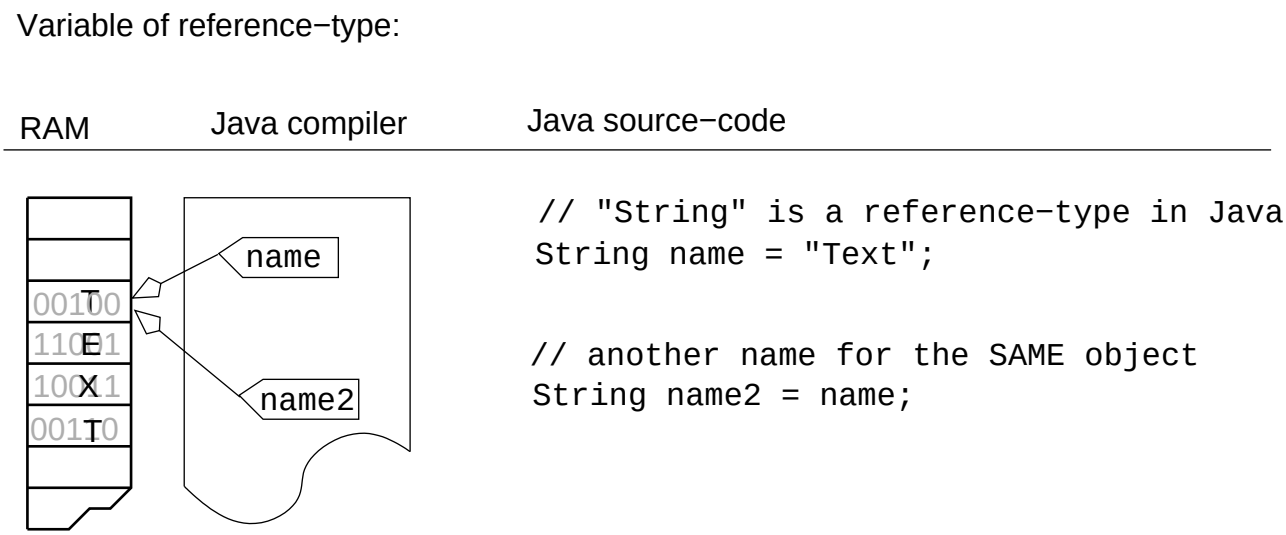
ClassOrInterfaceType := *ClassType* | *InterfaceType*

ClassType := *TypeName*

InterfaceType := *TypeName* — ClassType 146, InterfaceType 187, ArrayType 234

(ClassOrInterfaceType is used in ClassInstanceCreationExpression on page 151)

Like variables of primitive-type, variables of reference-type have identifiers that point to an address in RAM-space. But, the RAM-space-address of variables of reference-type is changed by assignment (=), NOT the value stored there, as it is the case with variables of primitive-type. IMPORTANT TO NOTE: IDENTIFIERS OF VARIABLES OF REFERENCE-TYPE CAN BE SEEN AS ALIASES AND THEIR POINTING-TARGET CAN BE CHANGED BY ASSIGNMENT:

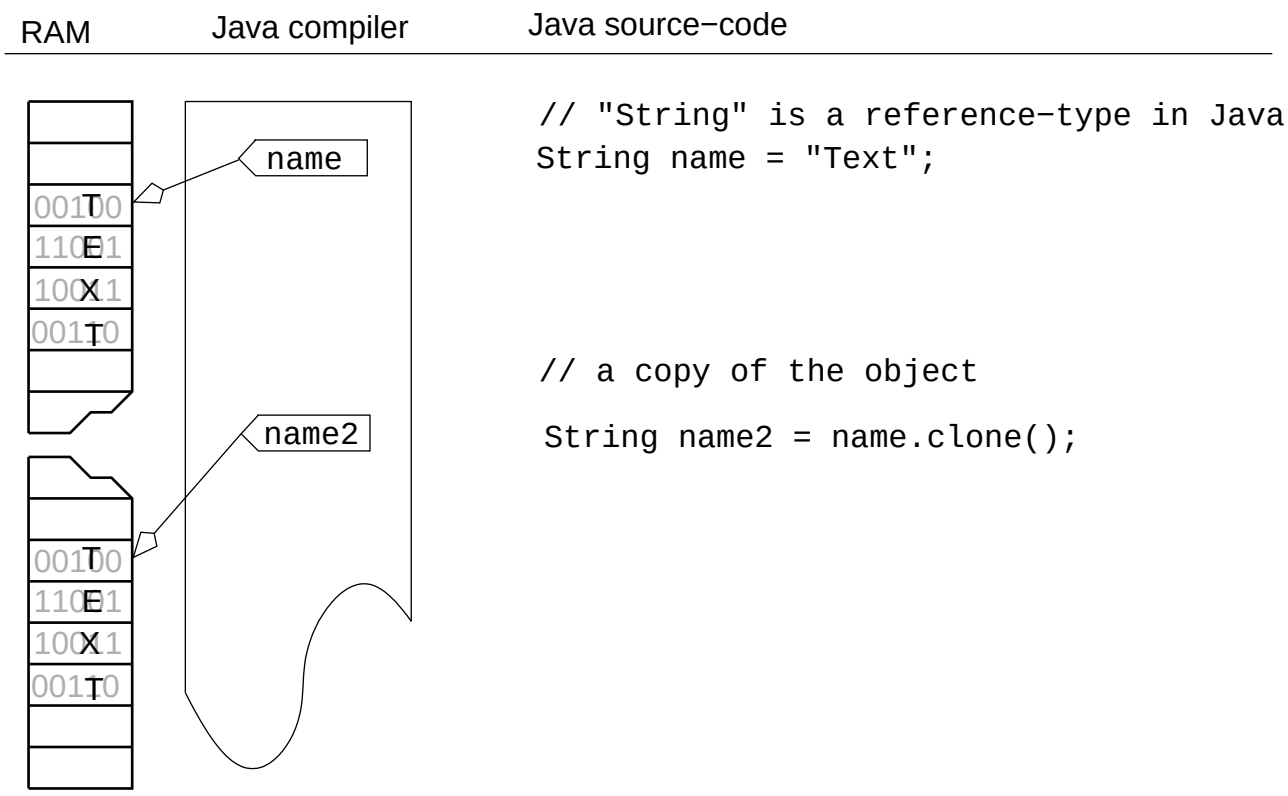


A reference is a container for a RAM-address which can be changed by the programmer by assignment. That's an important difference to Java variables of primitive-type where the references are fixed to an address of the RAM-space. And the act of assigning variables of primitive-type to each other with the equal sign (=) means copying. (see the graphics on page 83) For variables of reference-type assignment

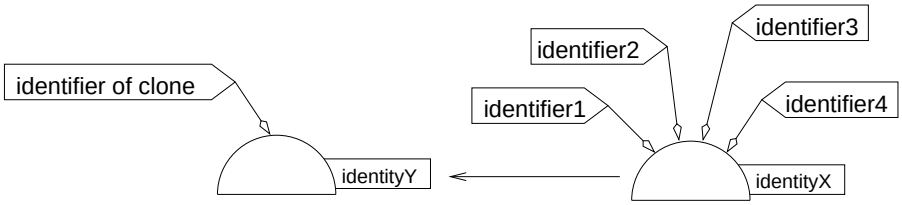
assignment by-reference

(=) means: If objects and associated names are given, reassignment of names by using the equal sign (=) only exchanges references, not the entire object, as indicated within the graphic above.
Copying of an object, a variable of reference-type, is done with a `clone()`-method as described below:

Copying a variable of reference−type with the clone()−method:

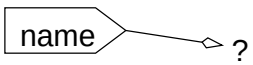


The `clone()`-method PRODUCES A NEW OBJECT that contains the same data AND RETURNS A REFERENCE TO THAT NEW OBJECT! (It does NOT "return the new object" itself!)
This statement is true in all generality: Java-methods with return-types of the primitive kind return the variable-value itself. But if the RETURN-TYPE OF ANY JAVA METHOD IS OF REFERENCE-TYPE, then, like the `clone()`-method, the method returns only a reference to the object!
The name of every reference-type could be suffixed with the name "Pointer" (StringPointer instead of `String`), to reflect the difference to the named value of data that has a primitive-type. Data of primitive-type has only one name, assignment with the "="-operator copies that value. Whereas data of reference-type can have multiple names, meaning that multiple pointers are referencing the same data; here assignment with the "="-operator just makes the pointers equal. The assignment-operator, used with identifiers of reference-type, does not copy the data!



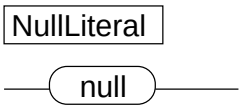

```
identifierOfClone = identifier1.clone();
```

Any name of reference-type can be assigned the literal **`null`**, which lets the name reference nothing.




```
name = null;
```

```
variableName = null; // an identifier of reference-type not yet assigned
```



NullLiteral := null

Instances of reference-types (to be more precise, instances of non-abstract classes and arrays) are created by prefixing so-called **constructor-method**s with the **new**-keyword or by calling so-called **factory-method**s, described later on page 151. (The `clone()`-method is a factory-method, because calling this method creates a new object and this method returns a reference to that object.)

constructors and factory-methods

```
ClassName variableName = new ClassName();
// Class-methods with the class-name are called constructors.
```

The new-keyword in conjunction with a constructor allocates the memory, creates and initializes the object. Then the variable-name (identifier) represents a reference to the object; the variable-name does not represent the object itself, contrary to names of variables with primitive-type!

Below the processes of declaration, memory allocation and instantiation are compared for variables of primitive-type and variables of reference-type:

variables of type-		Declaration	Allocation	Instantiation	
-primitive		*PrimitiveType name*	=	*value;*	
-reference		***ClassType name***	**= new**	***ClassType*();**	**← a constructor-call**
-reference (array)		*PrimitiveType* [] *name*	= new	*PrimitiveType*[];	(arrays, see page 234)
-reference (array)		*ClassType* [] *name*	= new	*ClassType*[];	

Java's **String exception**

In Java, the `String`-class has been allowed an exception to the general class-usage; a Java `String`-object can be instantiated like a variable of primitive-type:

```
String s = "text";
```

Only in this special case of the `String`-class, Java allows an instantiation like those used with variables of primitive-type. Actually the `String`-class also allows every `String`-object to be instantiated using the new-keyword prefixing the `String`-constructor:

```
String s = new String("text");
```

This standard way of instantiating objects of class-type has been used multiply in the previous programs.

Java: Classes

Java: Classes — The following sections use the class-concept together with the idea of a reference-type to describe the Java-specific realization of programmer-defined classes. This includes, for example, inner classes, anonymous classes and anonymous objects, supplemented by scoping- or accessibility considerations.
In the previous chapter, the description of proceduralization by using methods had more implications than could be forseen by the pure concept[49], the same is true with the implementation of the class-concept by the Java programming-language.

Java: Class-Declaration

Class-declaration may also be called class-definition or class-implementation. The standard Java-form can be seen below:

```
class ClassName { classMembers }
```

[49] Starting with considering named blocks of code, ending up with local variables making possible overloading, recursion and callback.