

Introduction to Programming

Dmytro Chibisov, Sergey Pankratov, Christoph Zenger

Programs, algorithms and data structures

Programming may be loosely described as writing a finite sequence of prescriptions for computer systems. This set of prescriptions – a program – must be so far written in artificial languages that can be translated into a code starting desirable processes inside a computer. In the computer science, the program written in a special programming language is nicknamed a source code, whereas its interior representation is called a machine code. A source code is expected to be readable by people and saved as a text file. Nevertheless, it should be so precise as to exclude any ambiguities when translated into a machine code.

Programs are intended to solve specific problems, and to this end they are constructed by a certain plan, according to which they must work. Such plan is known as an algorithm. In distinction to programs that are platform-dependent, algorithms can be invariant under the change of platforms, or “portable” as computer professionals used to say. In other words, the program may be thought of as a formulation of an algorithm in a certain language.

Algorithms work not only as plans for software design – they also operate on data. It means that data is transformed in a given way prescribed by the algorithm. In order to ensure that such a transformation would work efficiently, the data must be organized, stored and handled within certain structures called “data structures”. Algorithms and data structures are considered two main pillars of computer science.

There are several fundamental - generic - data structures that are used throughout all programming techniques, e.g. arrays, lists, trees, fields, queues, stacks, etc. Most algorithms are associated with specific data structures: thus, sorting algorithms typically work on arrays; lists can be ordered; a maximum element can be searched for in a list; one can insert an element into a queue or delete a node from a tree, etc. Such associations are reflected in the appropriate program structuring.

There exist thousands of programming languages, which can be roughly subdivided into two major groups:

- general-purpose languages, such as FORTRAN, Pascal, C, C++, C#, Java, LISP, Prolog,...
- special-purpose languages, such as TeX, LaTeX, HTML, XML, SQL,...

As far as computers are concerned, they understand only very simple instructions represented by binary numbers. This machine language is hardly readable by the normal humans.

Object technology

The contemporary computer science and its spin-off – the software industry – seem to lean towards object-oriented programming (OOP). OOP has the ability to represent objects from the real world – in distinction to more traditional “procedural” or imperative programming, which tends to operate primarily with procedures, which are less naturally adaptive pieces of

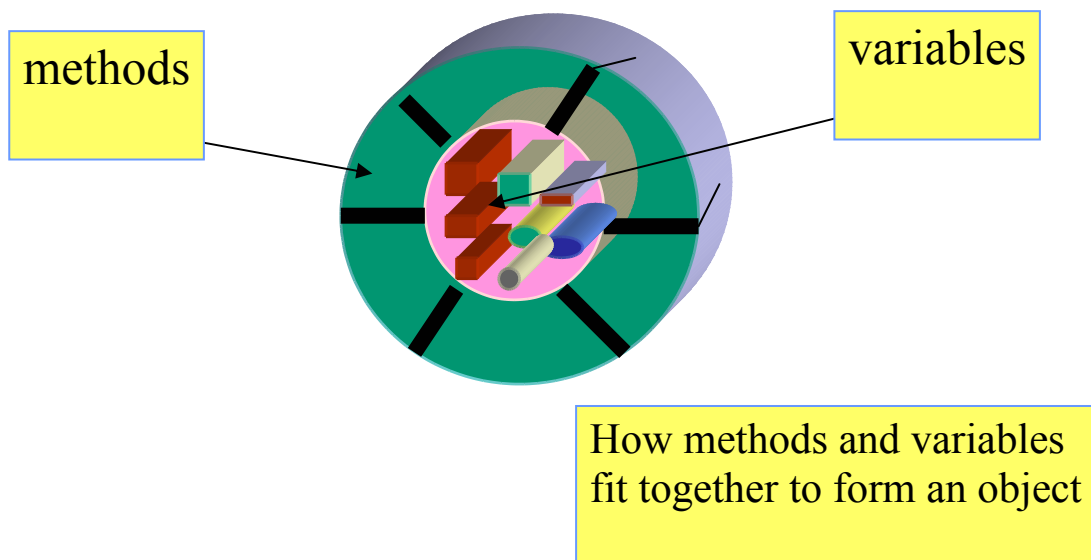
software. Sometimes this difference is formulated as “objects first” in OOP versus “commands first” in procedural programming. Objects are elementary entities that can be found in a certain state, the latter being characterized by the values of object parameters.

Object orientation is considered to be the modern and obvious choice primarily because of its modeling capability. Figuratively speaking, real-world objects - employees, bank accounts, parts of a machine, etc. can be represented “inside a computer”. Data in OOP is also represented by objects. Objects have two components: fields and methods. Fields tell us what an object is, whereas methods tell us what an object does. Fields and methods reflect an object’s real-world features. For instance, an object must have a clearly defined identity, e.g. a user-defined name, and an object’s state is defined by fields. The object’s behavior is specified through a set of methods, which are, in effect, operations acting on object’s private data. Methods may also invoke interactions with other objects.

What is an object?

An object is a software package that contains a collection of data and procedures. It is these latter that are mostly called methods, to distinguish them from traditional programming procedures. Data elements are called variables – their values can vary with time.

When a program is run, messages are passed between objects. When an object receives a message, it responds according to defined methods.



OOP abstractions

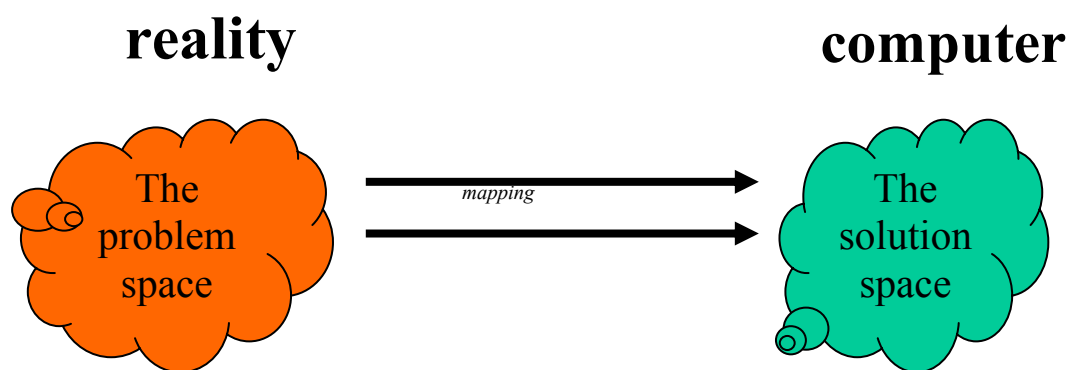
An idealistic approach to OOP:

Elements both of the problem and of the solution spaces are referred to as “objects”. The idea is to describe the problem in terms of the problem itself, not in terms of the computer used to obtain the solution.

Actually, what is a programmer's ultimate task? It may be formulated as establishing the correspondence between the problem to be solved and its computer representation. Programming languages are just means to implement this mapping.

First languages obviously tended to imitate the computer (Assembly, BASIC), while OOP claims to represent elements of the problem space.

The concept of software objects stemmed from the need to model real-world phenomena on a computer.



A remark: *The whole hypertrophied programming industry mass producing software, which is difficult to write and expensive to support, is in fact a consequence of multiplicity of means to map problems to their computer solutions.*

Origin of object-oriented concepts

Basic concepts of OOP were introduced in the Simula programming language developed in Norway in 1960s. Simula (= Simulation Language) was created to support computer modeling of real-world processes, mostly in physical engineering. The authors, O.Dahl and K.Nygaard, wanted to construct models of complex physical processes containing many thousand components. More specifically, systems to be modeled were defined as a set of independent objects working for a common purpose, with the system being able to check the state of its elements. Very soon it became obvious that to model such a complex system modular programming was essential. In Simula, modules were based not on procedures as in conventional languages, but on physical objects to be modeled. A very important concept – that of a class – was introduced in the language. One could declare a class, create objects of this class, assign names to these objects and, what is probably the most important, organize classes in hierarchies.

This choice appeared quite natural to breakdown the problem: each object possessed a range of behavior and had to contain information on its status. The interaction between objects produced simulation. The idea of originators of OOP could be formulated as a rhetoric question: *Why should we look for an artificial way to structure data and procedures, when the real world has already organized them for us?*

Main features of object technology

An industry-standard definition of object-oriented technology may be summarized in three properties:

1. Objects that ensure encapsulation of procedures and data
2. Classes that implement inheritance within hierarchy
3. Messages that support polymorphism across objects

To summarize the concept of an object in simple terms, one may list the following features:

1. Object is an exemplar of a class
2. An object has an identity, e.g. a user-defined name
3. The state of an object is defined through object (instance) variables
4. The object's behavior is specified through a set of methods – operations that act on private data and call on operations of other objects
5. An object interacts with other objects by sending/receiving messages. A message contains the name assigned to the method associated with object. This is probably the most efficient type of relationship used both in biological and physical systems.

Data encapsulation

Encapsulation is understood as packaging together all related data and procedures. Each object hides its internal structure from the rest of the system. More precisely, an object behaves in such a way as to hide the implementation – since an object may choose to expose some of its data. Nevertheless, internal methods remain hidden.

The key to object encapsulation is the message interface – it surrounds the object and serves as the contact front for incoming messages. Message interface protects an object's internal contents (variables and internal methods) from being corrupted by other objects. This is quite convenient not only from theoretical, but also from the practical viewpoint, since an object, once fully tested, is guaranteed to work forever – there is no way for any other object to access data or methods hidden behind the object's interface. Encapsulation limits changes to a single object, hence there is no need to modify subroutines, to make synchronization, etc. As a result, objects do not disturb each other's data structures.

One can also describe encapsulation in more general terms. All programs consist of two main components: code operators and data. Usually, terms “code” and “program” are used interchangeably, but, strictly speaking, the term “code” is narrower: it denotes the part of a program that carries out actions (methods), while data is the information that modifies these actions. Encapsulation is the mechanism that merges code and actions, which the code operates with, and safeguards the both components – code and actions - from inadvisable external influence and misuse.

Classes

Objects having similar behavior are unified into a class. In other words, a class is a description of all properties of all objects of the same type. In OOP, specifically in Java, a class is a software template defining variables and methods for a particular kind of object. As soon as a class is specified (declared), an arbitrary number of objects belonging to this class can be created. Such object exemplars are called instances of the class, thus fields characterizing such objects are just instance variables. One can see from this construction that instances may contain only their particular values of variables. Using common OOP terminology, one can say that properties of an object reflect its structure (static properties) or its behavior (dynamic properties), static properties being described by instance variables while dynamic ones are described by methods.

Thus, objects and classes are the key concepts of OOP. One may ask why it was useful or even necessary to introduce the concept of a class, once objects have already been present. The answer is that modeling rarely involves a single sample of an object, and usually one needs to consider many objects of the same type. So, it would be very inefficient to redefine the same methods for each occurrence of an object. A class manifesting all essential features of similar objects seems to be a fair solution.

Classes bring order

- ❖ Objects interacting through messages form the core of object technology, but it is the concept of classes that makes the object approach effective in modeling complex systems.
- ❖ Classes are elementary building blocks of object-oriented (e.g. Java) programs: the description of data and related methods (behavior) lies within classes. For instance, every Java application consists of at least one class.
- ❖ Classes define how objects will behave - for example, dogs bark, cats meow, frogs quack – and what properties an object will contain when it is constructed (instantiated).

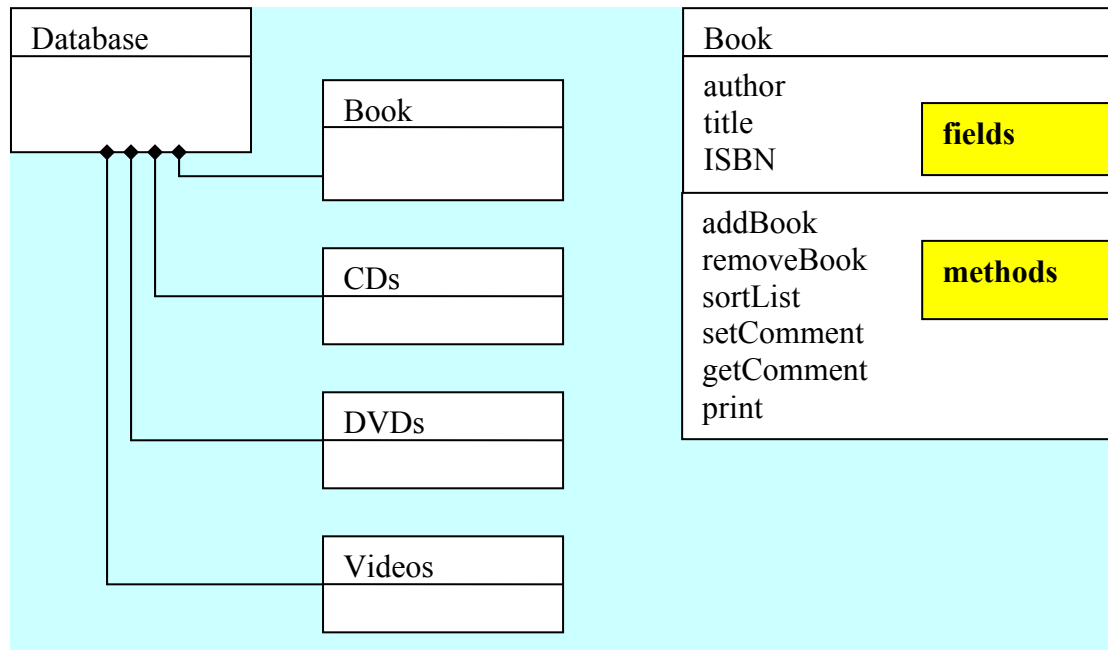
Example: one can create three objects of the latter classes, named Dog, Cat, and Frog. To implement the behaviors, one can create a method for each of these objects. Such method may represent the sound produced by each animal. This method can be called *speak* or *sayHello*.

- ❖ Classes make visible even very complex hierarchies (like in biological classification). In this sense, classes bring order and help the developer to structure a program.

Example: how would we ensure the functionality of a computer catalogue of all the books, CDs, DVDs and videotapes, for example those that we own? The functionality we would like to provide should at least include the following:

- allow us to enter information about books, CDs, DVDs and videos
- store this information so that we could use it at any time
- envisage a search function allowing us to find each book by a certain author, title, ISBN, etc.; to find each musical CD by the album title, name of the band or the singer, number of tracks, and so on
- permit us to add/remove information on any book, CD, DVD, videotape

To implement the respective application (a database), we must first decide what classes to use to model the situation. In this case the most direct approach would be to create a class `Book` to represent book objects; a class `CD` to represent CDs, a class `DVD` to represent DVD objects, and `Video` to represent videotapes. Each of these classes holds fields and methods. Once we have defined these classes, we can create as many book, CD, DVD, and video objects as we want. Besides, we must declare a database class that would contain the four above collections, providing the functionality to store lists of all book, CD, DVD, and video objects. All these lists can be printed out (e.g. to the terminal)



A class and its instances

We have seen that declaration of a new class provides a template or a blueprint that describes the data contained within, and the behavior of objects instantiated according to the new type. The term “instantiate”, which means to produce an instance, refers to creating an object that belongs to a particular class. For example, *Munich (München)* is an instance of the class *City (Stadt)*, and in an object-oriented code we must create this specific object by giving it a name. All objects of a given class are supposed to be identical in form and behavior, but may contain different data in their fields (variables). In strongly object-oriented paradigm there are no other objects than class instances: any object is an instance of some class.

What is the main distinction between classes and objects from a programmer’s viewpoint? Classes provide a **static** description of a set of possible objects – exemplars or instances of the class, whereas objects are **run-time** elements created during a system’s execution.

Class vs. instance

A brief summary: In object oriented languages, it is possible to define:

- *instance* variables and *instance* methods.
- **static** or *class* variables and **static** or *class* methods.

Instance variables and *instance* methods can only be accessed through an object of the class.

Class variables and *class* methods can be accessed without first instantiating an object.

The class name alone is sufficient for accessing *class* variables and *class* methods by joining the name of the class with the name of the variable or method (using dot, e.g. `Car.color`).

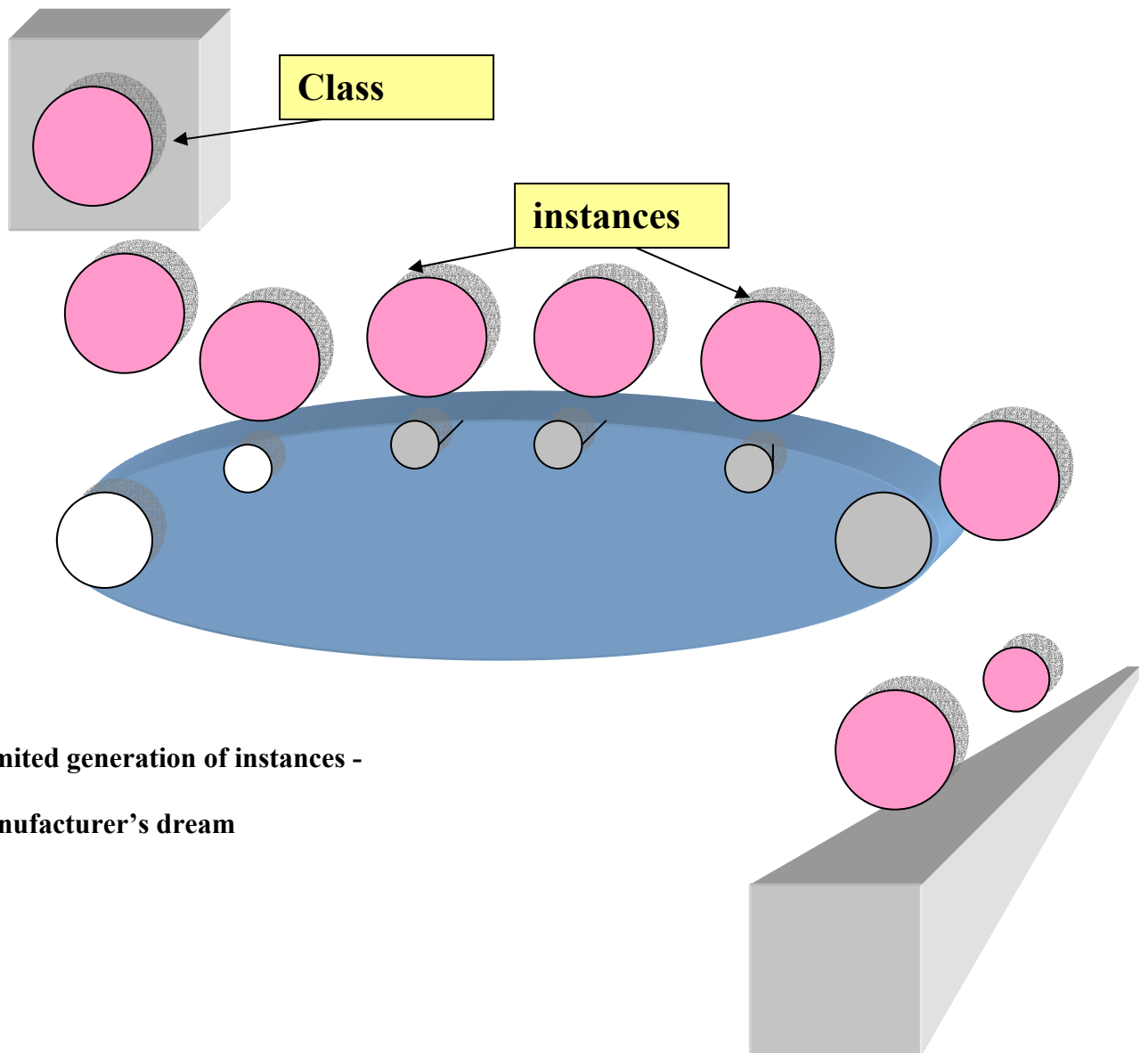
Multiple instances

When you have a class, you can create an arbitrary number of objects – instances of this class. For example, from the class **Car** we can create **car_1**, **car_2**, ... Each of the newly created objects may have its own position, color, size, etc. We can change any of these attributes of an object by calling a method on this particular object – this does not affect others. Here, in passing, we may recall that a method can have any number of parameters, e.g. the **changeSize** method for the **Car** class can have the following set of parameters (the signature):

```
void changeSize (int newWidth, int newLength, int newHeight)
```

The set of values of all attributes defining an object determine the object's state. Certain methods may change the state of the object if being invoked. For instance, `moveForward` method changes the *y*-coordinate attribute (we remember that such varying attributes are usually called fields). Objects generated from the same class all have the same set of fields.

Once a class has been defined, all instances of that class can be guaranteed to have an identical form and to function exactly as prescribed by the class. Objects can be produced in any quantity and can be created nearly instantaneously. The process of instantly generating objects, accompanied by a negligible production costs, may seem a materialization of a manufacturer's (e.g. Henry Ford) archetypal dream.

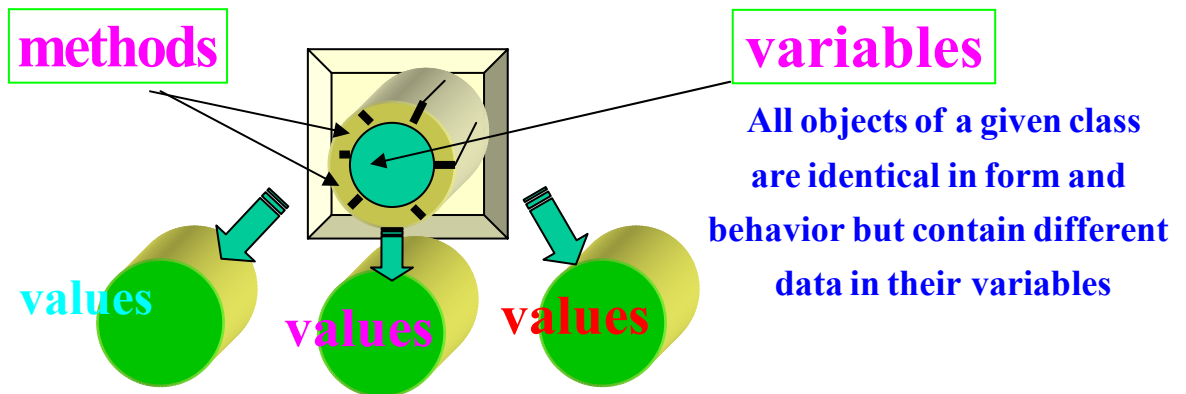


**Unlimited generation of instances -
a manufacturer's dream**

However, although a reliable mass production becomes no more a problem, this is far from being satisfactory in software production (called nowadays software engineering). The problem is that the goals in production and in engineering are fundamentally different, and this distinction leads to important implications. Let us recall our starting question: what do we need the software for? The answer is: to model the situations in the real life. Software components should emulate business processes or solve mathematical problems (which is also an approach to modeling), and to this end the software systems must be really adaptive. In this sense, adaptivity is much more important than productivity. Adaptive objects representing products, customers, payments, shipments, matrices, tables, etc., if properly engineered, can, for example, sustain many years of service being only slightly modified as requirements evolve. If we only knew how to clone objects from a class pattern, the modeling value of such approach would be rather low, because despite the great manufacturing capability we would have to model each new situation from scratch. Thus, there must be methods allowing us to adaptively modify objects as building blocks for modeling software.

State and Behavior

We have already mentioned that an object is said to have *state* and *behavior*. At any instant in time, the *state* of an object is determined by the values stored in its *variables* and its behavior is determined by its *methods*



A *method* is an OOP term for the fragment of code that describes some behavior or action related to an object. For example, a class used to build a computer model of a car might include the method `openDoor`.

Hierarchy and inheritance

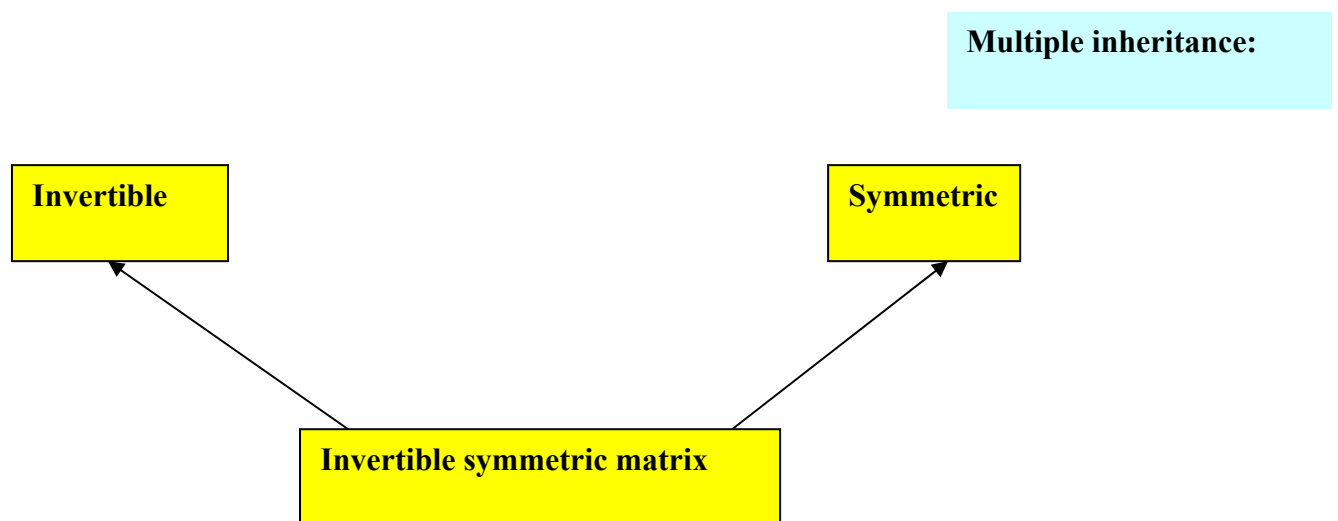
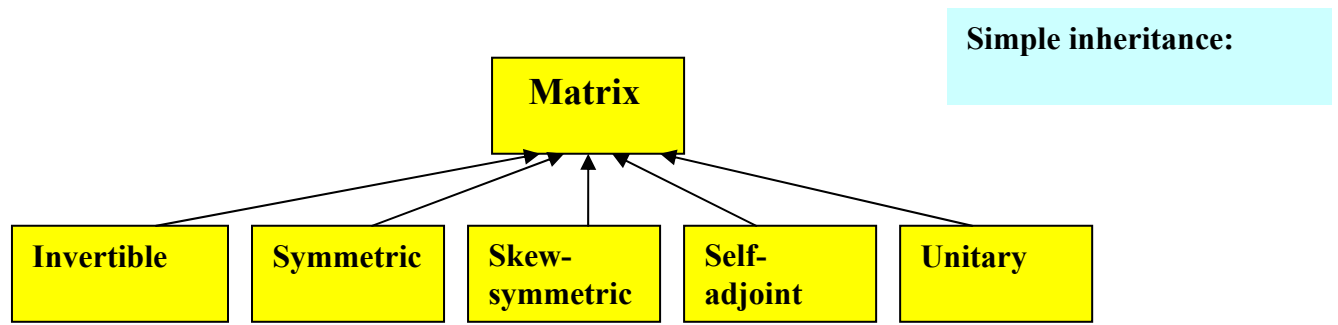
Classes can be defined in terms each other. More specifically, the class of the object from which another object is deriving its properties is called **superclass**, i.e. a more general class. On the contrary, the derived class is called a **subclass**. Superclass is the base class, while subclasses are produced by refinement, for example a “Dog” is a more specialized notion than “Animal” or a “Car” is more precisely specified than a “Vehicle”. The ability to derive a class properties from those of another class is called **inheritance**.

Thus a **tree structure** representing the inheritance relationships in a set of classes arise. A class hierarchy has a single top node and may have any number of levels

Due to inheritance, classes can use methods and variables defined for all the above classes on their higher branch of hierarchy. Simply by declaring Class B to be a special case of Class A, one automatically gives it an access to everything Class A can do (e.g. Car is a special case of Vehicle inheriting a lot from this superclass).

Later we shall see how inheritance works in Java programming. Now we can only mention that different object oriented languages support different inheritance types. For instance, Java

and another object oriented language extensively used in numerical computations – Smalltalk - allow only simple inheritance, namely from a single ancestor class. (Smalltalk was historically a Java predecessor, which effectively means that it did not undergo the strong influence of the C language.). Still more popular in object oriented numerical simulations C++ language supports multiple inheritance, which allows the class to inherit properties from more than one ancestor class. Sometimes, multiple inheritance is quite convenient in dealing with mathematical objects, but it tends to increase complexity. For example, an invertible symmetric matrix is at the same time invertible and symmetric, and this is a typical multiple inheritance



Advantages of inheritance in programming

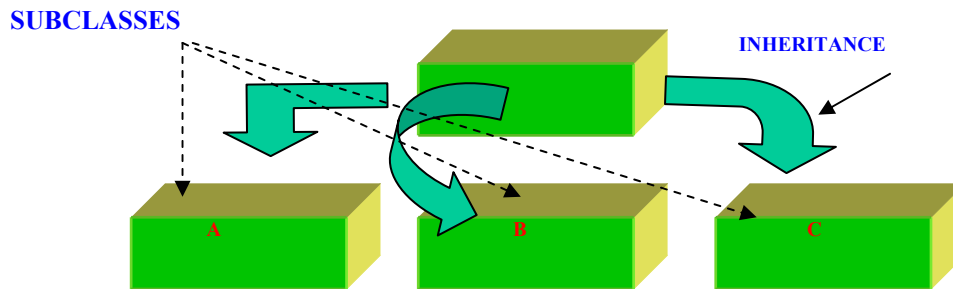
- ❖ **Code duplication can be avoided:** there is no need to write identical pieces of code several times
- ❖ **Existing code can be reused:** if a class similar to the one we need already exists, one can make a subclass of it and reuse a portion of the existing code. Due to inheritance, programmers can reuse the code in the superclass for a variety of applications
- ❖ **Extendibility:** it becomes easier to extend an existing application
- ❖ **Changes** in fields and methods shared by different subclasses can be made only once
- ❖ **Support and maintenance** of applications becomes easier, as the hierarchical structure of interclass relationships is clearly expressed

- ❖ One can implement **abstract classes** defining generic behavior. A large part of such abstract class remains undefined, with other programmers filling in details making specialized subclasses

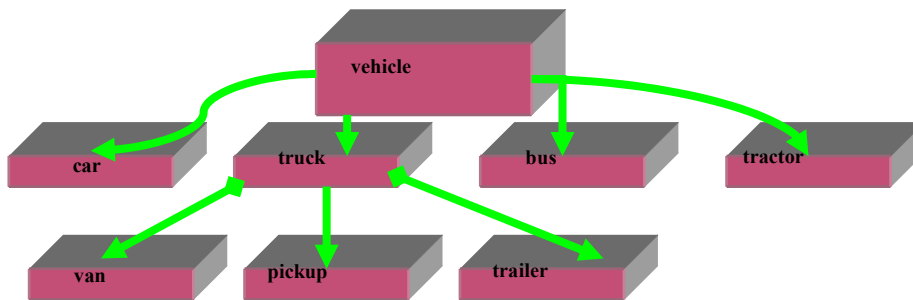
Inheritance is usually depicted by an arrow \uparrow directed from subclass to superclass.

Arranging classes in hierarchies

- **Subclasses of a Superclass**



- **Subclasses of the class *Vehicle***



Polymorphism

This intimidating term (Greek – many forms) simply means that different objects can respond to the same message in different ways. For a programmer, this is important since it becomes possible to use the same interface for a number of actions (methods). For instance, a unique *print* method can be defined for all documents – text documents, drawings, worksheets, tables, etc. - to be printed by sending the unified message *print*; we don't care how this method actually works for a particular document. This example hints that languages supporting polymorphism make it possible to manipulate objects from different classes, not necessarily related through inheritance, by common methods.

Various objects can interpret the same generic message differently. A *Car* object can implement its own version of *moveTo* message, as well as a *Train*, a *Ship*, an *Airplane*, a *Person*. The procedural programming does not work this way - a *moveTo* message generates a list of commands: *carMoveTo*, *TruckMoveTo*, *TrainMoveTo*, *ShipMoveTo*,...

One may notice that human communication is intrinsically polymorphic: one can ask a number of people the same question and get different answers. Object interactions have a similar versatility: each object can have its own unique response to the same message.

How polymorphism works

Let us consider how the flexibility of polymorphism can be used in the programming practice. Suppose we have to write a program, e.g. in graphics package, creating shapes on the screen: circles, triangles, rectangles, etc. The program must place the shapes on the computer screen, reposition them and change their size. In most of object-oriented languages, we would start by declaring shapes:

```
Circle circle;
Rectangle rectangle;
Oval oval;
Triangle t;
```

In order to make the shapes grow, we must invoke a method, for example:

```
circle.grow(change);
```

where the parameter in parentheses denotes the change of the object, e.g. in percentage. Analogously, we can invoke similar methods to make other shapes grow

```
rectangle.grow(change);
oval.grow(change);
t.grow(change);
```

which means that each class of objects has a method `grow`.

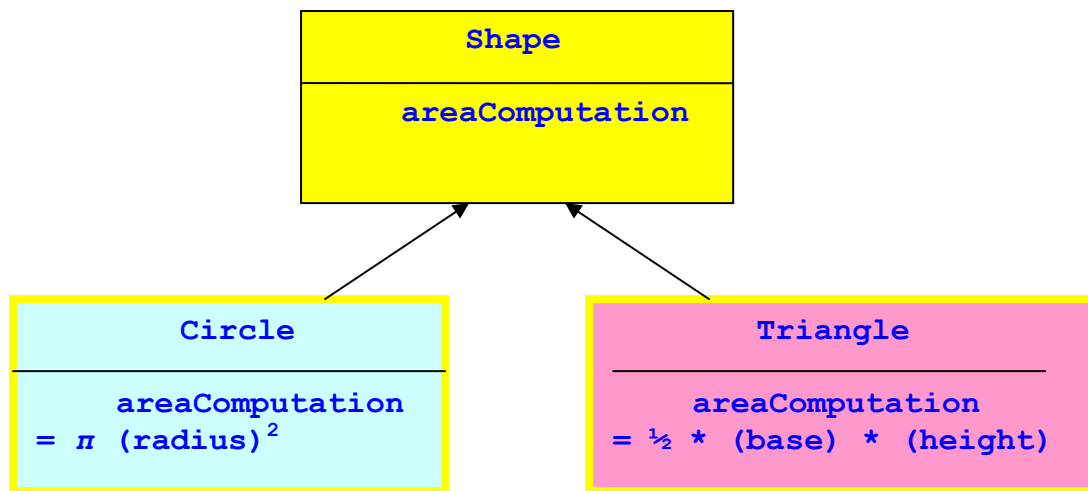
When the program runs, the polymorphic system selects the correct version of `grow` from all the methods with the same name. It does this by knowing the type of an object (triangle, rectangle, circle, etc.), which has already been declared. Here we see again that the same method is used for different objects – this is polymorphism.

Furthermore, we may invoke the same method to enlarge or scale any shape:

```
shape.grow(size);
```

Polymorphism can be conveniently connected with inheritance by providing a common interface for different classes within the same inheritance hierarchy.

Consider for example the following class diagram:



Here Circle and Triangle are subclasses of the superclass Shape. The method `areaComputation` returns the area of the respective geometrical figure and is declared in the class Shape. No method for area calculation can be defined without explicit determination of a specific shape. However, subclasses can inherit the generic method and rewrite its definition by using appropriate area computation algorithms. Again, we may notice that the same message is being sent to all the classes, but their reactions may be different. This property adds great flexibility to the programming process, allowing one to easily treat inhomogeneous objects. If we are to handle many various shapes, e.g. rectangles, parallelograms, circles, etc., we still can broadcast just one message (in this example, `areaComputation`) and anticipate a correct answer from each of them. We do not need to know beforehand what specific shape each object has.

Advantages of polymorphism

The above examples demonstrate that polymorphism can make the program creation process simpler and faster. Let us now try to summarize the advantages of polymorphism in object-oriented programming.

The run-time system invokes the needed version of a method applied to a particular object. In the preceding examples, the respective methods were `grow` and `areaComputation`.

There exists the possibility to apply the same method name to different objects. This possibility is especially convenient when the objects in question represent **subclasses** of a single **class**. In the above examples Circle, Triangle, Rectangle, Oval are subclasses of a class Shape.

In other words, a method can have a multitude of implementations, but just one of them is automatically selected when the program is running – depending on the type of the object that is being handled.

So polymorphism allows one to write a single statement instead of a series of statements, thus making the whole code much more concise and transparent.

Java specifics: when you are writing a code in Java, you are not aware *a priori* that Java correctly selects the appropriate version of the invoked method. Polymorphism invisibly ensures such selection.

OOP and mathematical objects

Our primary aim is to learn how to exploit the advantages of OOP for numerical evaluations. Most numerical algorithms had been designed before the ubiquitous spread of computers and long before OOP and Java – the most popular object oriented language – appeared. Numerical algorithms were invented to accelerate human computation. Can OOP and specifically Java make computation easier?

In order to answer this rather a philosophical question, let us formulate more specific questions:

- ❖ How can object-oriented concepts be applied to manipulate mathematical objects?
- ❖ Can one illustrate mapping between mathematical and computer (software) object using OOP?
- ❖ To what extent the behavior of the software object can reproduce interrelationships in numerical mathematics?

We shall try to answer these questions using some practical examples from numerical modeling, but before this endeavor, we have to study some foundations of OOP and of Java programming techniques. So far we can only note that:

1. Mathematical objects are organized in **hierarchies**, e.g. natural integers are contained within signed integers, which are included in rational numbers, which are contained within real numbers, which form a subset of complex numbers, etc. It is almost always the Russian doll principles.
2. Mathematical objects use **polymorphism**, which is manifested by the fact that one operation may be defined for many entities. For example, addition or multiplication is defined for numbers, vectors, tensors, matrices, polynomials and other classes.
3. **Inheritance**: general properties can be formulated as an abstract concept, e.g. a group, and used to prove a specific property for a given object. This deductive approach traditionally was extremely powerful, especially in theoretical and mathematical physics - without being called “inheritance”.

So the object-class-method structure of OOP is similar to that of mathematical constructions.

OOP as an environment for scientific computing

Software construction for scientific computing is a difficult task. Codes in scientific software are mostly large and complex, based on rather sophisticated mathematics and vast amounts of specific domain knowledge. Such codes usually must process large data sets, so there is an additional requirement of efficiency and high performance. To make proper optimizations, fair knowledge of modern computer architectures and compilers is necessary, which requires a lot of supplementary training and tends to complicate scientific codes even further.

Traditionally, most scientists and engineers have been programming in FORTRAN and C (if at all). Meanwhile, new programming techniques have been created during the last decade in order to manage programming complexity and to build close mapping for scientific abstractions. Under the layers of new “object oriented” terminology, a backbone of profound

ideas can be found, indicating the way to faster and more convenient software creation processes. If we take FORTRAN as a competitive environment for scientific coding, we may see that although it has excellent compilers, it is not object oriented and FORTRAN codes are not easily portable. For the scientific computing, object orientation is important because it allows one to easily investigate the interior object structure and to remove objects. Such inspection of inner structure of selected objects can be exercised largely due to encapsulation of data and procedures, while leaving the other parts of the program intact. Object-orientation in modeling, analysis, design and programming has also proven its merits during the recent years allowing greater complexity, robustness, extensibility and reusability of codes.

In practical terms, discussing the advantages of OOP for scientific computing, one must answer the following crucial questions:

- How to incorporate objects into programs?
- How to maximize code efficiency?
- How to ensure code reuse between many different algorithms?

Five features of a pure object-oriented language

1. Everything is an object (a student, a building, a bank account, a matrix, a function, a service, etc.).
2. A program is a collection of objects telling each other what to do by sending messages; the latter is a request to call a function belonging to a given object.
3. New object is created by making a package containing existing objects.
4. Each object is an instance of a class defining its characteristics (data) and behavior (functionality).
5. All objects of a given class can receive the same messages.

Objects are natural building blocks

Object programming reflects the Nature's techniques to manage complexity. One must recall that all living creatures are composed of elementary building blocks – cells, which are organic packages combining related **data** and **behavior**. Data is mostly contained in protein molecules inside cell nucleus. Behavior – from energy conversion to movement – is exercised by the structures outside the nucleus. Thus cells encapsulate data and behavior.

Encapsulation is the principle that proved to be quite successful in natural systems. Cells are surrounded by a membrane protecting inner workings of a cell from outside intrusion. Membrane allows only predetermined exchanges with other cells. Chemical messages are recognized by the membrane and passed through to the cell's interior.

Membrane is an interface hiding the cell complexity. Message-based interaction simplifies the system's functioning: a cell sends the appropriate chemical message, and the receiving cell responds. This analogy shows that objects in OOP are similar to cells.

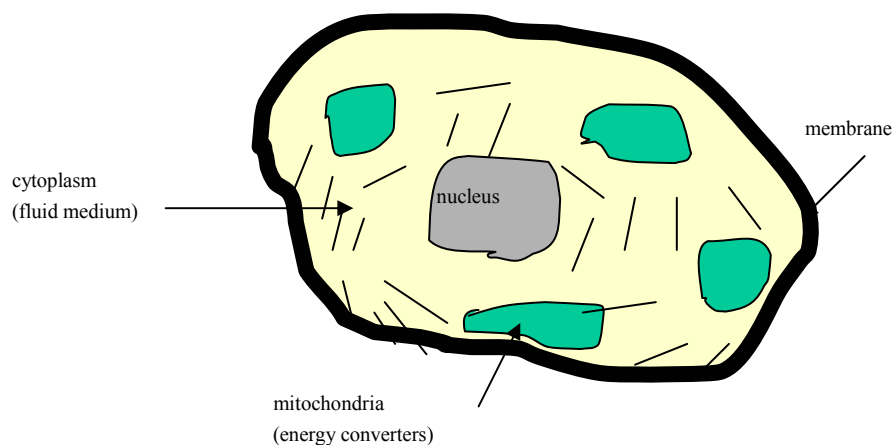
In principle, any existing thing is an object. Cars, animals, flowers, matrices are examples of objects. Objects can be characterized by certain features. Cars, for example, can be typified according to their color, size, power, and other attributes. Cars may undergo certain operations like starting engine, changing gear, turning on headlights, etc. Similarly, matrix

objects possess such characteristics as the number of rows and columns, rank, and so on. There are also operations on matrices like addition, multiplication, tensor product, decomposition into other matrices, etc. This description of objects hints that the internal object structure must embrace two parts: attributes, i.e. data, and methods, i.e. operations on objects.

In fact, attributes describe an object features in a specific moment of time. Thus attributes tell us something about an object, but based on them we are not fully aware how the object can act. Attributes may be thought of as local data associated with a given object: e.g. a dog has attributes like race, sex, age, weight; a polynomial has degree (power of the senior term), number of terms, and coefficients for each term. Here we see that a set of attributes that we declare for an object depends on modeling requirements. For example, in the case of objects representing the class **Matrix**, the attribute **determinantValue** may be extremely important for analytical models and practically of no importance for numerical modeling.

Cell as a universal building block

- ❖ All cells share a common structure and operate on the same basic principles. Within this genetically determined structure, infinite variability is possible. Objects tend to be equally flexible.



Cell and object variability is not chaotic but hierarchically classified.

Object-oriented programming guidelines

Before we proceed to the description of Java programming and use of Java for numerical applications, let us formulate the main technical challenges of object approach.

- Object orientation: what does it mean to declare a class and create an object?
- How to call the methods and interact with them?
- How to display an object (even if all that you see is lines of code)?

Programming with Java

Java requires you to program using OOP techniques. In fact, it is not possible to write a program in Java without resorting to object-oriented approach. No matter how good a

programmer you may be using procedural techniques, still you must program using objects in Java.

This course is mainly aimed towards object-oriented implementation of scientific computing applications, using Java in the context of numerical methods. Java is now competing to become the programming language of science. Java is available for practically all computer platforms and has been adopted by both non-professional and professional programmers. By non-professional programmers, we can maintain also scientists using computer as an auxiliary problem-solving tool. Until recently, computational scientists – often the most demanding group of programmers - tended to reject Java because of its reduced execution speed as compared to C or FORTRAN. However now the same group is beginning to recognize Java's potential as a programming tool for the most mathematically intensive computing applications. The latter now range from high energy physics to earthquake prediction. Java based object-oriented software for solving large sparse systems of linear equations (SLE) is under development. To obtain efficient SLE solvers, sophisticated algorithms and data structures are needed, especially to provide efficient software for distributed and parallel platforms.

Java has attracted much attention in the information technology (IT) world primarily due to its wide use on the Internet. One may notice that network computing (over fast and long-haul LAN, Internet, Intranet, etc.) is now beginning to play a substantial role in supporting collaborative scientific and engineering endeavors. Java can be regarded as a programming language built on the intersection of object oriented technology and network computing. In this connection, Java may become the key technology for scientific problem solving environment (PSE) allowing the seamless information flow between heterogeneous subsystems. Usually, the following attractions of Java as a programming language for high-end users are listed:

- Platform independence, incorporating both the architecture neutrality and portability. In practice, however, implementation of Java on different platforms can be slightly different, which may lead to some inconsistencies as far as the code behavior goes. Nevertheless, porting Java codes is substantially less laborious than porting programs in other languages. No platform specific calls or procedures must be learned.
- Java has a progressively enlarged set of libraries covering the range from commerce to mathematical functions. Multimedia and images are also included in this set of libraries. A comprehensive list of numerical libraries can be found at the Java Numerics Web site (<http://math.nist.gov/javanumerics/>). Java libraries and packages are discussed in subsequent sections.
- Comparatively easy programming. Syntax of Java has been purposely designed to make programming easier (as compared e.g. to C/C++). This fact contributed to the popularity of Java, which has become a language selected by educational institutions for learning object oriented software development, computer modeling, software engineering and Internet technologies. As a result, a new generation of IT specialists is raised who have learnt object-oriented programming through Java, thus making the language still more attractive for software development.
- Java has one of the best known electronic (and possibly also hardcopy) training resources, which also results in a growing number of people using it.
- Widespread use on the Internet. Java has an extensive library of network classes based on TCP/IP and higher level protocols such as FTP and HTTP.

This is a considerable improvement over pre-Internet age languages, such as C, C++, FORTRAN, etc., making the task of adding Internet functionality to Java programs much simpler than while using previous languages.

- Distributed computing environment. Java objects can be dynamically loaded, controlled and transferred over the network, which facilitates implementation of workgroup tasks.
- Java is a dynamic language. It has some features that allow one to adapt to a changing (dynamic) environment. Since the language is to a large extent an interpreted one (see below), classes and libraries do not have to be recompiled each time when changes are produced in other modules. For instance, new methods and variables can be added to classes without a need to recompile the dependent (e.g. derived) classes.
- Multithreading. This feature allows a single Java program to perform more than one task at the same time. This is considered to ensure better user interactivity and real-time behavior. Besides, there is also the possibility to fully utilize multiprocessor systems due to multithreading. The concept of multithreading is discussed in the final part of the present course.
- Safety. What is so special about Java? It lets you write so called applets (see below) that can be downloaded from the Internet and played **safely** within a Web browser. You do not have to write an applet to the hard disk.

What is Java?

Java (with a capital J) is a high-level, third generation programming language. Being compared to numerous other programming languages, Java is most similar to C++ and, lately, to C#. Java is said to be a *strongly object-oriented* language, which not only means that a program in Java is composed of interrelated objects, but also that everything should conform to the object-oriented paradigm. In other words, it is impossible to write even the simplest code that would not fit into an object model – no Java statements exist without of a defined class. Even the main program – be it an application or an applet – is a class. This peculiarity forces the programmer to think always in terms of objects and classes (even when there are no natural objects to work with).

Historical Background

Java was developed around 1992-1993 by engineers at Sun Microsystems, originally under the code name of Oak. The goal of the project was to create a programming language that was simple and economical in order to fit in consumer electronic devices, e.g. cable TV switch boxes, hi-fi systems, washing machines and the like. Nowadays we would call this technology “embedded systems”. Such devices are typically controlled by chipsets that have a very limited computing power and memory, therefore codes must be very compact. Besides, different manufacturers would tend to use a variety of CPU architectures, therefore the language should ensure architecture neutrality. To achieve this neutrality the concept of a **virtual machine** was invented (see below): any platform having a correctly implemented virtual machine could run the Java code. In addition, the system should be programmed quite easily, ideally by electrical or electronics engineers without the need of prolonged and highly specialized training.

It is interesting that initially neither Sun nor electronics companies were interested in the system, and approximately by 1994 the project was stopped. However, in the meantime the Internet was gradually becoming a ubiquitous communication medium, and the Sun engineers

realized that compactness and architectural invariance were just the right features for emerging Internet technologies. The first one hundred per cent Java-enabled Web browser (Hot Java) that appeared in 1995 vividly demonstrated the efficacy of Java **applets** – small interpreted and platform independent programs loaded directly from the server, and starting from 1995 Netscape incorporated Java technology into their products. As a result, dynamic Java applets have become widespread over Web pages, and Java itself gained enormous popularity. One can read more about the Java early stage in: G. Cornell, C. Horstmann. **Core Java**. SunSoft Press, 1997.

Java as a platform

Nowadays, apart from its usability for Web authoring, Java is considered one of the leading platforms for software application development. What is a platform? Roughly speaking, it is an IT buzzword denoting a combination of hardware and system software. A software platform is loosely understood as a set of programming slots for many applications and devices. In this sense, Java is more than just a language – it has some features of operating system providing “the glue” that links tasks, applications and procedures into a cross-platform interface. A wide spectrum of platform-neutral class libraries, in particular those intended to compose GUI (graphical user interface) is also a great advantage for operating in heterogeneous environment.

What is the problem with a particular choice of a platform? The matter is that applications are closely tied to specific hardware and operating systems, for instance a Windows program will not run on UNIX workstation, a Mac application will not run on IBM mainframe and so on. As we have already seen, Java was designed as platform independent from its very inception: e.g. Java applets are stored on a host site and can run on **any** computer, no matter under which operating system it works. The only requirement is to have a Java-enabled browser. The term “Java-enabled” means that inside the browser there is a program called **Java Virtual Machine** (JVM) that can run Java for a given computer. The Java code coming over the net is in a standard form called **bytecode**.

The Java bytecode

The idea of introducing the virtual machine, as we have seen, is to solve the problem of platform-dependence. For this purpose, the Java compiler produces not the executable code for a particular machine (like e.g. FORTRAN or C compiler), but a standard format code – exactly the same on every platform. It is this generated code, nicknamed bytecode, which is interpreted by the Java run-time system – JVM. Here is an example of a bytecode fragment (in hexadecimal):

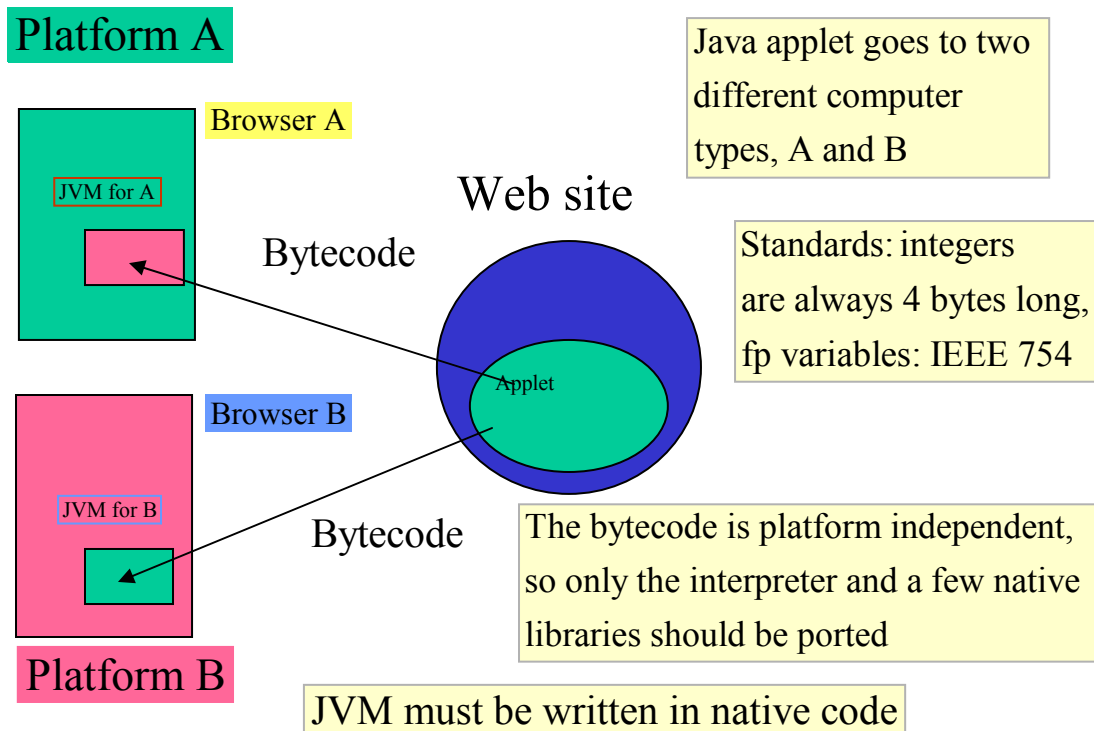
```
BE FE BA 2C 00 02 3E 2D ..
```

This sequence looks like a machine language, but unlike the latter, it is the same for all platforms. The JVM can understand the bytecode, e.g. in the applet coming from the Internet, and run it properly on a particular computer.

A Java program (source code) that has been compiled into an intermediate code – a bytecode - still needs an interpreter to execute it on a given platform. The Java interpreter incorporated into the JVM reads bytecode and instantly translates it into the machine native language (on the fly). It is in this sense, that Java is considered to be an interpreted language (at least

partly). Compilation happens just once, whereas interpretation occurs every time the program is executed.

How Java works



Pieces of Java

Compiler: javac
Interpreter: java
Programming language: Java

} collectively referred to as Java

Programming process in Java

There are three steps of creating a Java program:

1. writing the code;
2. compiling the code;
3. running the code.

As a rule, three windows in any computer windowing system correspond to the above programming steps.

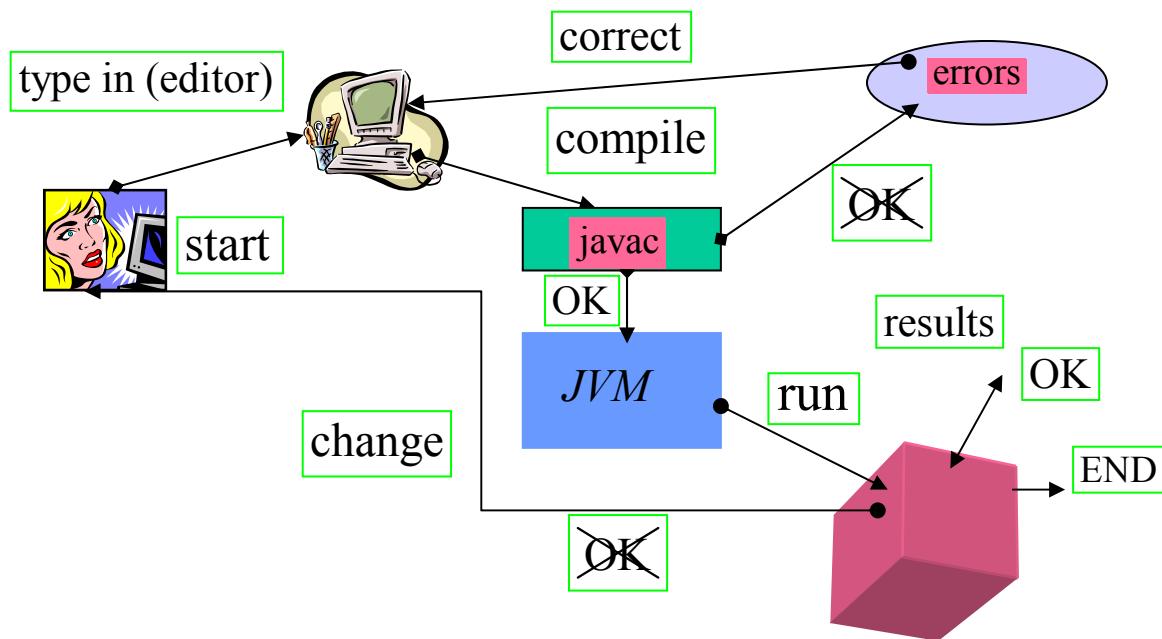
1. The process of writing the code is largely reduced to creating sequences of instructions (lines of code), which are written in a particular programming language. This creative process of producing a source code is expressed through an editor (e.g. Notepad, Emacs, vi, Aditor, etc.). There are a lot of available editors in the freeware or shareware markets; however practical experience shows that powerful text processors like MS Word, Word Perfect, and even Star Office together with its flavor Open Office are not convenient, since they tend to save data in their proprietary formats, whereas the source code should always be saved as a simple ASCII text.

Java programs look like stylized English. The syntax in Java is important and should be learned precisely, since any mistake causes the compiler to reject the code. One must learn very carefully how to type, save and, at the second step, compile the program. Java is case sensitive, e.g. **CLASS** is not the same as **class**, so one must be rather attentive in using the correct letter case. The source code should be always saved in a file ***.java**. Here lies some minor difficulty with Microsoft Windows system. The matter is that previous versions of Windows text editors automatically added the “.txt” extension, so one could unexpectedly finish up with ***.java.txt** file in the source code, which is not compiled. The remedy may be enclosing the filename in double quotes (“filename”) in the **Save** dialog box. Newer versions of Windows editors provide drop-down menus allowing one to save files in the desirable format, however one must be attentive.

When one is working under UNIX or Linux, GNU Emacs (Xemacs) seems to be a good choice. Emacs is probably the most versatile text editor in the UNIX world and very convenient for source code creation. Using Emacs, you can easily apply multiple windows and buffers, so that working with many files at once is greatly simplified. Besides, Emacs can be readily tailored as a programmer’s working environment, both supporting programming in a number of languages and facilitating compilation tasks. Emacs is free of charge, which is also an advantage.

Once the text is ready, it is submitted to a compiler denoted as *javac*. The latter has two main functions:

- 1) Checking for syntax errors
- 2) Translating the program into a form that can be run on a computer – the compilation target for Java is the JVM



The term “compiler” stems from the Latin verb “compilare”, which means to collect the trash, sometimes even with the connotation of robbing.

The syntax example for compiler:

```
javac [options] file1.java file2.java
```

The compiler *javac* produces the *.class file, provided all compilation errors have been removed.

The standard options are, for instance, `classpath [-classpath]` that defines the “environment”, that is the path where the *.class file should be found, or the directory `[-d]` where the *.class files are saved. By default, these files are saved in the same directory as .java files (source code). It is important to note that the compiler can operate on more than one source code file - in principle, a list of files can be simultaneously compiled.

To start the java application, one must use the interpreter program *java*.

The syntax example for interpreter:

```
java [options] classfile [argument]
```

Here one must differentiate between standard and non-standard options. For example, `-classpath` is a representative of standard options, whereas an option `-XmxN` defining the upper bound for the reserved memory is assumed non-standard.

The programs *javac* (Java compiler) and *java* (Java interpreter) are the most important tools for the creation of Java applications. These tools, together with other Java development programs are included in the so called Java Development Kit (JDK), now often called also the

Software Development Kit (SDK). The latter is a sufficiently complex developer system for Java programmers. In fact, installing JDK is the first thing that must be done to start learning Java programming. Certain hardware and software requirements should be fulfilled in order this installation would be correctly executed. For example, RAM should exceed 48 MB and free space on a hard disk should not be less than 200 MB.

Two approaches: applets vs. applications

We have already mentioned that Java programs can be written and executed in two main fashions:

- Stand-alone **application** from the command line.
- **Applet** which runs a Java-capable browser.

Programming an "application" in Java is substantially different from programming an "applet." Applets are designed to be downloaded and executed on-line under control of a browser. Hence, the functionality of applets is restricted by security requirements, i.e. by attempts to prevent downloaded applets from hurting your data or computer. No such restrictions apply to the functionality of a Java application.

Properties of a good program

A **good** program should be **clearly structured**. The structure of a program indicates how its parts are interconnected. Connections between the parts of a program should be reasonable and sound.

While creating a program, one should aim, primarily, to build the required model or to achieve the needed solution. However at the same time, the program should also be **readable, reusable and efficient**.

Correctness: much software is being released with many remaining errors (bugs). For instance, in commercial C programs it was considered admissible to have in average one error for every 55 lines of code. If the comparative proportion of errors had been permitted e.g. in the construction industry, the observed landscape would have contained a lot of ruins.

There are two golden rules to achieve a fair correctness level in software engineering:

1. Follow good programming practice while writing a program – mostly by example and code reuse.
2. Test the program thoroughly throughout its development.

Data types in Java

Every variable in Java has a declared type determining what values it can take and what operations are allowed. Java variables never have undefined values, e.g. an integer is always four bytes and has a sign.

There exist 8 primitive data types in Java. Below these data types are listed together with their predefined properties.

boolean

1-bit variable, which is allowed to take only two values: **true** and **false**.

The latter are considered special language constants, e.g. `true` and `false` are not the same as `True` and `False`, `TRUE` and `FALSE`, `one` and `zero`, `zero` and `non-zero`, etc. The `boolean` variables can be transformed in no other data type, and conversely no other variable can be transformed into `boolean` one.

byte

8 bit width, signed. Value range from -128 to 127

short

16 bit, signed. Value range from -32768 to 32767

int

32 bit, signed. Value range from -2 147 483 648 to 2 147 483 647

As relates to all number types, variables of the type `int` can be converted into variables of other number types i.e. `byte`, `short`, `long`, `float`, `double`.

Examples: 73, -205, 1024

long

64 bit, signed. Value range from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807

Examples: 73L, -205L, 194698L

float

32 bit, signed. Value range from 1.40129846432481707e-45 to 3.40282346638528860e+38.

Floating point arithmetic and data format are defined by IEEE 754

Examples: 73.5, -205.5, 19.469

double

64 bit, signed. Value range from 4.94065645841246544e-324d to

1.79769313486231570e+308d

Data format and arithmetic are defined by IEEE 754

Examples: 73.5, -205.5, 19.469

char

16 bit, unsigned.

Objects of the `char` type can be transformed into no other type and vice versa. Characters in the `char` type are defined through Unicode.

Examples: 'a', '5', 'x'

Apart from the above primitive data types, there are so called “complex data types”.

Sometimes they are also known as object types emphasizing that they hold references to respective objects. Object types are those defined by classes, and some classes are defined by the standard Java system (e.g. `String`), whereas others are those classes that we can write ourselves.

String

`String` is an object or a reference data type. `String` contains 0 or more number of symbols enclosed in double quotes. Actually, strings are always enclosed in double quotes.

Example: “This is a string”

array

Arrays are objects. Multidimensional arrays are created as arrays of arrays (see below)

What is the meaning of such compartmentalization? To disclose this meaning, we must recall that Java was born as a simple language for representing automatic behavior of electronic devices, in contemporary language – embedded systems. There should have been no ambiguity whatever in how to interpret fields and expressions. This is known as a **strongly typed** and also **strongly classed** language. One may notice that currently the most modern programming languages follow the same pattern. Terms *Strongly typed* and *strongly classed* mean that every field and expression has a type and belongs to some class; moreover, only the expressions of the same types and classes can be used together. Simply speaking, this requirement ensures that strings and numbers cannot be mixed up, nor, for example, can we entangle cars and animals, etc. The subdivision into clearly defined types prevents us from carelessly making a mess out of classes, objects and methods, especially when there are thousands lines of code and control is naturally weakened.

The main difference between types is how values are stored in the memory. Primitive values are stored directly as variable values, whereas for object types a reference to the object is stored. Primitive types in Java are non-object types. Primitive types have no methods.

A type also governs how much storage space is available for the values that are stored in variables (fields) of a given type. In numerical computations, the four – **int**, **long**, **float** and **double** – are most commonly used. The **int** type is needed to represent usual positive and negative integer numbers within a range extending approximately two billion; **long** is intended for even longer integer numbers or for expressing results of calculations with **int** types; **double** is designed for real numbers – in fact, numbers with fractional parts – or to express the calculation results, with real numbers being involved (one real number is enough).

Java programming basics

All Java programs consist of a set of interdependent classes - one or more **class** definitions. So, first of all a class should be defined. There is a certain pattern in Java pertinent to a class definition: after the keyword **class** stands the name of the class. Then a block follows where variables and methods are declared. One must pay attention to the fact that in Java, in distinction to some other object-oriented languages, methods must always be declared inside of the block defining a class. Variables that should be valid for the whole class are standing inside the class block but outside of the method block. Everything that stands inside the latter (e.g. names) is local and valid only for a given method.

Thus, the constitutive skeleton of any Java program is a class definition. All methods (functions) and variables must be defined *inside* a **class** definition. There can be no freestanding methods or global variables.

The *primary* class defined for a Java *application* is sometimes referred to as the *controlling class*. Later we shall see that a stand-alone Java *application* requires a method named **main** in its *controlling class*, whereas an *applet* does not require a **main** method.

When a class is defined, a new *type* emerges. The new type definition can then be used to instantiate one or more objects of that new type. The new class describes both the *data* contained within, and the *behavior* of objects instantiated according to the new type.

In connection with new types we may reiterate that the data is contained in variables declared within the class (often called interchangeably *variables*, *data members*, or *attributes*). The

behavior is controlled by methods defined within the class (often called also *member functions*). One must always remember that each variable belongs to a certain data type.

As an example, let us consider the class of all people “Human”. Each person is an instance (object) of the class “Human”. So, we can create an object having the value “paul” of the variable (attribute) “name”. Another object would have e.g. the value “peter”. Let us see how the process of object creation is manifested in Java programming:

```
/* defining the class */
public class Human
{
    /*declaration of attributes*/
    protected String name; //declaration of the attribute "name"

    /* methods are defined*/
    public Human(String the_name)
    {
        name = the_name; //The class variable "name" will be
                        // assigned the value of the method parameter
    }

    public String getName( )
    {
        return name; //The method simply returns
                    // the value of the class variable
    }

    public void setName(String the_name)
    {
        name = the_name; // one more assignment
    }
}
```

The class File

Let us recollect that compiled Java programs are stored in "bytecode" form in a file with an extension of `.class` where the name of the file is the same as the name of the class defined in the program (e.g. the primary or *controlling* class).

To summarize the steps needed to start programming in Java, the following checklist may be useful:

Getting Started

HOW TO COMPILE AND RUN A JAVA APPLICATION

1. Download and install the JDK following the installation instructions (now JDK 1.4 is available). You can have all the relevant information from <http://java.sun.com/j2se/1.4>

Download and install also the documentation

2. Using any editor that can produce a text file (such as Notepad), create a source code file, with the extension of the file name being `.java`. This file will contain your actual Java instructions.

3. Change directory to the directory containing the source file. It doesn't really matter which directory the source file lies in, but it is convenient to put all Java files in a separate directory.

4. Assume that the name of the file is `myFile.java` (just for reference purposes)

5. To compile the file, enter the following command at the prompt:

```
javac myFile.java
```

6. Correct any compiler errors that may appear. Once you have corrected all compiler errors, the Java compiler *javac* will execute and return immediately to the prompt **with no output**. At that point, the directory should also contain a file named `myFile.class`. This is the compiled file.

7. To run the program, enter the following command:

```
java myFile
```

Declaration of a class in Java

In the preceding section a piece of code was demonstrated, built up according to classical object-oriented principles. Let us typify the specific features of such coding. We have already mentioned that primarily a class should be defined. The syntax for defining a class in Java is shown in the box:

```
class MyClassName{
    . . .//The simplest class definition
} //End of class definition.
```

This syntax defines a class and creates a new type named `MyClassName`. The definitions of variables and methods are inserted between the opening and closing brackets. One may note that there is no semicolon following the closing bracket in a Java class definition.

Frequently, a class definition can already say a lot about a class. One can, for example, define a class having the following features:

- indicate the superclass of the defined class
- declare class scope modifiers (also denoted as qualifiers): private, protected, public and also abstract or final
- list all interfaces implemented by the defined class (see below)

In general, declaration of a class has the following form:

```
[modifiers] class ClassName [extends SuperClassName][implements
InterfaceX]{

    ..... // here the class body
}
```

All the items between [] are optional. If one does not declare them explicitly, the Java compiler would assume certain default values.

We have already mentioned that every class in Java has a superclass. If the latter is not specified, the Java compiler assumes it to be the **Object** class. The **Object** class is a special one defined in the package `java.lang` (see below) and considered an ancestor of all other classes. Thus, the **Object** class stands at the top of the inheritance hierarchy.

To summarize, we may say that **superclass** is the immediate parent class of a new class, and the new class itself is known as the **subclass**. Sometimes, the word **superclass** is used to indicate the collection of classes in the inheritance hierarchy from which a specific class is derived.

The explicit declaration of a superclass is performed by the keyword **extends** plus the name of a superclass (see examples below)

In principle, classes can be empty i.e. containing nothing except the class name. The general syntax of an empty class definition in Java looks like

```
class NameOfClass{}
```

Creating objects

A large Java program typically has many classes, each of them allowing many methods that call each other using statements. In the class **Human**, we can create two incarnations, two human objects – **paul** and **peter** (see below). This is an illustration of how objects might be spawned from class definitions. One may notice that apart from a name of the instance, some other parameters characterizing the object may be specified. All these data values are referred to as the *object state*. This data is stored in object fields.

It is important to note that objects of the same class all have the same fields. Only values in a field are different for different instances. For example, a sphere has a field ‘radius’ and a pyramid has a field ‘height’. A concrete sphere – the sphere – and a given pyramid – the pyramid – acquire specific (instance) values of, respectively, radius and height. That is why fields are also called instance variables, and as for any variable, the value stored in a field can be changed.

From the example with sphere and pyramid classes, it is clear that each field must have its own declaration in the source code. Moreover, each field must be of a certain variable type (e.g. **int**, **double**, etc.) and can be characterized by a certain access modifier, for example:

```
private int numberOfStudents;
```

Types, number, and the name of fields are defined for the whole class – remember that the classes serve as universal templates for object - and not in an object. For example, the class **Sphere** in some computer graphics Java application may define for each spherical object six fields: **radius**, **xPosition**, **yPosition**, **zPosition**, **color**, **isVisible**. The types for the fields are also defined in the class. Here the first four are of the type **int**, then respectively **String** and **Boolean**.

When an object of the class **Sphere** is created, it automatically acquires the same fields as the respective class, which means that the values of those fields are stored in the object. In the above **sphere** example, this default ensures that each created sphere has its own color.

Methods and their parameters

We can recall that in accordance with general principles of object orientation, objects have methods that are used to interact – communicate - with them. For example, one can use a method to make a change in the object or get some information from it.

A minimal Java program must contain the following:

- defining a class
- writing a **main** method including the concepts: static methods, parameters, etc.
- declaration of the variables including the variable type
- a statement to create the object (e.g. **new**)
- calling a method, for example `void moveForward (int distance)`.

Remark. The expression in parentheses is called *signature of the method*; it defines a type (here `int`) and a parameter name (here: `distance`) The Java literature usually refers to the *method signature* as a collection of information about the method. It includes the method's name, *type*, and visibility, as well as its arguments and return type. In other words, the *method signature* can probably be thought of as providing all the information there exists about the *interface* to the method. In other words, it provides all the information that you need to know to be able to invoke the method.

- an assignment to a variable
- providing the list of parameters and the method signature

Remark. Methods can have parameters (e.g. `moveRight (int distance)`), and these parameters, in accordance with the general principle of strong typing, should belong to a defined type. Methods can be specified by what kind of data they are to return. If a method is not bound to return anything, the keyword `void` is used. Sometimes the `void` keyword is called the `void` return type. Thus, in the last example, it is necessary to write

```
void moveRight (int distance);
```

Java keywords

You must have probably noticed that short pieces of Java code given above as illustrations contain some strange words like `void`, `public`, `main`, `static`, `args`, etc. This esoteric terminology is typical of object-oriented languages and must be thoroughly studied. To this end, let us discuss all these Java incantations one by one.

The main() Method

We have seen that the first thing to do when attempting to program in Java is to declare a class. From here it follows that for every Java application there must be a main, controlling class. And consequently in a controlling class the **main** method should be defined for a Java application. (There is also a **main** function in a stand-alone C++ program.)

Signature of main() Method

When discussing methods, we saw that all methods have signatures. The **main()** method of the controlling class of every Java application must also have a signature, in fact one of the following two signatures. The first one seems to be preferable, since it better describes an array of **String** references (which is what is taken as an argument):

```
public static void main(String[] args)
public static void main(String args[])
```

We have also seen that a method in general may be treated as a sequence of actions that in the source code are included in {...} following the method title. Later we shall observe that in the canonical “Hello World!” example the **main** block contains just one line of code.

The difficulty here is that the **main** method in a Java program must always provide the formal argument list regardless of whether it is actually used in the program.

The generic method can also be interpreted as a function in a mathematical sense, therefore a necessary list of parameters within (...) is required (a method signature, see above). For the **main** method, the parameters **String [] args** or **String args[]** are given (although for the “Hello World!” example the above parameters are of no importance).

The terms **public**, **static** and **void** characterize properties of the **main** method.

It is important to note that the *controlling class* does not have exclusive rights on the **main** method. It is true that the controlling class of every Java application **must** contain a **main** method, but other classes in the same application are also allowed to have a **main** method. This fact may bring confusion, so the question arises, why you might want to do this?

The answer is that it may be desirable to provide a **main** method for a class in an application (that will not necessarily be the *controlling class*) to allow the class to be tested in a stand-alone mode, independently of any other classes.

public

The keyword **public** indicates that the method can be called by any object. Later we shall discuss the keywords (access modifiers) **public**, **private**, and **protected** in more detail. This is an important subject, but now we just need to mention that the term **public** is related to access control. Access rights exist for classes and their members including attributes or fields (variables) and methods. In case class variables are declared **public**, they can be universally accessed, i.e. by members of other classes. Access rights are also called *visibility* in Java

static

The keyword **static** indicates that the method is a *class* method. Such methods can be invoked without the need to instantiate an object of the class. This property is used by the Java interpreter to start the program by calling the **main** method of the class. It is often said that the method **main** is an entrance point to any Java program, which in effect means that the interpreter must identify it in the command in order to launch the program.

In short, the keyword **static** is used in Java to make a variable or method to become a *class* variable or method, whereas *instance* variables and *instance* methods can only be accessed through an object of the class in Java.

void

We have already mentioned that the keyword **void** indicates that the method does not return any value. This fact may be expressed more accurately as the result of the method (action) is of an indefinite type.

args

The formal parameter **args** is an array of type **String**, which contains arguments entered at the command line. Note once more that the **args** parameter must be specified irrespective of the developer's desire to enter a command-line argument and whether or not the code in the program actually uses this argument.

Supplementary material: the length property

In Java, **args** is an array object. Array objects have a property named **length**. The **args.length** property can be used in the code to determine the number of arguments actually entered by the programmer. The runtime system monitors the entry of command-line arguments and constructs the **String** array containing those arguments.

If the array of strings contains data, the first string in the array corresponds to the first argument (in distinction to other object-oriented languages, e.g. C++, where the array must commence with the name of the program). Command-line arguments along with strings and **String** arrays will be discussed later in more detail.

Java syntax: keyword table-I

Class/Object	Modifier	Primitive data type	Error handling
class interface new extends implements instanceof return	private protected public abstract final native static synchronized transient volatile	byte short int long float double boolean char	try catch throw, throws finally
			<p>The idea is: badly formed code shall not be run. Not all errors can be detected at compile-time; the rest must be handled at run-time</p>

Java syntax: keyword table-II

Loop	Condition	Packaging/Module	Constant/Reference
do while for break continue	if else default switch case	package import	void null this super true false

Commentary: A **package** is what one gets when using the **import** keyword to bring in the whole library.

The keyword **protected** is a cross between **private** and **public**. It is related to **inheritance** and permits the access to methods both inside the class and to those of derived classes.

In general, it is advisable to follow the Sun Microsystems programming conventions: <http://java.sun.com/docs/codeconv/index.html>

Comments in Java

It always makes sense to record your ideas and comment on the software you are developing. These comments would help not only other people to comprehend your way of thinking, but also yourself, when you decide to reconsider your code in some later time. Java supports three forms of comments, the first being inherited from C:

```
/* This is a comment supported in C and Java*/,
```

the second is inherited from C++:

```
// This is a comment supported in C++ and Java.
```

Everything between `/*` and `*/` is ignored by the compiler, this comment can be used on **multiple lines** and large blocks of code (multiline comments).

Everything on a **single line** after `//` is neglected, so this kind of comment can only be used at the end of the line (single-line comment).

Comments can be used liberally. However, it is recommended that comments should add real meaning to the code.

javadoc

The third style of comments: those starting from `/**` are handled by the compiler in a special way. This symbol indicates that the respective comment should be included in automatically generated documentation. Sometimes, the `/**` comments are called “announcements”.

Besides, there exists a special program in Java called *javadoc* and incorporated in JDK (i.e. delivered by the compiler), which provides a set of HTML pages to be filled up with commentaries.

```
/** special documentation comment  
used by the javadoc tool */
```

The *javadoc* tool is a program that is used to produce documentation for an application. This type of documentation is very useful for on-line or on-screen documentation.

Let us see on a simple example how a source code might be commented in Java. This example refers to a standard first Java program (see below).

Example:

```
// This is the Test program in Java
public class Test {
    public static void main (String args [])    {
        /* Let's print the line Hello World!*/
        System.out.println ("Hello World!");
    } // main ends here
} // Test ends here
```

The first program in Java

Like nearly all fairy tales that are commenced with “Once upon a time” (and analogously in other languages), Java starts with a “Hello World!” program. In fact, not only Java: ever since the early days of C, programmers have been experimenting with new languages by presenting the canonical “Hello World!” program. A number of contests have been sponsored recently by some US universities to create the smallest Java “Hello World!” program. This is not solely a curiosity: in the process of producing ultimately economical Java `.class` file that, when executed, would display on the console the text “Hello World!” (and only this text), one can learn a lot about the Java bytecode (actually class files) and about the inner working of the Java Virtual Machine (see e.g. *The Java Developers Journal*, v.7, No.7, p. 44, 2002).

This ubiquitous program is, in fact, an example of the class in Java (in the preceding example, we named it `Test`):

```
public class Hello {
    public static void main(String[ ] args)    {
        System.out.println("Hello World!");
    }
}
```

Let us discuss this Java version of “Hello World!” slightly more in detail with the aim to better understand on this primitive model the following Java language concepts introduced previously:

- The keyword `class` shows the beginning of a class
- Everything within `{..}` is denoted as a *block*. The block beginning right after the class name `Hello` includes the program parts describing methods and fields referenced.
- The `main` method in the controlling class of an *application* is *static*, which means that `main` must be a *class* method. The `main` method defined in the `class` definition controls the flow of the program
- *Class* methods can be invoked without the requirement to instantiate an object of the class.
- Running an application: when a Java *application* is launched, the Java interpreter `java` finds and invokes the `main` method in the class whose name coincides with the name of the class file specified in the command line. For example, to start the interpreter and run a Java application named `Hello`, a command-line statement such as the following:

```
java Hello
```

must be executed at the operating system prompt.

This statement instructs the operating system to start the java interpreter *java*, and then instructs the interpreter to find and execute the java application stored in the file named **Hello.class**.

The summary of the first programming experience:

This is a sample Java application named **Hello.java**.

When compiled, it produces a class file named **Hello.class**.

When it is run, the interpreter calls the **main** method defined in the controlling class. One must recall that the **main** method is a class method, and class methods can be invoked without the requirement to instantiate an object of the class.

The program displays the following words on the screen:

Hello World!

Analysis of code fragments

The first fragment shows the first line of the class definition for the controlling class

```
class Hello { //define the controlling class
```

class Files: The name of the *controlling class* should be **the same** as the name of the source file that contains it. Here we can remind once more that files containing source code in Java have an extension of *java*, whereas files produced by the compiler containing the controlling class have the same name as the controlling class, and an extension of *class*.

The second fragment begins the definition of the **main()** method.

```
public static void main(String[] args){
```

The next interesting fragment induces the string **Hello World!** to be displayed on a standard output device (e.g. the screen). This is a rather strong statement:

```
System.out.println("Hello World");
```

The following fragment ends the **main()** method, and also ends the class declaration for the class named **Hello**.

```
    }//end main  
}//End Hello class
```

A remark: many class files can be produced

It is important to note that a number of class files may be generated, that is the *java* compiler creates a separate file for every **class** definition contained in an application, even if **the same** source file contains two or more **class** definitions. Thus, the compilation of a large application can produce many different *class* files.

Jar Files

In order to consolidate all the generated *class* files into a single file, the so called *jar* file can be used. So jar file is a tool designed for more compact storage, distribution and transmission of files in Java.

Practical advice: is your Java environment correctly configured?

In connection with compiling and running the “Hello World” program, you can make sure that your Java environment is installed and configured correctly. To test it, you can, for example, type after the command line prompt:

```
javac nofile.java
```

If you get the response

```
error: Can't read: nofile.java
```

then the compiler seems to work right and you can start programming. If, on the other hand, the compiler responds as

```
javac: Command not found
```

or similar to it, then you are about to get an error message even in case your code is perfect. The problem may be that either your Java environment (JDK) is not correctly installed or class path is misconfigured.

We have discussed that there are three steps necessary to create a Java program: 1) writing the code; 2) compiling the code; 3) running the code. Under Windows, compiling and running the code may be done in a DOS shell:

```
C:> javac Hello.java
C:> java Hello
Hello World!
C:>
```

We may notice once more that we use the `.java` extension when compiling a file, but we do not use the `.class` extension when running a file.

Variables in Java

We have already touched upon the variables in Java; now let us discuss this issue more in detail. A variable in Java is treated as a named piece of memory that can be used to store information on a Java program. It is important to remember that each named piece of

memory, which is declared in the program, should store data of one particular type. For example, if we define a variable to store integers, it cannot be used to store e.g. 0.25 or a string.

Thus, to use variables, one must notify the compiler of the name and the type of the variable, that is termed as *declare* the variable. The syntax for declaring a variable in Java is to put the type of the variable before the variable name, as shown below. It is also possible, but not necessary, to initialize a variable in Java when it is declared.

```
int x1, x2 = 0;
```

This statement declares two variables of type **int**, initializing one of them to the value of the zero character.

In all programming languages, one can distinguish two groups of variables: **global**, i.e. those existing the whole program run-time, and **local**, i.e. required by a certain procedure or needed for a subroutine. There is a tendency not to have global variables in Java: *each variable is either local or is incorporated into a class definition*. One may consider class variables a replacement construct for global variables.

Here we dare to remind you that *class* variables are called **static** – they are ascribed to a fixed class. In practice, the **static** modifier means that a variable is attached to the class and not to the instance.

In case *class variables* are declared **public**, they can be universally accessed, that is by members of other classes. If the variables should be made *read-only* for other classes, one has to declare them **private**, with access *methods* being declared **public**.

Apart from class variables, there may be two more kinds of variables: local variables and member variables.

Local variables belong to functions (methods) and may be declared when needed. As a rule, local variables are declared in a method or within a block of code. The scope – visibility – of a local variable extends from its declaration to the end of a respective method, which is indicated by the first right curly bracket. In other words, the scope of a Java variable is defined by the block of code within which the variable is accessible.

The scope also determines when the variable is created, i.e. memory allocated to contain the data stored in the variable, and when it becomes a possible candidate for destruction, that is the memory would be returned to the operating system for recycling and re-use.

To sum it up, one can say that the scope of a variable places it in one of the following categories:

- member variable
- local variable
- method parameter

Since the work of a Java program is divided between several classes, objects and methods are created within them. Methods modify the state of the created object, i.e. the set of instance variables.

Local variables are typically associated with loops and because of this are sometimes called loop variables. As we shall see it later, Java applications in scientific computing are largely reduced to loops, therefore you may frequently encounter the term “loop variables” in this context.

Member variables characterize a class (class variable as a member of a class) or a member of an object instantiated from that class. Member variables represent a variable part of a class or an object and can be declared anywhere within a class, but not within the body of a method of the class. So, respectively there may be object member variables, or *instance* variables and class member variables.

A member variable declaration must have, at minimum, two components: the data type of the variable and its name, for example:

```
type variableName; // a minimal member variable declaration
```

A member variable’s name begins with a lowercase letter – to distinguish it from the class name. In connection with the variable declaration, it is pertinent to mention good programming style: to keep types, variable names and comments all lined up, for example:

```
int         temperature; // in Kelvin
int         pressure;   // in Pa
int         frequency;  // in MHz
double      salary;     // in Euro
double      tax;        // progressive; initially 30%
long        a, b, c;     // unknown parameters
double      x, y, z;     // real unknowns
```

One cannot declare more than one member variable in the same class with the same name, although a member variable and a method are allowed to have the same name.

An important technical question is: where exactly should the variables be declared? In principle, there are no strict rules specifying the exact place where variable declarations should be grouped together. Variables, in which objects store their data, can, strictly speaking, be declared anywhere in a Java class or method, but usual programming practice tends to place such declarations at the beginning (less frequently, at the end). As we have already mentioned, the syntax of a variable declaration may have the following forms:

Declaration of variables
type variableName;
type name1, name2, name3;
type name = value;

Thus the declaration of variables introduces, in general, more than one variables of the given type. Pay attention to the last form in the above table: it can be used to initialize a variable simultaneously with its declaration.

Declaration of constants

We used to ascribe names to some numerical values. For example, in the business jargon the value of 1000 is nicknamed “a grand”; the number 12 was historically called a dozen, and ten days is a decade. In mathematics, the number 3.141592 653589793238462643383279.. is called pi (π), the exponential constant 2.178281828459045235360287471352.. is called e , the Euler’s constant expressing the limit

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} - \ln n \right)$$

is called γ ; in the modern mathematical and computer modeling based on dynamical systems theory two Feigenbaum’s constants appear, and so on. There are also physical fundamental constants like the Avogadro number $N_A = 6.022136 * 10^{23}$, the fine structure constant that is very close to 1/137, etc. Giving names to quantities has a mnemonic value – numbers are easier to remember and use. In Java, named quantities whose values are not going to vary are called constants and declared with the help of the keyword **final**. For example,

```
static final int      majorityAge = 18;
static final double   caloriePerJoule = 4.187;
static final int      speedLimit = 100;
static final int      retirementAge = 65;
static final int      secondsPerHour = 3600;
static final double   kmPerMile = 1.609;
```

Here the **static** modifier indicates the *class* field. It means that the declaration of constants occurs only at class level, not inside methods – even not inside the **main** method. The second keyword, **final**, points to the fact that the contents of the respective field cannot be changed in the process of running the program – in other words, it is a constant.

Manipulating strings

Java has a useful collection of methods for character processing. The main use of strings is in input-output (i/o) and in user interface, for instance:

- text input from the user:
in a stand-alone application text may be entered at a prompt;
in an applet –from TextField or TextArea;
- content of many types of files may be considered as sequences of strings;
- filenames and directories (folders) are strings;
- we need strings to specify folders and URLs and to manipulate files;
- to display messages on a screen one has to put a text on command buttons

Most often, the strings in the Java language are used to declare variables and to provide initial values, e.g.

```
String myName = "Pankratov";
```

(but not restricted to such usage)

More professional use of strings may be illustrated by the following examples

```
String x, y, z;
String myCountry = new String("Germany");
/* This illustrates that the string length may be arbitrary */
```

Using the command `new` we allocate space for the `String` value.

We can assign one string to another, e.g.:

```
x = z;  
y = "France";  
y = ""; //a zero-length string
```

The old string is erased and replaced by updated value (the space occupied by the old value is freed due to automatic garbage collection)

Concatenation: performed using operator `+`

```
x = x + "whatever"; // this is termed as appending
```

- String values are placed between double quotes. But what if we need to display double quotes on a screen?

```
g.drawString("A "spiral" approach to teaching Java", 50, 75);  
  
/* This is wrong, and the compiler would inform you of the error */  
/* 'escape' characters, e.g. \, instructing the language to handle the  
following symbols as ordinary characters, not as the language elements */  
  
g.drawString("A \"spiral\" approach to teaching Java",50,75); //OK
```

- In the `String` class a length accessor method is defined with the following signature:

```
public int length ( )  
/* Returns the number of characters in this string */
```

Constructors

Constructors create new instances of a class, i.e. objects.

Constructors have the same name as their class and are, in fact, special methods. Let us consider, for example, a point in 3D space:

```
class ThreeDPoint {  
    double x;  
    double y;  
    double z;  
    ThreeDPoint (double xvalue, double yvalue, double zvalue) {  
        this.x = xvalue;  
        this.y = yvalue;  
        this.z = zvalue;  
    }  
}
```

Constructors are used alongside with the keyword `new` to create an object of the class

The **this** keyword

The keyword **this** produces a reference to the current object. It is used within the method body. Let us consider an example (needed for Java graphics):

```
public class Color {
    double hue, saturation, brightness;
    public void setColor (double hue, double saturation,
        double brightness) {
        this.hue = hue;
        this.saturation = saturation;
        this.brightness = brightness;
    }
}
```

The keyword **this** refers to the member variables (fields), because they are obscured (usually the term “shadowed” is used) by the parameters **hue**, **saturation**, **brightness**.

In case there are n objects of the same type $a_1..a_n$ and only one method $f()$, how can one ensure that a method is invoked on a particular a_i ? We must ensure, in compliance with the object-oriented paradigm, that a message should be sent to an object. The keyword **this** is an identifier denoting the current object – the particular one to be manipulated.

Many programmers always use **this**, when referring to member variables of the current object – to make the code more explicit.

Let us take two objects, a and b , and assume that we need to call some method only on one of the two, say on a . Let, for more concreteness, a , $b,..$ be some geometrical objects and let the method be their area calculation. Then we may write:

```
float area = this.calcArea ( );
```

Such style helps us to indicate which object is involved in the invocation of the method. One may note, however, that the style

```
float area = calcArea ( );
```

is also admissible in this case.

A more important utilization of the keyword **this** may be observed when it is necessary to use the same names for variables or arguments to the constructor (or to any other method) and for field names – parameters. For example, let us consider again a point in 3D space:

```
class ThreeDPoint {
    double x;
    double y;
    double z;
    ThreeDPoint (double x, double y, double z) {
        this.x = x;           // we want to emphasize that
                             // the variables of the current
        this.y = y;           // object are being used, i.e.
                             // x = x + change
        this.z = z;
    }
}
```

```

String getString ( ) {
    return "("+ String.valueOf(this.x) +",""+
        String.valueOf(this.y) +",""+
        String.valueOf(this.z) +")";
}

/* setting coordinates; such style is useful when the names of the
method parameters coincide with those of field (instance) variables.
Here x,y,z mean parameters and this.x, this.y, this.z are instance
variables */

public void setX (double x) {
    this.x = x;
}
public void setY (double y) {
    this.y = y;
}
public void setZ (double z) {
    this.z = z;
}
public double getX( ) {
    return this.x;
}
public double getY( ) {
    return this.y;
}
}
}

```

- **Commentary:** The class body contains the declarations for, and possibly the initialization of all data members (both *class variables* and *instance variables*) as well as the full definition of all *methods*. It might happen that declaration of a variable or argument within a method with the same name as a field (instance variable) hides or “shadows” that field. One still can bypass local variables or parameters that hide member variables having the same name, in order to access the member variable. One can refer to *instance variables* by prefixing the respective field by the keyword *this*. The reference named **this** is a reference to the object on which the instance method was invoked.

Hidden member variables

Since the usage of the **this** keyword is rather ubiquitous, we shall try to discuss it slightly more in detail. Usually, instance methods belonging to an object have direct access to the instance variables belonging to that object, and to the class variables belonging to the class from which that object was instantiated. (We must recall in passing that class methods never have access to instance variables or instance methods.)

However, the name of a method parameter or of a constructor parameter can be the same as the name of an instance variable belonging to the object or a class variable belonging to the class. It is also admissible for the name of a local variable to be the same as the name of an instance variable or a class variable. In this case, the local variable or the method parameter is said “to shadow” – to hide - the member variable having the same name. So, the problem stems from the fact that names can be duplicated.

To illustrate this point, let us create a sample program discriminating all types of variables and using the keyword `this` to access hidden member variables. The reference keyword **this** is used to bypass the local variable `xX` and the parameter `yX` and to access a hidden class variable `xX` and a hidden instance variable `yX`.

The output from this program should be:

```
yX parameter = 25
local xX variable = 4
Instance variable yX = 5
Class variable xX = 15
*****/

class ThisTest {
    int yX = 0;
    static int xX = 0;

    //Constructor with parameters named
    // yX and xX
    public ThisTest(int yX,int xX){
        this.yX = yX;
        this.xX = xX;
    }//end constructor
    //-----//

    //Method with parameter named yX
    // and local variable named xX
    void myMethod(int yX){
        int xX = 4;
        System.out.println(
            "yX parameter = " + yX);
        System.out.println(
            "local xX variable = "
                + xX);
        System.out.println(
            "Instance variable yX = "
                + this.yX);
        System.out.println(
            "Class variable xX = "
                + this.xX);
    }//end myMethod
    //-----//

    public static void main(
        String[] args){
        ThisTest obj = new ThisTest(5,15);
        obj.myMethod(25);
    }//end main method
} //End ThisTest class definition.
```

The important points

Each time an instance method is invoked, it receives a hidden reference named **this**. The latter is a reference to the object, on which the method was invoked.

One may observe the following:

- When the code refers to **yX** or **xx**, the reference results in either an incoming parameter or in a local variable having that name.
- When the code refers to **this.yX** or **this.xx**, the reference results in the corresponding instance variable and class variable having that name.

To summarize this situation, we may note that the code in the method can use such reference to access any instance member of the object on which it was invoked, or any class member of the class from which the object was instantiated.

However, one must also note that when class methods are invoked, they cannot refer to any instance members of any object instantiated from the class. They can only access class members of the same class.

Inheritance in Java programming

We have already discussed the principle that OOP classes tend to be defined in terms of each other. This is ensured due to a mechanism called inheritance. In order to see how this mechanism works in the Java programming practice, let us start from an example. We can create a piece of a source code modelling various media: books, CD, DVD, video, etc. For musical part of the media, we might have

```
public class Media
{
    private String title;
    private int playTime;
    private boolean haveIt;
    private String comment;
    // methods and constructors omitted
}

// Subclass DVD:
public class DVD extends Media
{
    private String singer;
    private String orchestra;
    private int numberOfSongs;
    // methods and constructors omitted
}
```

We see that in the DVD class only the fields unique to DVD objects are defined. Nevertheless, the latter have fields for **title**, **playtime**, etc. inherited from the superclass **Media**.

Now, if we would like to include one more subclass of media, this time video, we can write a piece of code for the Video subclass. The latter piece of code follows the same pattern as the DVD class:

```
public class Video extends Media
{
    private String leadingActor;
    //methods and constructors omitted
}
```

How to declare in practice that the newly created class is a subclass of another? One may notice here that the **extends** keyword assists in the definition of a subclass, and its own additional fields appear. If you want to create a subclass – simply include the clause **extends**

in the class declaration. The meaning is that a subclass is a class that extends another class, i.e. a subclass inherits the state and behavior from all its ancestors.

From this example we can deduce the general pattern:

The **extends** keyword is used in the class declaration to specify the immediate superclass of the newly created class

```
class NewClass extends SuperClassName {  
    //body of class  
} //end class definition
```

Here we may reproduce an example given by D.J. Barnes and M.Kölling in their very interesting and useful book “Objects First with Java”. We know that a poodle is a dog, which is a mammal, which is an animal. We know some things about poodles - they are alive, agile, they can eat, bark, reproduce themselves – not because they are poodles, but because they inherit all these characteristics from superclasses: dogs, mammals, etc. Thus, a class inherits the variables and methods of its superclass, and of the superclass of that class, etc., all the way down the hierarchical tree to the single class **Object**, which is the root of all inheritance.

Now, let us see what happens to an object that is instantiated from a class within the inheritance framework. Such object contains all the instance variables and all the instance methods defined both by that class and by all its ancestors. However, the methods may have been changed or overwritten along the way. Therefore, access to those variables and methods must be restricted through the use of modifiers - **public**, **private**, and **protected** keywords. Thus, one must always pay attention to access modifiers. Members defined as **private** will be inaccessible to other classes. The same privacy rules regulate relations between derived classes: a subclass cannot access or change private fields of its superclass.

On the other hand, a subclass may call any **public** methods of its superclass as if they were its own – no additional variable is needed.

Let us consider another example also demonstrating how inheritance works. Here we shall see that classes can be specialized due to inheritance. For instance, we may consider the class **Student** as a special case of the class **Human**, which means that in addition to e.g. the attribute name, the class **Student** possesses further attributes such as matriculation number and curriculum. Java makes such specialization possible with the help of the **extends** keyword.

```
public class Student extends Human  
{  
    /* declaration of attributes */  
    private int matrNumber;    // matriculation number  
                                // as an integer  
    public String curriculum; // curriculum as String  
                                // variable  
  
    public Student(int MatrNumber, String Name, String Curriculum)  
    {  
        super(Name);          //parameter "name" will be given by
```

```

                                // a special constructor
    matrNumber = MatrNumber; // the parameters (beginning by a
                                // capital letter) are assigned
                                // to class variables (small letters)

    curriculum = Curriculum;
}
}

```

We have already mentioned that every class in Java by default extends - directly or indirectly - the class named **Object**. A new class may either extend **Object**, or extend another class that extends **Object**, or extend another class down the inheritance hierarchy.

Java programming examples

Let us write the source code for a Java application that will display your name and address on the standard output device. We must show the command-line statement that would be necessary to execute a compiled version of such application.

```

/* File Name.java
This is a Java application that will display a
name on the standard output device.

The command required at the command line to execute this
program is:

java Name */

class Name { //define the controlling class

    public static void main(String[] args){ //define main

        System.out.println(
            "MyName\ The Technical University of Munich\85748,
            Garching, Germany");

    } //end main

} //End Name class

```

Here we see once again that the method `main` is considered “an entrance point” to a Java program. Let us now see how we can create objects, e.g. new persons:

```

class Human { //define the controlling class

public static void main(String[] args) //define main
{
    //let us create an object of the class Human
    Human peter = new Human("peter");

    //the method getName() of the object 'peter' is called
    // and returns the value
    System.out.println(peter.getName());

    // the attribute 'name' of the object 'peter' is set
    // through calling the method setName
    peter.setName("peter von blabla");
}
}

```

```

// another object, this time of the class Student,
// is created
Student.paul = new Student(555, "paul", "mathematics");
// here 555 is the student matriculation number;

// again, calling the appropriate method we get
// the result
System.out.println(paul.getName());
}

```

The Java model of a university

Let us try to produce a code for an abstract university consisting of several institutes (departments), each institute in each turn consists of chairs, each chair being characterized by its owner (head). For concreteness, we used names and addresses of the Institute of Mathematics and of the Institute for Computer Science (Informatics) of the Technical University of Munich. In this example, besides the inheritance, we may see how the “get” and “set” methods work.

```

/* Creating a class „University” - declaring the class*/
public class Uni
{
/* Declaration of attributes */

    protected String name; //declaration of the attribute „name”

/*Declaration of methods */

    public Uni (String the_name)
    {
        name = the_name;          // the value of the method parameter
                                  // is assigned to the class variable
                                  // „name”
    }

    /* Methods contain statements; initially we declare only methods
    containing simple statements, e.g. printing*/

    public String getName ()
    {
        return name;              // the method simply returns the value
                                  // of the class variable
    }

    public void setName (String the_name)
    {
        name = the_name; //one more assignment
    }
}

```

```

/* So far, this is an abstract class - no object has been created yet */

/* Creating the class "Institute" -
   Class declaration */

public class Institute extends Uni
{
/* Declaration of attributes */

    public String address; // attribute address is declared

    public Institute (String Name, String Address)
    {
        super (Name);
        address = Address;
    }

/* Declaration of methods */

    public String getAddress ( )
    {
        return address; // the method simply returns the
                        // value of the class variable
    }

    public void setAddress (String the_address)
    {
        address = the_address; // one more assignment
    }
}

/* Class Institute is a subclass of Uni and a superclass for Chair (class
Lehrstuhl) */

/* Class „Chair“ (Lehrstuhl)declaration */

public class Chair extends Institute
{
/* Declaration of attributes */

    public String head; // declaration of the attribute
                        // "head"

    public Chair(String Name, String Address, String Head)
    {
        super (Name, Address) ;
        head = Head ;
    }

/* Declaration of methods*/

    public String getHead ( )
    {
        return head; // the method simply returns the
                    // value of class variable
    }

    public void setHead (String the_head)
    {
        head = the_head; //one more assignment
    }
}

```

```

public static void main (String [ ] args)
{
    Uni tum = new Uni ( "TUM");

    System.out.println (tum.getName ( ) );

    Institute inst1 = new Institute
        ("InstituteMath", Boltzmannstr3");

    Institute inst2=new Institute
        ("InstituteInfo", "Boltzmannstr3");

    System.out.println (inst1.getName ( ));
    System.out.println (inst2.getName ( ));
}
}

```

Class diagrams

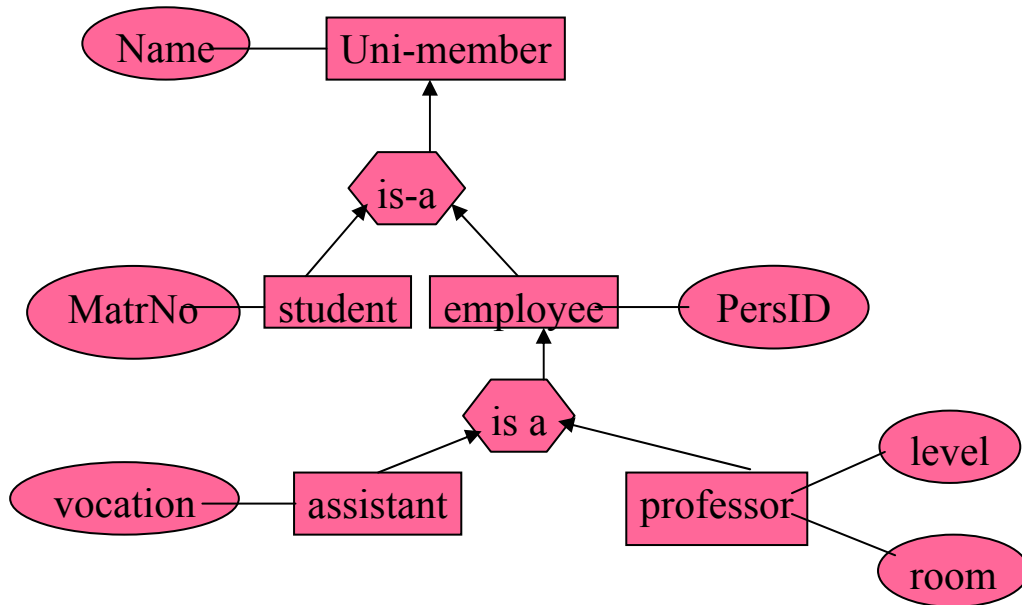
There exist standard methods of graphically depicting interrelationships within programs. Such methods are very helpful for understanding complicated program listings. The standard methods of graphical representation of interrelated classes and objects are especially efficient for object oriented software design and engineering. In fact, there are many types of diagrams used for graphical representation of complex systems, but the so called *class diagrams* are of primary importance.

Class diagrams describe the structure of a system in terms of classes and objects. Recently, a special graphic language have been developed called UML (Universal Modelling Language), where the rules of diagram construction have been formalized, with all elements, sequences, links, associations, aggregations, etc. to become standardized. One can find a comprehensive description of the UML with all the rules of diagram construction in the book:

B. Bruegge, A.H. Dutoit. Object-Oriented Software Engineering, Prentice-Hall, 2000.

We can draw a rather primitive class diagram (not complying with the UML rules) for a model of the university. For example, such class diagram may look like the following one:

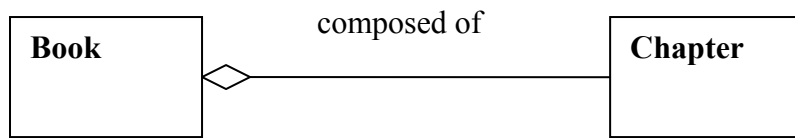
Entity Relationship



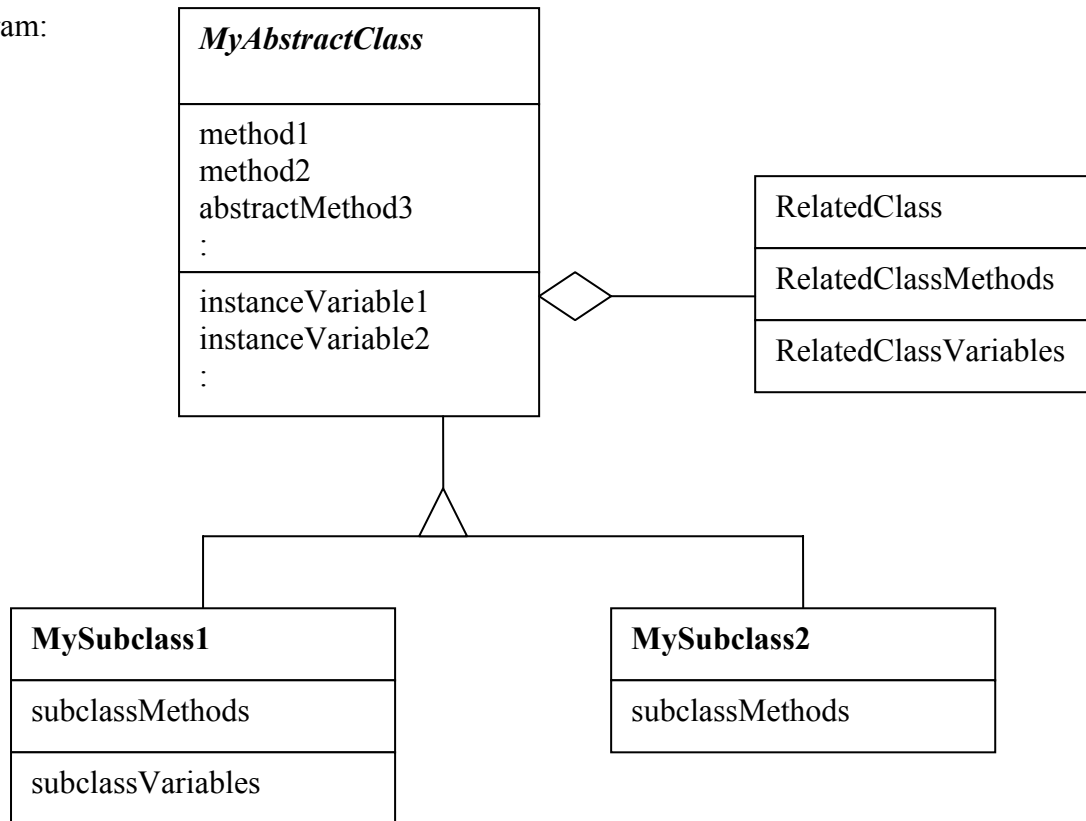
Standard class diagrams are drawn using the UML conventions. A class is represented by a rectangular box with two or three compartments. The top part contains the **classname**, usually typed in bold. If the class is an abstract class, the *classname* is typed in bold italics. The lower part of the class box holds a list of public object methods. The name of an *abstract method* is written in italics. The lowest – the third – compartment of the class box, if any, contains the list of all instance variables. It may happen that the class does not have any instance variables; in this case the class box consists of only two parts.

A vertical line with a triangle indicates class inheritance. Different sources give discordant views as to where this triangle should be located; probably the standard patterns have not been fully established yet. The traditional inheritance relationship is depicted as the line with a triangle pointing to the superclass, with the free end of the line being attached to the subclass. Another version is shown at the picture below. If there exists a number of subclasses, then the inheritance line in this version branches at the triangle.

A horizontal line beginning with a diamond, which is the UML aggregation symbol, indicates a one-to-many association – a composition. The statement “book contains many chapters” is displayed as



A typical class diagram:



Why do we need different access rights?

Let us consider an example: the bank account. For the latter, we can write the following piece of code:

```

public class Konto
{
    private int    number;
    private double balance;
    public void draw(double credit)
    {if (balance > credit)
        balance = balance - credit;
    }
}
  
```

We see from this example that internal data can be used only within the **Konto** class. On the contrary, **public** elements can be accessed from any class. What is the reason for such restriction? The matter is that if indiscriminately all members of the class **Konto** were

available to anyone, then one could do anything with that class, and there would be no way to prevent stealing and, in general, to enforce rules of conduct in the bank. The other reason has already been mentioned and it is data hiding: one can make changes within the class without much care of how it would affect the user. Internal working and user interface are clearly separated. From here one may deduce an advice: things should be kept as private as possible (if not in real life, but definitely in object oriented programming).

How to use operators in Java

In general, operators are defined as a map of one set onto another set, each of the sets being endowed with a certain structure, e.g. algebraic operations, ordering, topology, etc. In programming languages, operator is a grammar construct expressing some finalized action on an object, i.e. operators produce a new value of their operands. Such binary operators as addition (+), multiplication (*), subtraction (-), division (/), assignment (=) have the same meaning in all programming languages

In Java, nearly all operators work only on primitives. The exceptions are '=', '==' that work with all objects; the '+' operator is supported in the **String** class

The order of applying operators – precedence: rules in Java are similar to the general arithmetic, e.g. $Z = x + y - 5/6 + A$ differs from $Z = x + (y - 5)/(6 + A)$.

This example shows that brackets (or parentheses) are as important in Java as in any other programming language.

Another example:

```
(int) x + 9.5; // here the operator is applied only to x
```

In programming, a very common operation is to add 1 to a counter, for example:

```
int sum;
sum = sum + 1;
```

This statement is an assignment (see below), and it seems superfluous to type the variable name twice in every case, especially in large chunks of code. To abbreviate usage of increments and decrements, two special operators have been defined:

Increment and decrement operators	
Variable ++;	// add 1 to the variable
Variable --;	// subtract 1 from the variable

Using these operators, we may write for the previous case:

```
sum ++;
```

which is, in fact, a shortcut.

The assignment operator

If we have an expression $a = b$, what is the meaning of the equality sign?

In a number of programming languages including Java, the symbol = in the above statement has the following meaning: take the rvalue (right hand side, rhs) and copy it into the lvalue (left hand side, lhs). The rvalue may be any variable, expression or constant, whereas the lvalue must be a declared variable. Take, for example, the statement “ $X = 5$ ”; this statement is allowed, while the one “ $5 = X$ ” is forbidden – one cannot assign anything to a constant value. Why? Because there must be some physical space allotted to store a value. This interpretation of the equality sign distinguishes assignment in programming from that in mathematical expressions.

For primitive data types, the statement $A = B$ means that the contents of B is copied into A. For objects, it’s a different story: when one object is assigned to another, it is the reference and not the actual value that is copied from one place to another.

In principle, the result of the assignment statement is to evaluate the expression to a variable. The equality sign, which indicates such assignment, can be understood as “becomes”, e.g.

```
temperature = 100;
```

may be read as “the temperature becomes 100”. The value of the expression is 100 and this value is assigned to the variable **temperature** (in the programmers’ argot one is used to say that the variable “sits down” on a certain value).

Let us look at the general form: how we evaluate the expression and assign the resulting value to the variable? For instance, consider the atmospheric pressure and its normal value. In this case we may write the following statement: **pressure** = 762. An important principle here is tacitly conveyed, namely assignment requires that the variable type and the expression type should be the same. One cannot, for example, assign a **double** to an **int**, or something of the **int** type to a **Human**.

One more type of the assignment includes an operator and may be used as a shorthand trick, when the value of the variable is the first operand. An example:

```
i += 2;
```

that will add 2 to *i*. The term ‘shorthand’ is used here to emphasize that this expression produces exactly the same result as

```
i = i + 2;
```

but is shorter and is considered to economize resources in long coding samples. One must remember that the increment operator is += and not =+. The best way to remember this fact is probably to realize that the latter operation is an assignment and therefore could be confused with assigning a positive value to a variable and not adding this value, for instance the statement:

```
sum = + 10;
```

will handle +10 as a signed constant that should be assigned to the variable **sum**.

The next coding example also demonstrates a convention generally used in programming in connection with assignments:

```
oldWeight = 75;
```

```
newWeight = oldWeight + 10;
```

This statement may be interpreted as follows: `oldWeight` is assigned the value 75 and is subsequently used to calculate the value of the variable `newWeight`.

Arrays

Before we proceed to numerical methods, we need to discuss several important concepts needed primarily to use programming languages for numerical applications. One of these concepts is **array**. An array is a collection of variables of the same type, indexed by numbers. Usually, for mathematical applications, an array is understood as a special type of collection that can store a fixed number of elements. A matrix, for example, is an array of this kind. However, several other array types are also possible, e.g. an array of string literals like ["to", "be", "or", "not", "to", "be"]. In the **main** method the notation `args []` designates an array of **Strings**.

In computer science in general, the indexed objects that mathematicians extensively use, e.g. vectors, tensors, matrices, etc., are called arrays. Since indices are not available on a keyboard, they are ordinarily placed in square brackets. Thus, instead of defining a vector

$$x = (x_1, x_2, \dots, x_n),$$

a computer scientist will consider an array `x [1..n]` whose elements are `x[1]`, `x[2]`, Likewise, a matrix is an array with two indices of the form `A[1..m, 1..n]`, the matrix element usually denoted as $A_{i,j}$ in mathematics is written as `A[i,j]` in computer science. Of course, arrays with 3,4,..N indices may be used.

Apart from arrays containing fixed number of elements (fixed-type arrays), there may be collections with undetermined number of elements (flexible-type arrays). In Java, arrays are understood as being only of a fixed type. They can store both objects and primitives, whereas flexible-type collections in Java store only objects. For arrays, Java uses the syntax inherited from other programming languages, and the access to items held in arrays is readily supported in the Java syntax – in distinction to invoking the methods. Individual elements of an array can be accessed simply by indexing. An index in Java is an integer expression put between square brackets that follow the name of the array variable. For example,

```
students [a +10 - b];  
institutes [10];  
labels [5];
```

etc.

It is important to remember that arrays indices in Java always start at zero and increase to (n-1), where n is the array's length. For instance, the clock array would have valid indices not from 1 to 24 but from 0 to 23 (all inclusive). It is a common mistake to assume that the valid indices of an array start at 1 and to use the length of an array as an index. The length of the array `n` lies outside the array in Java and using it as an index produces a compiler error named `ArrayOutOfBoundsException`.

How to declare array variables?

An array is typically declared with the keyword `new`, for example:

```
Institute [ ] instituteArray = new Institute [10];
```

Here an array of ten objects of the class `Institute` is declared. The type of the variable `instituteArray` is `Institute []`, with the array dimension being equal to 10.

All elements of an array must be of the same type. One can iterate over the array's elements using an index in **for** or **while** loops (see below).

The general construction of array definition:

```
type nameArray [ ] = new type [limit];
```

is quite similar to object instantiation, except that `[]` before the assignment denote the identifier as an array. The first part of the definition specifies the type of the array. After the type the limit is indicated and not the initial parameters as in the case of objects. The style suggesting that the type name (e.g. `int`) is followed by a pair (it may be an empty pair) of square brackets is intended to indicate that one is declaring an array rather than a single variable of the type `int`. The part following the equality sign defines the array. The keyword `new` is to point out that new memory should be allocated for the array, and type `[limit]`, e.g. `int [10]` is here to specify that we want the memory capacity for 10 variables of the `int` type in the newly created array. If elements of the array are of `int` type requiring 4 bytes, the whole array would occupy 40 bytes (plus 4 bytes to store the reference to the array).

Another example of array declaration

```
private int [ ] clockCount;    /* here the clockCount
                               is a variable of the
                               base type integer */
```

is just **a declaration** for the array name – not **a definition** of an array. No memory has been allocated to store the array itself and the number of elements has not been defined. On the contrary, the statement of the preceding kind, namely

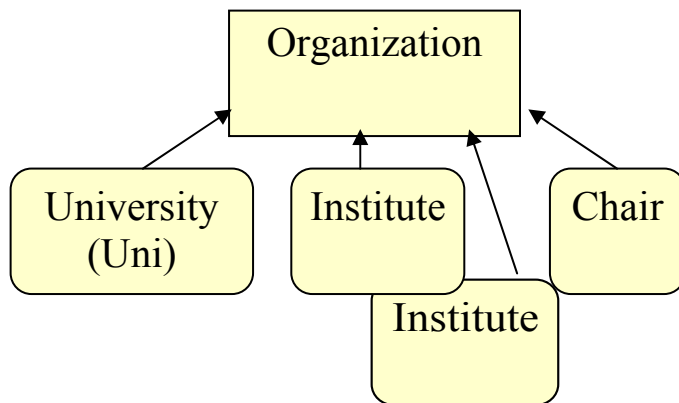
```
double [ ] nameArray = new double [10];
```

is, besides a declaration of the array variable `nameArray`, a definition of the array, since the array size is specified. The variable `nameArray` will refer to an array of `[10]` values of type `double`. So one should be careful in using the terms “to define” and “to declare” interchangeably. One can also note that each element of the array will have the value 0.0 assigned by default. In general, if the array contains numerical values, the initial value is zero, and false for boolean arrays.

Code example: array of institutes

We have seen above an example of a class diagram related to a simple model of a university. Here we can produce a still more primitive class diagram. Let us try to describe the relationships between the following classes: `University`, `Institute`, `Chair`, taking for beginning an ultimately simplified model

A simplified model of the University



All the organizations inherit their properties from the abstract Organization class

Here, the extension of functionality can be traced. Several method attributes (parameters) are called, some of them are inherited from the superclass, others are new. Parameters are separated by commas inside the round brackets.

The Uni class is characterized by its President, the Institute – by the Dean, the Chair – by the latter Head

```
/*Class declaration*/
public class Institute extends Organization
{
    /*declaration of attributes */
    public String dean; //declaration of the attribute "dean"

    /*declaration of the array Institites*/

    public Institute(String Name, String Address, String Dean)
    {
        super(Name,Address);
        dean=Dean;
    }

    /*Declaration of method */
    public String getDean()
    {
        return dean; //the method simply returns
                    //the value of the class variable
    }
}
```

```

    public void setDean (String the_dean)
    {
        dean = the_dean; //one more assignment
    }
}

```

The Organization Class

```

/*Class declaration*/
public abstract class Organization
{
    /* Declaration of attributes "Name,Address" */
    protected String name,address;

    /* Constructor */
    public Organization(String the_name, String the_address)
    {
        name = the_name; //Assignment of the value of the method
                        // parameter to the class variable
        address = the_address;
    }

    /*Method declaration */
    public String getName()
    {
        return name; // the method returns the value of
                    // the class variable
    }

    public String getAddress()
    {
        return address; //The method simply returns
                        //the value of the class variable
    }

    public void setName(String the_name)
    {
        name = the_name; //Assignment
    }

    public void setAddress(String the_address)
    {
        name = the_address; //Assignment
    }

    public static void main(String[] args)
    {
        /*creating the object Uni with 3 parameters, the
        parameters Name, Address are called through the
        constructor Organization, and the parameter President is
        described in object Uni */

        Uni tum = new Uni("TUM","BoltzmannStr,3", "Herrmann");
        System.out.println(tum.getName());
        System.out.println(tum.getAddress());
        System.out.println(tum.getPresident());
    }
}

/* Let us now introduce the array of institutes by inserting the class Uni
derived from Organization*/

```

```

/*Class declaration*/
public class Uni extends Organization
{
    /*Attribute declaration */
    protected String president; //Declaration of the attribute
                                //"president"
    private Institute institutes[] = new Institute[10];
    private int inst_counter = 0; //points to the last occupied index
                                //in the array "institutes"

    /*constructor*/
    public Uni(String Name, String Address,
               String President)
    {
        super(Name, Address);
        president = President; /* The value of the method
                                parameter is assigned to
                                the class variable "president" */

        /*two objects of Institute are created*/
        Institute inst1= new Institute("Institut fuer Informatik",
                                       "Bolzmmanstr3", "Mayr");
        Institute inst2= new Institute("Institut fuer Mathematik",
                                       "Bolzmmanstr3", "Scheurle");

        /* Adding these two objects to the array of institutes */
        addInstitute(inst1);
        addInstitute(inst2);
        printInstitute();

        /* Calling the method addChair_Uni*/
        addChair_Uni("Broy", "Institut fuer Informatik");
    }

    /* Method declaration */
    public String getPresident()
    {
        return president; //The method simply returns
                           //the value of the class variable
    }

    public void setPresident(String President)
    {
        president = President; //Assignment
    }

    public void addInstitute (Institute inst)
    {
        institutes[inst_counter]=inst; // adds the institute "inst"
                                        // in the array

        inst_counter++;                // increments the "institutes" counter
    }

    public void printInstitute()
    {
        int i;
        for(i=0; i<inst_counter; i++){

            System.out.println("Name of Institute: "
                               +institutes[i].getName());
        }
    }
}

```

```

        System.out.println("Address of Institute: "
                           +institutes[i].getAddress());
        System.out.println("Dean of Institute: " +institutes[i].getDean());
    }
}

```

One can proceed with an array of chairs in the same way.

Arrays of arrays

Arrays discussed above are one-dimensional arrays; they always use a single index (one-dimensional map of integers). However, in scientific applications multidimensional arrays are not less common. If you obtain – by measurement or numerically - the series of values for some physical parameter in m independent sessions, you will have a two-dimensional array. For instance, measuring the everyday temperature in 10 different points would produce an array of 365 elements for each location combined with an array of 10 independent sites. This combined array can be declared using the following statement:

```
float [ ] [ ] temperature = new float [365] [10];
```

This is a typical two-dimensional array, with one index - related to investigation sites - running from 0 to 9 and another one, corresponding to the day of the year, running from 0 to 364. An alternative would be to have a one-dimensional array containing 3650 elements.

Many examples of multidimensional arrays are produced by medical observations, when many parameters characterizing a patient's state have to be taken simultaneously for a number of patients. For demonstration purposes, we can produce here a coding example for a medical application, recording a blood pressure in a group of 10 patients observed during a month. In fact, there are two blood pressure values – systolic (upper pressure) and diastolic (lower pressure), but here, for brevity, we shall consider only the upper value ranging from 100 to 200. Since we do not carry out real measurements, we can generate the blood pressure data using the `Math.random ()` method, frequently employed in Java programming practice.

```

public class BloodPressure
{
    public static void main(String [ ] args)
    {
        // blood pressure array
        float [ ] [ ] bloodPressure = new float [10] [30];

        // generate blood pressures
        for (int i = 0; i < bloodPressure.length; i++ )
            // outer loop refers to the number of patients

                for (int j = 0; j < bloodPressure[i].length; j++)
                    // inner loop refers to observation days

                        bloodPressure[i] [j] =
                            (float) (100.0*Math.random ( ) + 100.0);

        // calculate the average per person
        for (int i = 0; i < bloodPressure.length; i++)
            // iterates over patients
            {

```

```

float average = 0.0f;
// place to store the average

/* Before executing the inner loop, the
   variable average is declared and
   initialized. The sum of blood pressure
   values is accumulated for each patient in
   the inner loop*/

for(int j = 0; j<bloodPressure[0].length; j++)
// inner loop sums up

average += bloodPressure [ i] [ j];
// accumulate blood pressure values for current
// patient

/* Output the average bloodPressure for the
   given patient */

System.out.println("Average blood pressure for
a patient" + String.valueOf(i + 1) + " =
" + String.valueOf(average/(float)bloodPressure
[i].length));

/* To get the average, the variable
average is divided by the number of
samples, which is bloodPressure [i].length.
Here the array holds blood pressure
values for each patient and the array
length is the same for all the
patients, so any index value can be
used */

    }
}

```

The `Math.random ()` method is a typical representative of the Java packages, which we shall discuss later. This method generates a value of the double type between 0.0 and 1.0. In our example, this value is multiplied by 100 and 100 is added to achieve the required blood pressure range.

In this example we used a very important programming construction called “looping”. Loops are ubiquitous in computer science, but they are of ultimate significance in numerical modeling. In fact, a majority of numerical models are reduced to loops when being programmed in Java. That is why to prepare ourselves to numerical methods, we must now focus on loops.

Repetition and loops

All the machines including computers are well-suited for performing the same operation over and over again. Computer, in general, obeys a sequence of instructions. In programming techniques, such repetitive activity is formulated in terms of **loops**.

An example is an iterative solving of mathematical equations, repeatedly obtaining better approximations. This procedure is always performed with loops. Another example is looking

through files for some needed information. In graphical applications, making a figure move on the screen (animation), is also performed with loops.

There are two kinds of loops being considered in Java (as in most programming languages): **conditional loops** and **counting loops**. In the preceding example, we resorted to the *counting loops*. The latter are introduced in Java (as well as many other languages) by the keyword **for**, and are therefore commonly nicknamed **for-loops**. A pattern for a Java for-loop statement takes the following form (see the above example):

Basic for statement

```
for (begin; check; update) {  
    body of the loop;  
}
```

The keyword **for**, brackets and the body of the loop are indispensable for shaping the loop structure. The **begin** part introduces the loop variables (there may be more than one of them). These loop variables must be tested and updated. In Java, if there are several loop variables, they should be *separated with commas*. The **check** part holds a comparison that is intended to provide a condition for the loop to be terminated; this part takes the current value of the loop variables and tests it against some prescribed boundary value, in many cases the array length. The **update** portion consists of assignments (see the respective section above) intended to modify the values of loop variables, iterating each time round the loop. It is exactly because of such repetitive iteration that **loops** have acquired their name. The body of the loop includes some statements; if there is only one statement, then the curly brackets can be omitted.

In the programming language semantics, loops are manifestations of so called *control structures*, the loop variable being called control variable. This name stemmed from the fact that in order to execute a loop, the program must evaluate once and for all the minimum and maximum bounds, controlling that all the changes should occur within these limits. If the loop variable transgresses the prescribed bounds, nothing happens – the program skips the loop and proceeds to the statement that follows (if it exists), so the loop variable remains unchanged.

In most cases, there is only one loop variable. Such variables in Java must be numbers, as a rule integers - although there is no formal prohibition to use, for instance, real numbers. However, using real numbers as loop variables may be considered a bad programming habit, since comparison of reals is principally inexact at limits. So, an unnecessary complication arises, whereas it is a good practice in general to write for-loops as simple as possible.

In connection to arrays in Java, it is important to remark that if we want to run a loop n times, we may choose between two versions:

```
for (int i = 1; i<=n; i++)  
and  
for (int i = 0; i<n; i++)
```

The first statement probably looks more intuitive, however since the arrays in Java use zero as a starting point, the second form is definitely preferable.

To get trained in for-loops, let us consider the simple example. Suppose that we need to produce six rows of sharp symbols, i.e. #####. Then we may write:

```
for (int i = 0; i<6; i++)
    System.out.println ("#####");
```

How can we prove that the program will print out exactly six rows? First of all, we can see that the loop variable *i* takes up the values 0 through 5, but not 6. Furthermore, once the initialization has been completed, the sequence in which the rest of the loop should be executed is 1) check; 2) body; 3) update. More specifically; *i* starts at 0 and is tested against 5, then a string of sharps is printed and *i* is incremented to 1. The process goes on, and when the loop variable reaches 5, it is checked against 6 and more sharps are printed. Then *i* is incremented to 6 and the check against 6 fails. The loop ends, and therefore we can have exactly 6 printed out lines of sharps. In the same fashion, we can organize the inner loop, which would print the #-symbol within a line (in this example, exactly 10 symbols).

Remarks – backward and empty looping.

One can also organize decrementing loops – descending down to the minimal bound.

Loops can also happen to be empty, when the starting limit already exceeds the terminating one, for example

```
for (int j = 0; j < finish; j++)
    System.out.print (String.valueOf(j)+" ");
System.out.println( );
```

In case finish is non-positive, only a blank line will be produced.

Loops can be used in simple arithmetic. For instance, let us see how we can produce a sequence of the first ten even numbers. We can write the following piece of code:

```
for (int number = 0; number < 10; number++)
    System.out.println (String.valueOf(number*2) + " ");
```

which would produce the sequence

0 2 4 6 8 10 12 14 16 18

Another version would be to explicitly incorporate even numbers in the update part:

```
for (int number = 0; number < 20; number+=2)
    System.out.println (String.valueOf(number) + " ");
```

To print the first ten odd numbers, we can write:

```
for (int number = 0; number < 10; number++)
    System.out.print (String.valueOf (number*2 + 1) + " ");
System.out.println( ); /* here we separated the body
                        from the output command */
```

This code fragment would produce the sequence:

1 3 5 7 9 11 13 15 17 19

In the example of the preceding section, we have the so called nested loops, i.e. loop within a loop. This concept implies that the body of a loop also contains a loop. In such cases, it is important to avoid confusion, so the loop variables should be different – kept local.

Conditional loops

This second kind of loops is formed in terms of while statements. The general pattern of while-loops looks as follows:

Basic while statement

```
while (conditions) {
    statements /* statements to execute the loop and
                modify the conditions */
}
```

The loop begins by checking the conditions and is executed only when the logical expression in the parentheses is evaluated as **true**. When this expression is **false**, the loop statement is ignored and the entire loop block is not executed. In the **true** case, as soon as the loop reaches its end, the program returns to the beginning of the loop and the conditions are tested again. This process is repeated until the conditions check is evaluated to **false**, then the loop stops, and the program execution passes to the statement following the loop block. As an example, let us try to write a program summing up integers:

```
public class SumLoop
{
    public static void main (String[ ] args)
    {
        int limit = 25; // sum from 1 to this value
        int sum = 0; // accumulate sum in this variable
        int j = 1; // loop counter

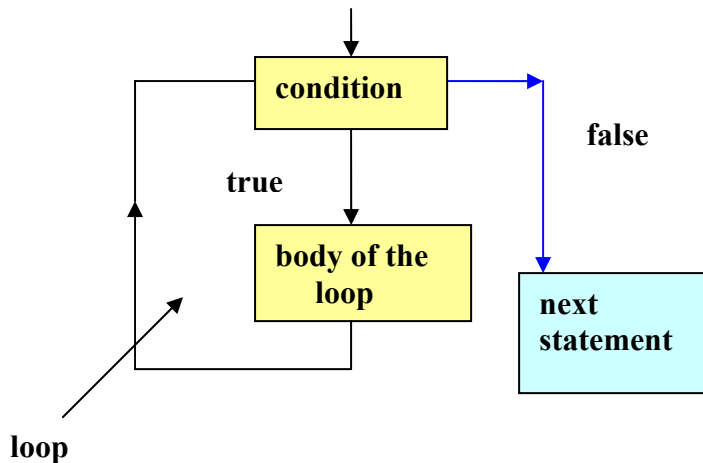
        /* loop from 1 to the limit value, each cycle adding 1 */
        while (j <= limit)
            sum += j++; // the current value of j is added to
                        // sum
        System.out.println(" sum = " + sum);
    }
}
```

The computation should give the result:

```
sum = 325
```

In this example we have a simple count (as before), and the loop variable is incremented in the loop statement, which results in the accumulation of the sum of integers. One needs to make sure that the statement within the loop would eventually evaluate the condition as **false**, otherwise the loop never ends (infinite loop). In our example, *j* will finally exceed the value of limit, and the **while** loop stops. Here we used *j* for a loop variable, although the general convention is that of using *i*.

Schematically, the **while** loop can be represented as



Conditional statements

The most widely used conditional statement in Java is the **if**-statement, which is also known as selection statement. The **if**-statement is extensively used in practically all programming languages, having the form:

```
if x == 0 then y = y + 1;
```

which means that if the contents of the variable x is 0, then y is incremented. In the opposite case, i.e. if $x \neq 0$, nothing happens and the computer executes the next statement (in case there is one). An example of the conditional statement with the **else** keyword:

```
if x == 0 then y = y + 1 else z = z - 1;
```

Please note that it is easy to confuse the assignment operator “=” with the boolean equality operator “==”. They have different meaning: the symbol “=” performs an assignment to the right hand side, whereas the operator “==” compares the right and left hand sides and returns “true” or “false”.

If the content of x is zero, y is incremented and the program skips the rest of the statement, so the value of z is not changed. In the opposite case, if x is non-zero, then the program ignores the beginning of the conditional statement and the variable z is decremented, whereas the value of the variable y is not changed. We may note that the conditional statement in this case involves three operands, therefore the conditional operator **if** is sometimes called a ternary operator. In Java, the conditional operator **if** mostly uses boolean data type (as the content of the first operand), that is:

```
boolean_expression ? expression_1 : expression_2
```

If the **boolean_expression** evaluates to **true**, the result of the conditional operation will be the value of **expression_1**, otherwise (**false**) the result is the value of **expression_2**

One may note that if **expression_1** is produced because **boolean_expression** gives **true**, then **expression_2** is ignored and vice versa. In the programmers' terminology, **expression_1** is frequently called the **then-part** and the **expression_2** (i.e. the statement following the **else** keyword) is called **else-part**. The general form of the conditional if-statement in Java is

Conditional if-statement

<pre> if (condition) statement; else statement;</pre>

A simple example:

```

if (number >=0)
    System.out.println("positive")
else
    System.out.println("negative")
```

The do-loop

The general form of the **do** statement is inherited from procedural languages. The **do-loop** begins by running through the loop body at least once *before* checking the conditions. In general, one can use the **while** statement instead of the **do-loop**. Below you will see the pattern of the **do-loop**

Do statement

<pre> Initialize the conditions do { Statements to execute the loop and change the conditions } while condition;</pre>
--

Recursion

Recursive methods play a special role in computational science. The matter is that several lines of code can execute a code that is very hard to describe iteratively. A recursive procedure is also a very fruitful programming technique: it is constructive and contains its own proof. A traditional example is a factorial: $n!$ that is defined as

$$n! := \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

In the programming language this recursive procedure is expressed as

```

int Factorial(int n) {
    if (n =0)
```

```

        return 1;
    else if (n > 0)
        return n * Factorial(n - 1);

```

Some people distinguish recursion and recurrence, considering the former the logical trick allowing one to construct a *recursive procedure* capable to use itself to compute its own value. A *recurrence* is understood as a well-defined mathematical function, which is written in terms of itself. In other words, it is a mathematical function defined recursively. For our purposes, though, the difference is rather inessential.

As another well-known example, the Fibonacci sequence is usually given. It is the sequence of numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...

The first two numbers of the sequence are both 1, while each succeeding number is the sum of the two numbers standing before it. Let us define a function $F(n)$ that returns the $(n+1)$ th Fibonacci number.

$$\begin{aligned}
 F(0) &= 1 \\
 F(1) &= 1 \\
 F(n) &= F(n - 1) + F(n - 2), \quad n > 2
 \end{aligned}$$

The function F is called a *recurrence*, since it is defined in terms of itself computed for other values. The respective code fragment looks like

```

public static int fib( int n )
{
    if( n <= 1 )
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}

```

Java packages

Experienced programmers practically never write codes from scratch – a lot of worthy work has already been done, and it would be utterly impractical to reinvent the wheel. That is why a lot of ready-to-use packages and libraries exist in all widely used languages, and Java is no exception. In fact, it is the comprehensive volume of packages and libraries that makes the language powerful. In Java, libraries and packages are synonyms (but in other languages this is not necessarily so). Java packages contain many hundreds or thousands of classes that have proved being useful to developers engaged in various projects. Java developers usually select classes from several packages and them in exactly the same way as they would have used their own classes. For example, these imported classes have fields, methods and constructors, instances are created with the help of new keyword and so on. One can perceive a package in Java is a bundled group of related classes. To import a package, one must put the import statement at the beginning of the source code file before class declarations, e.g.

```

import java.io.*;           //contains classes for input and
                           //output operations

import java.util.* ;       /* contains various utility classes,
                           including classes for managing data
                           within collections */

```

```
import java.util.Random;    //imports specific classes from

import java.util.ArrayList; //the utility package

import java.lang.*;        /* imports all classes from java.lang
                             package. These classes support the
                             basic language features including
                             the handling of arrays and strings */
```

Actually, you don't need to import the `java.lang` package – it is the only one that is imported automatically into any Java program. So classes included in this package are always available by default, which is an essential convenience, in particular for numerical techniques. For example, you can directly use the `Math` class, which provides a Java developer with standard arithmetic and trigonometric functions (about the same set of functions that one would expect on a good scientific calculator). Due to by default availability of the `java.lang` package, the `Math` class functions and methods can be used at any time. By the way, the class `String` also belongs to `java.lang`.

The import statement can also be used to import a specific class from a package into a program, for example:

```
import Geometry.Shapes3D. Sphere; // include the class Sphere
```

One must notice in this connection that the asterisk `*` can only be used to select all the classes in a package but not to select all the packages in the directory (so you cannot write `Geometry.*` in the above example).

Classes within a package may be allowed to have an unlimited access to each other. Besides, there are no conflicts between class names from different packages – each package creates a new name space.

How to create Java packages

There exists a special command in Java: `package`. For example:

```
package mypackage.plot;
```

It is considered to be a good programming practice when a developer:

- stores Java class files containing packages in dedicated subdirectories (in fact, this is a convention), e.g. `mypackage.plot` should be stored as `mypackage/plot`;
- limits a number of classes in a package. Although there are no rules for how many classes should be included in a package, the practical recommendation is to have between 2 and 12 classes per package. Each class may in turn contain between 2 to 12 public methods;
- follows the recommendation to group utility classes (i.e. those that use another classes) into a distinct package. Since the concept of reuse is one of the most important in object oriented programming, it is recommended to design such utility packages for general applicability.

How to interpret Java programs

Even small Java codes have many features (this is because Java is a universal language, with standard ingredients). An example:

```
import java.awt.*; //(1)
import java.applet.Applet; //(2)
public class Hello extends Applet { //(3)
    public void paint (Graphics g) { //(4)
        g.drawString("Hi friends!", 55, 15); //(5)
    } //(6)
} //(7)
```

The most important part of this code is line (5) – it instructs the computer to display some text (e.g. on the screen). Since this piece of text is of the String type, it should be enclosed in double quotes. The other elements important for interpreting the above piece of code are:

- The parameters (two numbers) specify the position of the String on the screen: 55 pixels from the left and 15 from the top
- Lines (1) and (2) specify the library packages used in the code. This program utilizes AWT (Abstract Window Toolkit) and the applet library (to produce an output)
- Line (3) is a heading (class name) announcing that this code is an applet. The applet (the class itself) is enclosed in curly brackets
- Line (4) is a heading specifying the method. The latter consists of statements grouped inside another pair of curly brackets. There is only one such statement in this program (line (5))
- Statements should be terminated by a semicolon (a Java norm)
- A Java applet is launched in a browser or in an applet viewer
`java.applet.Applet;`

Java classes `java.applet` in general support Java programs run from a Web browser.

`Applet` may be interpreted analogously to audio clip (`AudioClip`): it can be played repeatedly, set to beginning, stopped at any time, etc.

- `java.awt`

`awt` (Abstract Window Toolkit) is a set of tools to provide windows, buttons, text fields, scrollbars, and pull-down menus. Some graphics methods, e.g. `drawString`, are also incorporated in this package.

Numerical methods in Java

Java was not designed for scientific computing; therefore it naturally has some deficiencies with regard to this field. These deficiencies mainly stem from persistent attempts to simplify the language, especially as compared to C++, and to reduce the risk of introducing bugs, while preserving more or less the same syntax. The features that make Java attractive to numerical computing are basically the same that are encouraging developers to utilize Java for other types of applications. Let us briefly overview advantages and disadvantages of Java for numerical techniques.

Advantages (we have already touched upon some of them):

1. Platform independence. We have already seen that Java bytecode should run on various platforms. This universal code behavior and architecture neutrality make porting Java numerical codes much less laborious than porting scientific programs written in other languages. However, the property of Java platform independence is to a certain extent theoretical, since in practice Java is implemented on different platforms in a different way.
2. Robustness. Java is comparatively simple and thus reduces the risk bugs. Besides, Java has the run-time exception handling built into the language.
3. Java is dynamic. Since the language is an interpreted one (at the run-time level), classes and libraries do not need to be recompiled when changes are introduced. For example, one can add new functions, methods and variables without making new recompilations over and over again. This is an essential convenience for scientific computations.
4. Distributed computing environment. Java objects can be dynamically loaded, transferred and controlled over the network, allowing the developers to work jointly over the Internet or private networks. The networking functionality of Java is further supplemented by a comprehensive library of classes adapted to modern network protocols (e.g. TCP/IP, HTTP, FTP).
5. Threads and multitasking. Threads allow a single program to execute many tasks simultaneously (see below). This feature allows one to take advantage of powerful multiprocessor systems.

Usually cited Java drawbacks for numerical computing:

1. A relatively low Java performance. In general, an interpreted language tends to be slower than the one running a compiled – native – code. Moreover, Java's attempted robustness does not go without a handicap. Such features as automatic garbage collection, automatic array bounds check, and exception handling can be considered as speed degrading factors.
2. Some drawbacks are associated with treating arrays, which is indispensable for scientific computing. For example, how a sub-array can be specified in Java? By copying it from the original array, which produces data redundancy. Furthermore, multidimensional arrays in Java are not necessarily contiguous, in distinction to other languages used for numerical computations, such as C, C++ and FORTRAN, where it is a specification requirement to put multidimensional arrays into contiguous memory areas. In Java, when working with multidimensional arrays there is always a risk of having to jump between totally different and non-adjacent areas of memory.
3. Java does not allow to define a complex data type and to use it throughout a program, as though it were a pre-defined type, e.g. for mathematical expressions.
4. Very limited number of numerical templates. Templates have recently become a standard feature of most scientific computing implementations. Their value lies in drastically saving the development time by sparing the write various pieces of code (e.g. functions) from scratch.

These and some other deficiencies are currently being addressed by the Java Grande Forum (<http://www.npac.syr.edu/projects/javaforce/javagrande/>). The group of experts involved this Forum's activities is working on introducing changes to the Java language in order to make it the fully adequate environment for numerical modeling and especially for high performance computing (HPC).

Java numerical libraries

A number of numerical libraries recently developed for using in the Java programming environment can be divided into two parts: 1) original Java developments; 2) conversions from other languages. The first category includes, for example, JAMA – the Java Matrix Package – developed jointly by NIST (the National Institute for Standards of the USA) and by the MathWorks company (famous for its MATLAB product). The JAMA package provides all the basic linear algebra and matrix operations that include:

- ❖ Addition, multiplication and other elementary operations
- ❖ SLE (systems of linear equation) solvers
- ❖ Eigenvalues
- ❖ Decompositions (LU, Cholesky, SVD, etc.)
- ❖ Derived quantities (determinant, rank, condition number, inverse)

JAMA is a developing system and its further functionality is envisaged, such as special matrix types: band, sparse, complex, etc. More detailed description of the JAMA library can be found under <http://math.nist.gov/javanumerics/jama/>

Numerical precision of a computer

Before we proceed with Java codes, let us recall some elementary concepts needed for numerical computation. Most numerical algorithms perform their actions until the estimated precision of the result becomes smaller than some given value – the desired accuracy or tolerance. We shall see it later on examples, now we shall briefly outline the following parameters:

`machinePrecision` – the largest positive number that, when added to 1, gives 1.

The `negativeMachinePrecision` can also be defined, namely it is the largest positive number that, when subtracted from 1, yields 1.

`smallestNumber` – the smallest positive number different from 0

`largestNumber` – the largest positive number that can be represented in the computer

`radix` – r , the radix of the floating point representation. The machine precision may be computed by finding the largest $n > 0$ as

$$(1 + r^{-n}) - 1 \neq 0;$$

This can be done in a loop over n . Then the quantity $\varepsilon_+ = r^{-(n+1)}$ is the machine precision. In the same fashion, the negative machine precision is computed by looking for the largest integer n , so that

$$(1 - r^{-n}) - 1 \neq 0;$$

then $\varepsilon_- = r^{-(n+1)}$ is the negative machine precision.

Some variables can be defined for Java implementation of scientific computing environment.

The variable `defaultNumericalPrecision` can be declared as having the precision estimate for a numerical computation. For instance, two numbers, x and y , may be considered equal if the relative difference between them is smaller than the `defaultNumericalPrecision`.

The variable `smallNumber` contains a value that may be added to a number without changing the result of a computation (up to machine precision). For example, an expression of the type $\frac{0}{0}$ is undefined insofar as the limit is not found. For numerical algorithms, if such an undefined expression is produced, adding a small number both to the numerator and denominator can skip the exception “division by zero” and would help to obtain the correct result.

How to compare floating-point numbers?

When using a finite number representation, results of two different computations that would have produced the same value from the mathematical point of view may be different. In principle, it is not quite correct to directly check the equality of two floating-point numbers; one should rather compare them while maintaining a given relative precision. For example, let us compare two numbers x and y . To do this, we can construct the following expression:

$$\varepsilon = \frac{|x - y|}{\max(|x|, |y|)}.$$

The two numbers may be considered equal with a precision ε_{\max} , if ε is smaller than a given number ε_{\max} .

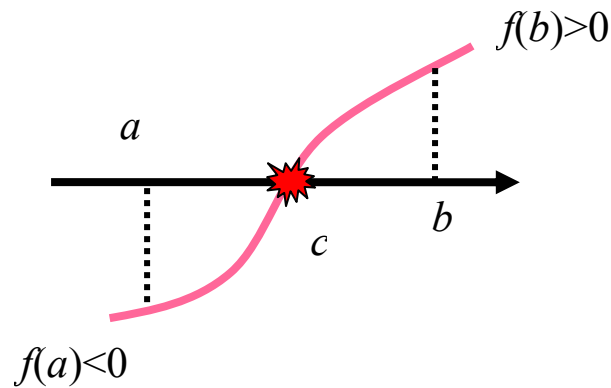
Below, we shall discuss some examples of using Java in typical numerical applications.

The bisection method

The bisection method in a narrow sense is a method for finding roots of an equation $f(x) = 0$ provided that $f(x)$ is continuous on the interval $[a, b]$ and that a root exists in this interval. The method is considered slow but sure: first the midpoint $c = a + (b-a)/2$ of the interval is calculated; then the function is evaluated at this midpoint. Three cases are possible:

1. $f(c)$ is zero, so c is the root
2. $f(b)$ and $f(c)$ have opposite signs. In this case the root lies in the interval $[b, c]$.
3. $f(b)$ and $f(c)$ have the same signs. In this case the root lies in the interval $[a, c]$.

This bisection procedure is repeated until the interval containing the root becomes small enough to achieve a reasonable accuracy ε . One can see that the number of function evaluations needed to achieve such accuracy is of the order of $-\log_2 \varepsilon$ (see below). For example, for $\varepsilon = 10^{-6}$ one needs about 20 evaluations. Thus we get a rule of thumb: for base 10 arithmetic you need about 3.3 bisections for each decimal digit of accuracy.



To determine zeroes of a continuous function, a sequence of segments is produced with the properties:

- 1) $f(a_n) \leq 0, f(b_n) \geq 0$;
- 2) $[a_n, b_n] \subset [a_{n-1}, b_{n-1}], n \geq 1$;
- 3) $b_n - a_n = 2^{-n}(b - a)$

The algorithm works as follows.

- Start $[a_0, b_0] := [a, b]$. Then calculate the middle-point: $c = 1/2(a_n + b_n)$
- Here we have two variants: either $f(c) \geq 0$ or $f(c) < 0$. In the first case we replace the right boundary point by c , in the second case the left point by c . The sequence a_n grows monotonously and is bound by b from above. Likewise, the b_n sequence monotonously falls and is bound by a from below. This means that there are limits for both sequences, c_a and c_b . Due to property 3), $(b_n - a_n) \rightarrow 0$, so $c_a = c_b := g$, and this value is the searched zero, since

$$0 \leq \lim_{n \rightarrow \infty} f(b_n) = f(g) = \lim_{n \rightarrow \infty} f(a_n) \leq 0 \rightarrow f(g) = 0$$

Where is the bisection method important?

In other words, why finding zeroes is important? Given the function $y=f(x)$, x_0 is a zero when $f(x_0) = 0$.

This can be applied to the general problem of computing the values of an inverse function: to find a value x such that $f(x) - c = 0$, where c is a given number. If we denote the inverse

function as $f^{-1}(y)$, then the problem may be formulated as: to find the value of $f^{-1}(c)$ for any c .

One can apply the bisection method to searching for the extremes of a function. If the derivative of the function can be computed, this problem may be transformed into finding zeroes of the derivatives.

Finding equilibrium points of a dynamical system or fixed points of a differential equation is the problem of vital importance in mathematical modeling. This problem is largely reduced to finding zeroes of certain nonlinear functions. Thus, the bisection algorithm can be successfully used here. The same procedure may be applied to difference equations.

If the measurements are assumed to be distributed according to the normal distribution, then one needs to find the zero of the function $f(x) = \text{erf}(x) - a$ (usually $a = 0.9$, then $x = 1.28$, $|x - \mu| \leq 1.28\sigma$).

Bisection algorithm

Here we can give a more formal treatment of the bisection algorithm

- **The main assumption:** we know two values of x for which the continuous function $f(x)$ takes values of the opposite sign, $x_+ : f(x_+) > 0$ and $x_- : f(x_-) < 0$. In this case, at least one zero of $f(x)$ exists in $[x_+, x_-]$.
- If the function is not continuous, the zero cannot be guaranteed, examples: $f(x) = 1/x$; $f(x) = \text{sign}(x) = |x|/x$, so the continuity condition is essential for the bisection method to be applicable.
- x_+ and x_- are the initial values for the method. The algorithm:
 1. Compute $x = (x_+ + x_-)/2$
 2. If $f(x) > 0$, set $x_+ = x$ and go to step 4
 3. Else, set $x_- = x$
 4. If $|x_+ - x_-| > \delta$ (the desired precision), go back to step 1

Initialization: $x_1 := a; x_2 := b; f_1 := f(x_1); f_2 := f(x_2); \delta := 10^{-5}$

Iteration: a) computing the middle-point $x_3 := \frac{1}{2}(x_1 + x_2)$

b) computing the value of the function $f_3 := f(x_3)$

Setting the new interval: c) if $f_2 f_3 \leq 0$ //zero lies between x_2 and x_3

then $x_1 := x_3; f_1 := f_3$

d) if $f_2 f_3 > 0$ //zero lies between x_1 and x_3

then $x_2 := x_3; f_2 := f_3$

End condition:

e) if $|x_2 - x_1| \leq \delta$, then $g := x_3$. Stop

f) if $|x_2 - x_1| > \delta$, then continue with a)

Remark: the function f can have many zeros in the interval $[a,b]$. The bisection algorithm delivers only one.

How efficient is the bisection method?

Now we are in a position to discuss the efficiency of the bisection method more in detail. For a given pair of input values, x_+ and x_- , the number of iterations required to obtain the accuracy δ may be expressed as

$$n = \left\lceil \log_2 \frac{|x_+ - x_-|}{\delta} \right\rceil$$

For instance, if the distance between two input values is ~ 1 , then the number of iterations needed to obtain the accuracy (tolerance) of $\delta = 10^{-6}$ would be 20, for $\delta = 10^{-8}$ $n = 30$ and so on. This example indicates that the bisection algorithm is rather slow.

The accuracy of each evaluation is $|x_+ - x_-|$, since the zero of a function always lies within this interval.

It is essential to know the initial values, x_+ and x_- , in order to start the bisection algorithm, which means that some methods to declare the input values should be provided

The bisection algorithm is the implementation of an iterative process. Typical examples of iterative procedure of finding zeroes ($f(x) = 0$) are represented by the following functions and initial intervals $[a,b]$:

- $x - \exp(-x)$, with initial interval $[0,1]$;
- $x - \cos(x)$, with initial interval $[0,1]$;
- $x^2 + 4x - 10$, with initial interval $[1,2]$

Now we shall present the Java implementation of the bisection method using the function $f(x) = x^2 - 6x - 12$ as an example.

```
public class Bisection {
    double x,f,x1,x2,x12,temp; // Declaration of variables
    int k;

    double func(double x) // Function declaration
    {
        f=x*x-6*x-12;
        return f;
    }

    void Bisect(double p1, double p2 )
    // Declaration of method Bisect() to calculate the roots
    // of the equation
    {
        x1 = p1;
        x2 = p2;
        if(func(x1)==0) { System.out.println(x1); k=1; }
        if(func(x2)==0) { System.out.println(x2); k=1; }
```

```

while(k==0) {
    x12 = (x1+x2)/2;

    if(func(x12)*func(x1)<0)
        x2=x12;
    else
        x1=x12;

    temp=(x2-x1);

    if( temp < 0.0001) {
        System.out.println("the solution is:" + x2);
        k=1;
    }
}

public static void main(String[] args) //Main method declaration
{
    double k1,q;
    int a,p1,p2;

    Bisection object = new Bisection();
    object.Bisect(-10,3);

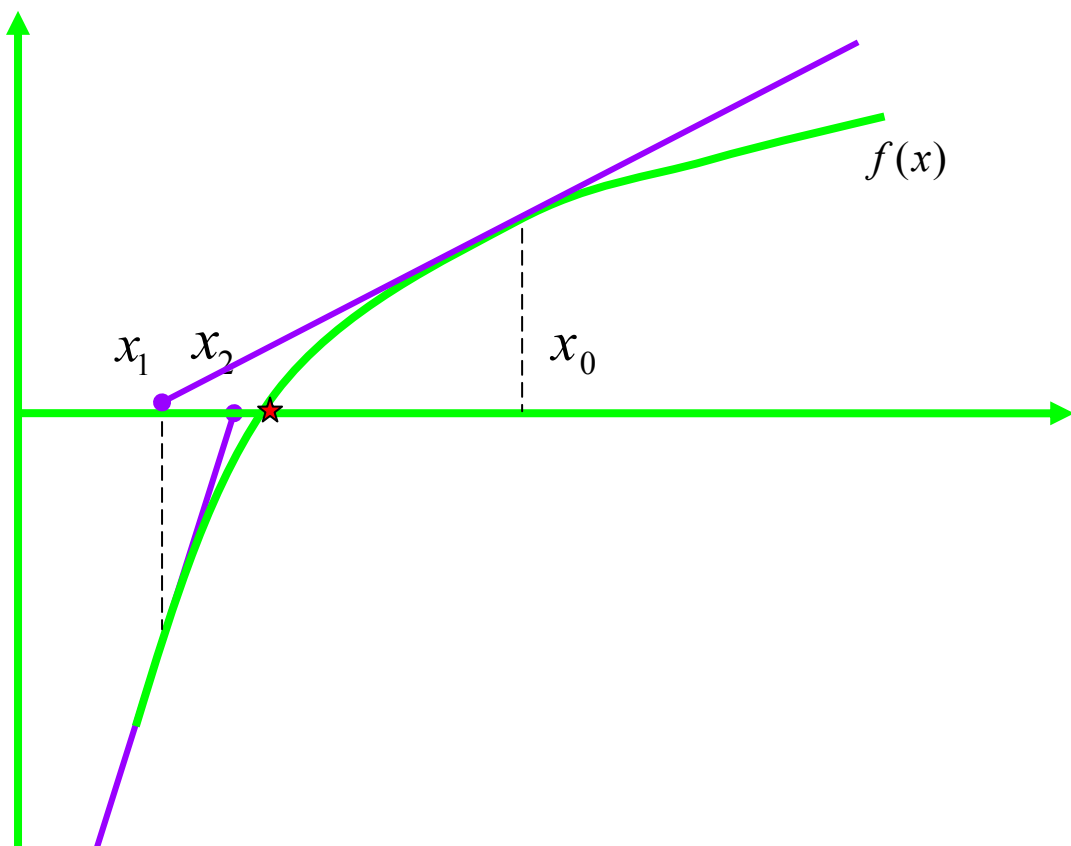
} // end of main method

```

The Newton's method

Apart from the bisection method, there are some other popular algorithms of finding zeroes of a function. Probably the most well-known one is the Newton's method – it is an algorithm working with successive approximations. This method can be better understood using its geometric interpretation:

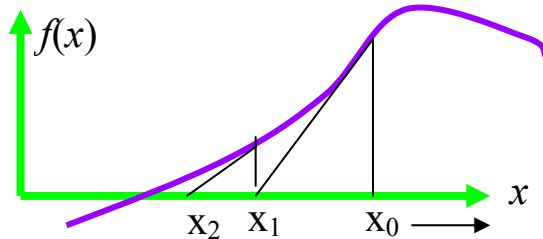
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



x_{n+1} is the point where the tangent to the curve at the point x_n crosses x -axis. The starting point can be any one where the derivative is nonzero. The function is replaced by a sequence of linear approximations – straight lines defined by preceding pairs $(x_n, f(x_n))$. From the mathematical viewpoint, the function $f(x)$ is replaced by its linear approximation (first two terms of the Taylor series), $y = (x - x_n)f'(x_n) + f(x_n)$

Limitations of the Newton's method

- The Newton's method depends on a guess or on a good initial estimate of the root, x_0 , as well as on knowing the derivative in this point, $f'(x_0)$. The method may not converge if df/dx is close to zero
- Each subsequent evaluation of the root is the next better estimate, with the stopping criterion: $|\text{next estimate} - \text{current estimate}| < \text{tolerance}$. If the initial estimate of the root is too far, the algorithm may fail to converge



If we move the initial value x_0 too far out, the convergence of the Newton's algorithm disappears (e.g. for $x_0 \rightarrow 2x_0$)

Now let us consider how this method can be implemented in the Java language.

```
public class myNewton {

    // method to compute values of a function
    public static double f (double x){
        double f=x*x-4*x;
        return f;
    }

    // method created for computing first derivative if the function
    public static double df (double x){
        double df = 2*x-4;
        return df;
    }
}
```

```

public static void main (String[] args) {

    double x=100.00;
    double tol=0.001;

    myNewton Newton_test = new myNewton ();

    // choosing a starting point
    if (Math.abs(df(x)) < tol){
    /* if the point is badly chosen (point of local extremum of
    the function-first derivative is close to zero),we need to
    choose another starting point */
        System.out.println("Point badly chosen, choose another
        starting point");
    }
    else {
        // otherwise, we are searching for the roots of the equation
        // using Newton's method

        while (Math.abs(f(x)) > tol){
            double x1= x-(f(x)/df(x));
            x=x1;
        }
        System.out.println("First root of the equation is " +
            Math.round(x));
    }

    // proceed with searching, assuming the existence of one more root
    System.out.println("Proceed with searching, still missing one root");
    x=-1*x;
    while (Math.abs(f(x)) > tol){
        double x1= x-(f(x)/df(x));
        x=x1;
    }
    System.out.println("Second root of the equation is " +
        Math.round(x));
}}

```

The secant method

The Newton's method is essentially dependent on the knowledge of the derivative to the function. However, in some cases the derivative df/dx may be unknown, so it would be impossible to apply the Newton's method. In such cases the so called **secant method** may be employed.

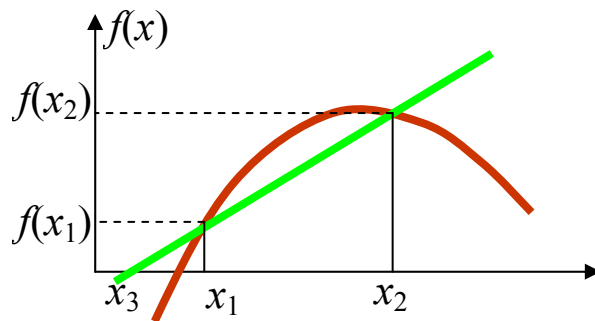
Assume that x_1, x_2 are initial estimates of the root, then the next – better – estimate is given by

$$x_3 = \frac{\begin{vmatrix} x_1 & x_2 \\ f(x_1) & f(x_2) \end{vmatrix}}{f(x_2) - f(x_1)}$$

The meaning of this relation is a geometrical similitude between triangles.

The algorithm uses: 2 initial estimates, tolerance, iterations.

The algorithm returns: 2 final estimates and iterations taken.



The secant method is advantageous when there is an implicit equation and it is difficult to compute the derivative first.

- Example:

$$aw \left[1 - \exp\left(-\frac{b}{c + pw}\right) \right] - Q = 0$$

This is an energy balance equation describing a typical heat exchange problem encountered in civil engineering practice and relevant to power engineering and power plant design problems. The meaning of this equation is: heat is transferred from steam condensing in a heat exchanger to water, w being the mass flow rate of the water (kg/s). The quantity Q (measured in kW) is the desired rate of energy transfer. Phenomenological (taken from the experiment) coefficients a, b, c, p serve as control parameters.

The Java code for the secant method may be written in the following form.

In order to implement the secant method in the Java language, let us define at first the abstract class Secant:

```
public abstract class Secant {

    // The declaration of an arbitrary function to be specified
    // in a derived class
    public abstract double f (double x);

    public int iteration;          // stores the maximal number of iterations
    public double estimate [];    // stores the calculated values

    // The main method implementing the secant algorithm
    public void solve (double x1, double x2, double tolerance, int imax)
    {

        // allocates the memory space for the values to be calculated
        estimate = new double [imax + 1] ;

        double f1, f2 ; // temporary variables, which store the function
                        // values at the presumed zero points
        double x ;      // temporarily stores the approximation to zero
    }
}
```

```

// the initialization of variables
f1 = f(x1) ;
estimate [0] = x1;
estimate [1] = x2;
x = x2;
iteration = 1;

// do-loop iteration
do {
    x = (x1*f2 - x2*f1) / (f2 -f1);
    iteration ++ ;
    estimate[iteration] = x;
    x2 = x1;
    x1 = x;
    f2 = f1;
    f1 = f(x1);
    // check whether the tolerance limit and/or the
    // maximal number of iterations is reached
    } while ((Math.abs(x2-x1) > tolerance) & (iteration < imax));
}
}

```

The secant method implementation corresponding to a particular function can be now obtained by the following derived class. We simply specify the method “f” as shown below.

```

public class ExampleSecant extends Secant {

    // method to compute values of a particular function
    public double f (double x){
        double f=x*x-4*x;
        return f;
    }

    // the main function
    public static void main (String[] args) {

        // creates the instance of the class ExampleSecant
        ExampleSecant secant_test = new ExampleSecant ();

        // starts the calculations by the invocation of the method
        // solve
        secant_test.solve((double)-1, (double)2, (double)0.001,
            (int)20);

    }
}

```

Heat and mass transfer with Java

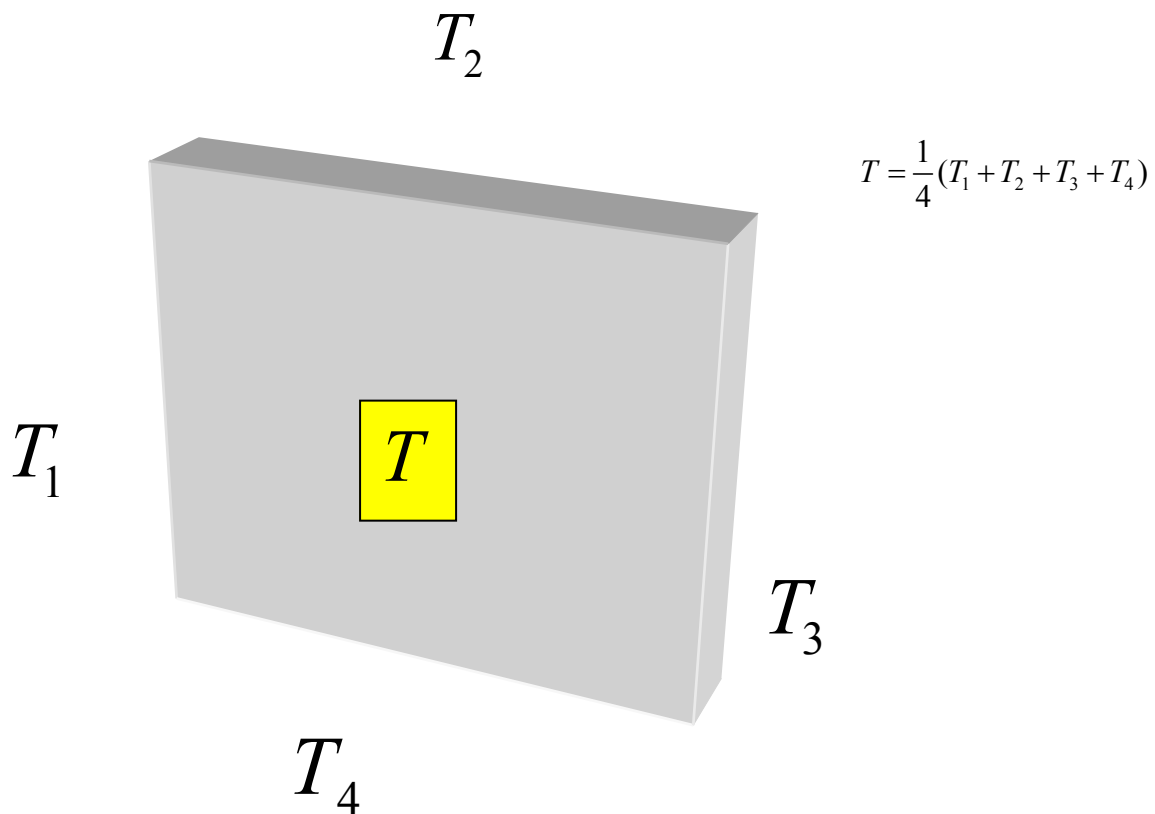
The above example related to finding zeroes of implicit function concerns the heat and mass transfer problem. A slightly more complicated example manifests the numerical solution of heat transfer equation

$$\frac{\partial T}{\partial t} = a^2 \Delta T$$

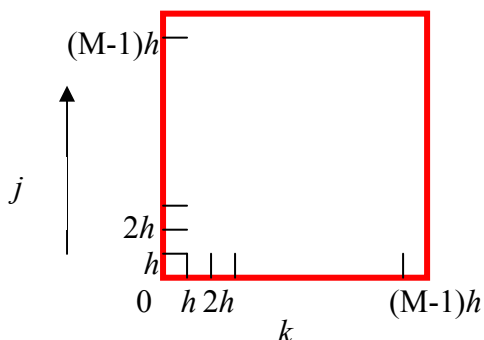
in the situation close to thermal equilibrium. If we discretize the stationary two-dimensional heat transfer equation, we arrive at the following algorithm for computing the temperature:

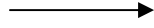
$$T_{i,j} = \frac{1}{4}(T_{i-1,j+1} + T_{i+1,j+1} + T_{i-1,j-1} + T_{i+1,j-1})$$

that is the temperature at the grid point (i,j) may be represented as the average of temperature values in the adjacent grid points. Using this algorithm, we may construct a Java simulator of the temperature distribution e.g. in a two-dimensional slab. Of course, the heat transfer equation should be supplemented by appropriate boundary conditions.



The averaging of the temperature is, in fact, the consequence of the conduction rate equation (the Fourier's law), $\mathbf{q} = -k\nabla T$, which may be interpreted as the linear interpolation between points with different temperatures. Formally, one can obtain the above relation by considering the asymptotic (steady-state) heat transfer equation in a 2D area:



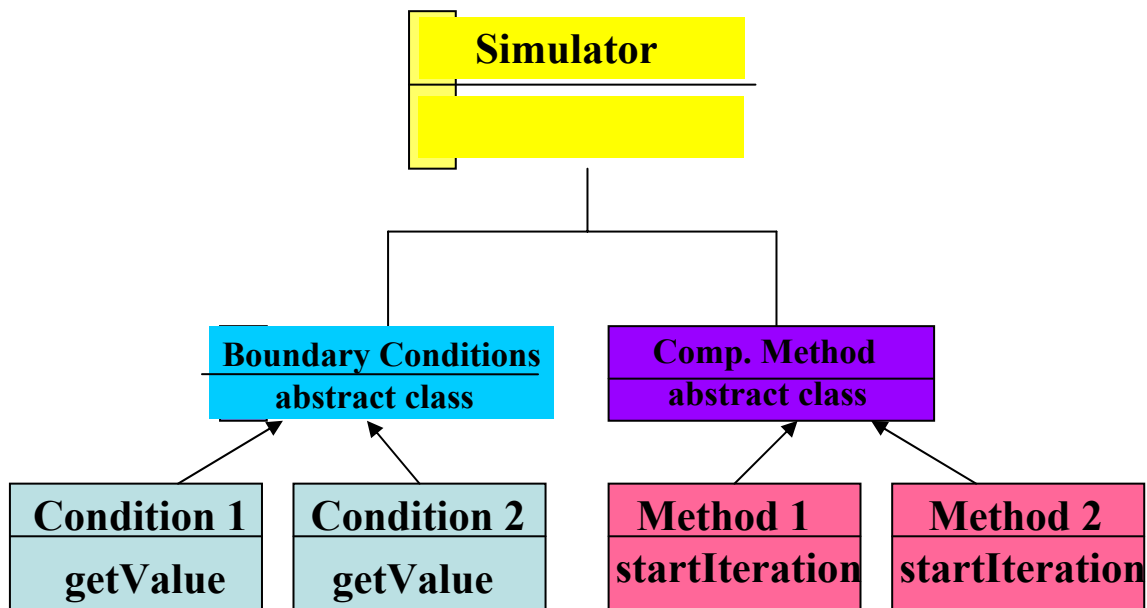


$$\Delta_2 T = 0$$
$$T_{k-1,j} + T_{k+1,j} - 4T_{k,j} + T_{k,j+1} + T_{k,j-1} = 0 \quad (*)$$

Let us now draw a class diagram for a temperature field simulation.

A project pattern: temperature

The class diagram



The above diagram introduces the main class “Simulator”, which provides the principle environment for two other necessary classes : “CompMethod” and “BoundaryCond”. The latter two correspond to the mathematical relationship describing the heat transfer problem (*).

Let us begin with the implementation of the simulation environment.

```
public class Simulator
{
    Matrix T;           // The Matrix used for computation

    CompMethod method; // The objects describing
    BoundaryCond boundary; // a particular computational method and
                        // boundary conditions.

    // The constructor
    public Simulator(CompMethod method, BoundaryCond boundary, int msize)
    {
        this.method = method;
        this.boundary = boundary;
    }
}
```

```

    T = new Matrix(msize);
}

public void startSimulation()
{
    // set boundary condition along each side

    int i;

    for(i=0;i<=T.getSize()-1;i++)
        T.set(0,i,boundary.getValue(i));

    for(int j=T.getSize()-1;j>=0;j--)
        T.set(j,T.getSize()-1,boundary.getValue(i++));

    for(int j=T.getSize()-1;j>=0;j--)
        T.set(T.getSize()-1,j, boundary.getValue(i++));

    for(int j=T.getSize()-1;j>=0;j--)
        T.set(j,0, boundary.getValue(i++));

    // passing the matrix with set up boundaries to the method
    method.setT0(T);
    // execute the iterative method to compute the temperature field
    method.startIteration();

}

// The computational result output
public void printResult()
{
    T.printMatrix();
}

// The main method
public static void main(String[] args)
{
    Simulator simcenter = new Simulator(new Method1(),
                                      new Boundary1(),5);

    simcenter.startSimulation();
    System.out.println("\nResult:\n");
    simcenter.printResult();

}
}

```

Before implementing the particular computational method, let us first define the abstract vision of the iterative computational process:

```

public abstract class CompMethod
{
    Matrix T; // The matrix, on which the calculations are performed

    // The empty constructor
    public CompMethod()
    {
    }
}

```

```

// Using this method, the matrix involved in the computation
// with specified boundary conditions can be passed to the particular
// method
public void setT0(Matrix T0)
{
    T=T0;
}

// abstract method describing the appropriate iteration
// should be implemented in derived classes
public abstract void startIteration();
}

```

For example, the above computational prescription (*) can be implemented as a particular method:

```

public class Method1 extends CompMethod
{
    public Method1(){}

    public void startIteration()
    {
        for(int k=1;k<100;k++)
        {
            for(int i=1;i<=T.getSize()-2;i++)
                for(int j=1;j<=T.getSize()-2;j++)
                {
                    T.set(i,j,(T.get(i-1,j+1) + T.get(i+1,j+1) +
                                T.get(i-1,j-1) + T.get(i+1,j-1))/4);
                }
        }
    }
}

```

The class responsible for the boundary conditions can be implemented using similar techniques. We begin with the implementation of an abstract class, which describes unspecified boundary conditions. To implement a particular physical situation, we make use of the inheritance.

```

public abstract class BoundaryCond
{
    // The constructor
    public BoundaryCond()
    {
    }

    // The abstract method, which returns the boundary value
    // at a given position
    public abstract double getValue(int i);
}

public class Boundary1 extends BoundaryCond
{

```

```

public Boundary1()
{

}

public double getValue(int i){if(i>10) return 3; else return 1.5;}
}

```

Finally, the Matrix class provides typical matrix algebra operations needed for calculations:

```

public class Matrix {

private int Size;
private double Mvalue[][];

/* This constructor creates a square matrix of the size n. */
public Matrix(int n) {

    this.Mvalue = new double[n][n];
    this.size = n;
    for (int z = 0; z < this.size; z++)
        for (int s = 0; s < this.size; s++)
            this.Mvalue[z][s] = 0;
}

/** This constructor creates a square Matrix of the size n
    and initializes it with the values of the matrix M. */
public Matrix(Matrix M) {

    this.size = M.getsize();
    this.Mvalue = new double[size][size];
    for (int z = 0; z < this.size; z++)
        for (int s = 0; s < this.size; s++)
            this.Mvalue[z][s] = M.get(z,s);
}

/** returns the matrix size */
public int getsize() {

    return this.size;
}

/** returns the matrix element for the position (row,column) */
public double get(int z, int s) {

    return this.Mvalue[z][s];
}

/** endows the matrix element in the position (row,column) with
    the value val */
public void set(int z,int s, double val) {

    this.Mvalue[z][s] = val;
}

/** endows the whole matrix with the value of the matrix M */

```

```

public void set(Matrix M) {
    for (int z = 0; z < this.size; z++)
        for (int s = 0; s < this.size; s++)
            this.Mvalue[z][s] = M.get(z,s);
}

/** gives matrix elements */
public void printMatrix() {
    for (int z = 0; z < size; z++) {
        for (int s = 0; s < size; s++) {
            System.out.print(" "+this.Mvalue[z][s]);
        }
        System.out.println();
    }
}
}

```

Threads

Probably from your daily experience you know that very often it is needed to do many tasks at once. When working with a computer, it is sometimes translated in the necessity to create many windows, almost identical copies of an image, or different versions of the same computation – all of them being simultaneously available. If you use a computer, you always tend to ask it to do several things at once: edit a file, receive e-mail, print another file (background printing), download a file from the Internet, play music, etc.

In engineering physics, e.g. while modeling a nuclear reactor, one can define multiple control rods and operate them independently. In this process, each rod can be handled by a separate thread. Thus, threads allow one to expand a Java code from just a single program into a multitude of simultaneously running programs. Hence, threads can be a cost-effective way of managing the computer resources, primarily processor power and memory. This fact may be extremely important for complicated models and programs.

It is the programmer's task to ensure that threads could be created and annihilated at will, and that they should be transparent for the user.

Where are the threads typically applied? Very often by creating user interfaces. Imagine that an applet or a graphics application is designed to draw shapes or to display a text, and the user wants to stop this activity. A typical way to do it is to provide a button "cancel". In Java terms, this means to have a `cancel` method within the code. If such "cancel" method (e.g. a button) is provided, the user can press this button, so the program would stop computing or outputting and would wait for new events. However, if the button is handled by a separate thread, there are much more opportunities to react (see below).

We have mentioned above that typically the user tends to ask a computer to do a number of tasks at once. Nevertheless, in reality a single processor can do only one thing at a time, but because modern processors work rather quickly (in the human time-scale), it can share its time between concurrent activities without being noticed by the user. A program, in general, is a series of instructions, which the computer obeys in sequence. However, there are usually diversions from this sequential order, for example due to `for` and `while` loops, conditional and

selection (if) statements, but the main program still has to be executed. Multithreading might be regarded as the extension of the looping philosophy. In order to thread a Java program, one needs primarily to identify the methods that can run independently. These methods are within a class, so it is the class that becomes a thread – usually in multiple incarnations, i.e. instantiations.

In the simplest case, threads are just identical copies of the same class and run independently. In a more complicated case, threads are allowed to interact. This process is quite similar to physics, where the interaction between objects adds complexity and versatility to the picture under consideration. (One may note in passing that many concepts of object-oriented programming are rooted in natural sciences.)

Modeling examples:

1. In modeling a plant, in particular controlled by a computer, one must ensure that all separate activities such as conveyor belts, furnaces, valves, pumps, etc. would not interfere.
2. Suppose we would like to keep track of how many cars enter two parking lots near a building. Here one may consider two independent counters residing in two separate threads, which are identical in every respect except the name (e.g. East and West parking). So, one can define a single thread class and instantiate it twice.

We have already mentioned that in multithreading the processor is dividing its available time between different threads, producing the impression that they are all executed at the same time. How is this process reflected in the Java language?

In Java, multithreading is supported directly by the language. **Thread** is a class in the **java.lang** package, which means that it is always available and no special **import** method is required. This fact is regarded as a considerable advantage of Java, especially for high performance computing.

Any class the programmer wants to become a thread class must inherit from **Thread**, or, in other words, any class is declared as threaded by extending the **Thread** class:

```
public class Classname extends Thread {
/* Then each new thread is created like any other object declaration */
.....
}
Classname threadname = Classname (parameters);
/* If it is unnecessary to give the thread a name that is known to its ancestors, one can simply
create a thread of the given class */
```

The **Thread** class contains a number of methods, the three most important of them are: **start**, **run**, and **stop**

There exist also **sleep** and **yield** methods. Both **sleep** and **yield** cause the thread to unload the processor. The **sleep** method makes it to wait for a prescribed time, whereas **yield** forces a thread to give way to other processes.

After a thread has been created through the constructor, the method **start** is invoked, which in its turn makes **run** to be called. The **run** method consists of a loop that continues until certain

conditions are fulfilled, and then the loop ends. As a result, the thread dies out – we see here a similarity with a loop being terminated..

The main method is by default a thread, and from it other threads are beginning. This means that threads can hierarchically spawn their own threads

A modeling example with thread

For example, we can write a program named **Bounce** describing a ball bouncing a limited number of times, e.g. 100 repetitions controlled by the for loop. When the thread **Bounce** has finished looping, it exits the method **run** and ceases to exist. The **stop** method needed to terminate the program can be received through the **Thread** class. For instance, we can set up a new button “Stop” in the user interface:

```
button stop = new Button ("Stop");  
add(stop);
```

A professional way to terminate the thread is to establish a Boolean flag, e.g. named **keepGoing**, which is initially set on **true** when the object is created and made **false** when the user presses the stop button. The respective loop takes the form:

```
while (keep going) {  
    ..... // some_comment  
}
```

Java supports priorities for threads: between 1 (low) and 10 (high), so that the operating system knows which thread to run first.

Interacting threads

Why do we need threads at all? Actually, they are mini-processes running in parallel, so that the same code can be reproduced a number of times or the same task can be implemented more than once. However, often threads must pass information to each other, for instance, some results of mini-computations. Since threads are objects, communication between them utilizes ordinary object-oriented techniques – calling methods. In the spirit of object-oriented approach, this is performed by sending signals to change the state, e.g. to change to another mode by invoking stop method, or by updating non-private variables.

Shared resources and synchronization

It may occur that different threads begin competing for the same resources which can provoke conflicts or can lead to undefined states and unexpected (“random”) results. To avoid a possibility of such conflict, Java provides a so called synchronization method. This method is designated by the **synchronized** modifier. The meaning of the synchronization is as follows: Java ensures that all other threads that potentially can access the same resources (e.g. manifested by variables), except the one that has gained an access to a method to be synchronized, will be halted, until the selected thread finishes its job and exits from the method.

The necessity to resort to such mutual exclusion occurs quite often in numerical methods, when some data is accessed by two or more threads. For instance, when two threads are counting numbers and each is incrementing the value of a shared integer, without synchronization, conflicts are possible even in a simple program.

Syntax of synchronization:

```
Public synchronized void criticalMethod( )
```

Conclusion: achieving efficiency and reusability

A complex programming task might require a lot of resources, such as time, memory and storage space, but simple tasks should not. It follows from here that the best way to spare resources is to reduce the program complexity. Now we can provide just a few tips how to keep the program compact, without compromising its main goals:

- do not declare too many variables – only when needed;
- minimize the number of common statements and maximize the use of methods;
- in numerical methods and associated programming: make common calculations just once and store the results.

When writing a new program, do not try to do everything from scratch. Look around for existing classes and packages that can be used or, at least, adapted. This approach is called reusability, and Java is especially well-adapted to it. We have seen that Java has special class constructs to assist the developer with reusability, namely inheritance, abstractions, constructors, multiple object instantiations, references, interfaces, etc. Thus you can reuse many results obtained by other clever and competent people. And if, from the very start of Java programming, you will try to write your classes in such a manner that they have a general appeal, you also will be called clever and competent.