The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific
and Cultural Organization

International Atomic
Energy Agency

310/1779-2

Fourth Workshop on Distributed Laboratory Instrumentation Systems
(30 October - 24 November 2006)

Object Oriented Programming Principles

Olexiy TYKHOMYROV
Department of Experimental Physics
Dnepropetrovsk Experimental Physics
Proulok Naukovij, 13
49050 GSP 50 Dnepropetrovsk
UKRAINE

Object Oriented Programming Principles

Olexiy Ye. Tykhomyrov*

Departament of Experimental Physics
Dnipropetrovsk National University
Ukraine.

Supporting material for
lectures given at the:
The Fourth Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 30 October – 24 November 2006

LNS

*tiger@ff.dsu.dp.ua

Contents

1 Introduction

1.1 Acknowledgements

The material covered in this Chapter was written as a compilation of ideas and examples from a variety of authors, but Ulrich Raich was the very first enthusiast in our group who prepared the notes on Java for the Real-Time College in Dakar in 1998. Lots of ideas and examples were taken from those notes in the preparation of this Chapter. Catharinus Verkerk suggested lots of improvements and ideas about planning the lessons. Lot of material from the notes of Prof Richard Baldwin[6] and Paul Ramos[7] was used as well.

1.2 Historical Development

During Real-Time Colleges (1992-2002), **X**-Window System and **Motif** have been used to build Graphical User Interfaces – GUIs for short. GUI programming is always a tricky business and needs a steep learning before reasonable results can be obtained. Java allows a programmer the ability to do much more than just programming and building nice-looking GUIs. Java is popular, Java is free, Java may be run on almost all hardware platforms. So why not teach Java?

Why OOP (Object Oriented Programming)? Java is an OOP language. It means that to write a program in Java, you *must* posses at least some basic knowledge of OOP. You may write a program in C++, which is also an OOP language, without understanding OOP paradigm; but this is almost impossible with Java. Java is a pure OOP language.

Object **O**riented **P**rogramming, or **OOP** for short, (we will use this abbreviation also for **O**bject **O**riented **P**rogram(s)) is the latest fashion in computer science. OOP is so cool, that some people want to re-write all programs and operating systems using an OOP language, say C++ for example. People of thought are talking about OOP and Java but. . . let's better to explain this item.

When the first computers came into the market, computer hardware was terribly expensive; a Digital minicomputer in the early seventies cost several hundreds of thousands of dollars but was extremely limited in performance and resources when compared to a 1000 dollar PC you can buy nowadays in a supermarket.

Since a computer costs much more than the cost of labour, programming has been done in a way to maximise use of the precious hardware resources. This means the programs were *hardware oriented*. Those days it was extremely important to fit the code into the small amount of available core memory, of which each bit was hand wired. Disk space was around 5Mbyte and part of this space had to be used to keep also an operating system. A program should have used the central processor of the computer very carefully, so each piece of code was written to optimise CPU time as well. Naturally many programs were written in assembler, the most hardware oriented language. For scientific computation FORTRAN (**For**mula **Tran**slation), the language specifically designed for such kind of applications, was used.

As computers became more elaborate and more powerful, and hardware resources got cheaper, the amount of software to be provided grew catastrophically and soon exploded. The demand for new software could not be fulfilled by employing the methods used at the time and the first software crisis appeared. People learned that writing large programs without thinking of modularisation and structuring was simply not feasible. "Spaghetti programming", the path a program would take which resembled a pot of spaghetti, was the result.

One of the super-programmers Donald Knuth estimated the mean quantity of "GO TO" statements in FORTRAN programs to be around 13%. So the enemy of all programs was identified and banned. A new language was invented similar to ALGOL[1] in order to implement *structured programming* concepts. This new language called PASCAL is still used as an excellent language for teaching, and to a lesser extent as a general purpose programming language. Following the modern trend in object orientation PASCAL was extended to become a new language called OBJECT PASCAL. Using OBJECT PASCAL as the underlying language a powerful development environment known as DELPHI, used primarily to build client/server applications for Microsoft Windows platform, was produced by Borland.

The main distinguishing feature (grown from ALGOL) is that, each program was subdivided into blocks with one entry and one exit. This was true not only for *functions* and *procedures* but also for *if-then-else* clauses, *for*, *repeat-until* and *do-while* loops. Functional entities needed to be identified and the problems to solve were subdivided into functions and procedures, where each subroutine implemented one of these functional units. Each subroutine had a well defined interface, i.e. (parameters of a given type to be passed) and it was even possible to have several programmers working on the same project, each one implementing a few procedures, and then sticking them together to form a working program, which most of the time worked. To run a program (written in FORTRAN or PASCAL) a *translator* was used to produce machine-dependent code of a target computer. The resulting binary code would only run on the single type of machine and unfortunately was incompatible between computers. To run this program on any other target machine at lease some adaptation and modification of the source code was needed to be carried out.

Computers became more and more powerful, and memory and disk space became cheaper, but software was still incredibly complex, huge and non-portable. It was the new software crisis. In addition the **Internet** changed the view of computing within a short period and most of todays computers are included in some network or another. Writing programs for a single type of CPU became too costly (now the cost lies in the software and not on the hardware side any more) and we needed a language whose "binary code" could run on any machine. Even better, if it can be downloaded over the Internet and executed on any machine without the user being aware of the download process. Just as PASCAL was the miracle language that could solve all problems and solve them forever, Java and OOP are supposed to do the same for all future problems; so you recognise how

[1] ALGOL stands for **Algo**rithmic **L**anguage

important OOP is.

In this Chapter we concentrate on Object-Oriented Programming (OOP). Notes are intended to be independent of any specific language. As a practical matter, it is necessary to use some language for illustration purposes, and rather than to conjure up some artificial language, examples in this Chapter are provided using the Java language.

Some languages such as C do not readily support OOP. Other languages such as C++ support OOP, but don't mandate that you use the object-oriented features of the language.

Java requires you to program by means of OOP techniques. In particular, it is not possible to write even the simplest program in Java without taking an object-oriented approach. You may develop an excellent program, or you may develop a poor program, but your Java program **must** use objects.

Although Java has some very complex and sophisticated features such as multi-threading, it is not difficult to learn how to program in Java. To program in Java, you have to:

1. Learn how to productively utilise the large set of class libraries containing dozens of classes and hundreds of methods which are provided as part of the *Java Development Kit*[1] or *Java Standard (Enterprise) Edition* to supplement the language;
2. Learn how to design and program in the object-oriented paradigm.

The first of these challenges can be met on a gradual basis. In other words, it is not necessary to know the entire class library to produce useful Java programs. Learning new tools and Java capabilities certainly help you to produce more powerful programs.

However the second challenge cannot be met on a gradual basis. It is not possible to create even the simplest Java program without programming in *the object-oriented paradigm*.

1.3 Functional and OOP Paradigms

To solve a problem we usually think about it *in general*, or at an *abstract* level. If we have to calculate resources (eg. money), we *decompose* the problem. In case of limitations, the human may use some tricks. For example, if we do not have enough wallpaper, we may not cover some hidden places behind furniture, pictures, etc. It means we try to *organise* the solution by knowing the available resources and according to a plan.

These three elements of solving any problem, ie. **abstraction**, **decomposition** and **organisation** together are known as a **paradigm**. Previous programmers' experience was based on the *functional* paradigm.

Imagine that we are going to reach a place by car so we start to think how to do that. According to functional programming paradigm we should have thought of

[1] Java-1 version

the problem in terms of *functions* acting on *data* . However in the OOP paradigm, we think of the problem at three levels as follows:

On abstraction level - think of the problem in terms of a process that solves it, i.e. how to drive a car, how to collect necessary things, etc.

On decomposition level - think of how to break all this processing down into small manageable processing units, or functions; e.g. how to move a car from one point to another. We recognise driving in the town differs from driving on the highway. We think how to do stopping/starting the car etc. So we create functions.

On organisation level- set up our functions so they call each other according to the rules. We set up sequences of function calls, passing different arguments to the functions, etc.

This is the clue: previous approaches to programming tended to separate data from the methods used to manipulate that data, or at least don't strongly encourage them to be considered in concert. This is not typical way of thinking; we have to define a set of data structures and then define a set of functions acting upon these data structures.

In a real situation we think first about the way in general, and only having chosen the way we start to think how to get through the traffic in each of the cities and villages along the way. Then we think about details, for instance where is the best place to stop for eating, sleeping etc, and after that we may think how to schedule the journey, to be in a particular restaurant at a particular time. This is an example of *object-oriented thinking*: we think in terms of objects interacting.

On abstraction level we think in terms of independent agents (objects) working together such as the car, ourselves, things to take etc.;

On decomposition level we define the kinds of objects on which to split the global task;

On organisation level we create the appropriate number of objects of each kind.

2 An Object-Oriented World

In OOP paradigm, from the programmers point of view, an object-oriented language *must* support three very important *explicit* characteristics. We use these concepts extensively to model the real-world problems when we are trying to solve with our object-oriented programs. These three concepts are:

- polymorphism,
- inheritance,
- encapsulation.

The *implicit* characteristic is **abstraction**. We use it to *specify* new *abstract data types*, or ADT for short.

2.1 Polymorphism Example

The Greek word *polymorphism* means *one name, many forms* and perhaps is the most difficult term to explain, especially in real-world terminology. Let us look at a car again and say that polymorphism is somewhat akin to the automatic transmission in your car. Suppose, it has three functions: *Neutral*, *Drive*, *Reverse*.

We know for sure, that there is only one *method* that is referred to as *Reverse*. Whenever we select *Neutral*, the engine of a car is disengaged from the transmission mechanism.

While we are driving a car to move forward, we select *Drive*. Depending on various conditions at *runtime*, the automatic transmission system mechanism decides **which version** of the *Drive* function to apply in every concrete situation. The specific version of the function will be used depending on the current situation. This is an analogue to so-called *runtime polymorphism*.

Therefore, the car exhibits *polymorphic behaviour* when going in the forward direction, but exhibits *non-polymorphic behaviour* when going backwards.

> *Polymorphism is used to customise the behaviour of an instance of a type (an object) based on existing conditions.*

We can summarise this:

> *We live an object-oriented world. The object-oriented programming paradigm attempts to express computer programs in such ways that model how we understand the world.*

An introductory description of OOP can therefore be based on the following guideline:

> *The solution to the problem should resemble the problem, and observers of the solution should be able to recognise the problem without necessarily knowing about it in advance.*

A good example of this guideline from the computing world is the use of OOP to develop a **stack** class from which stack objects can be instantiated. If a stack is implemented as a **class**, instantiated as an **object**, and documented appropriately, programmers familiar with stacks, queues, and other similar data structures will recognise it as a stack without other prior knowledge.

2.2 Inheritance Example

Another important concept of OOP is *inheritance*. Let's form an analogy with a person who bought an old house. Having bought it, the person does not normally demolish old house and re-build it from scratch, but rather he will usually make improvements on it. In OOP terms it means the person *derives* from the existing house, or saying in other words, he *extends* it, so he is to build *subclasses* of the house that already exists. The new house *inherits* from the existing one.

Creating new improved objects using new definitions that *extend* existing definitions is very common practice. This is one of the mottos of OOP, that encourages the reuse of existing elements of programs.

Do not reinvent the wheel! Reuse the code!

2.3 Encapsulation Example

If we have decided to use a car for journey to the place, consider the steering mechanism of a car as an example of **encapsulation**.

From the time the car was invented, the steering mechanism has not changed functionality. It presents the same interface to the users; everybody knows how to use this mechanism through its *interface*. If we turn the steering wheel clockwise, the car will turn to the right, if we turn it counterclockwise, our car will turn to the left. Therefore in object-oriented paradigm this mechanism is an object. We can use it without having any idea about **implementation** — everything is hidden for us under the hood, only a few of us have more details. They even know that **implementation** of this mechanism is different for cars from different firms. The *interface* is naturally standard: in case of changing the mechanism by means of rental trailer we have some difficulties in using.

In turn, the steering mechanism object contains a lot of other more detailed *embedded objects*, each of which has it is own *state*, *behaviour*, *interface* and *implementation*. Those interfaces are not exposed to a driver, but to the other parts of the steering mechanism that use them.

A common approach in OOP is to hide the implementation and expose the interface through encapsulation.

2.4 A Word on OOP Vocabulary

The style of programming and the vocabularies between pre-OOP languages and OOP ones are different. Here you have an example: there are no *procedures* anymore, there are *class methods*.

Yet another example: an object-oriented programmer may define an *abstract data type* by *encapsulating* its *implementation* and its *interface* into a *class*.

One or more *instances* of the class can then be *instantiated*. An *instance* of a class is known as an *object*. Every *object* has *state* and *behaviour*.

The *state* is determined by the current values that are stored in the *instance variables*.

The *behaviour* is determined by the *instance methods* of the class from which the *object* was *instantiated*.

Inherited abstract data types are *derived classes* or *subclasses* of *base classes* or *superclasses*. We *extend superclasses* to create *subclasses*.

Within the program the programmer *instantiates objects* and sends *messages* to the *objects* by invoking the class' *methods* (or *member functions*). *Instantiate an object* means create an *instance* of *a class*.

An object oriented program defines *abstract data types*, *encapsulates* those abstract data types into *classes*, *instantiates objects*, and *sends messages* to the *objects*.

To make these words even more confusing, almost every item or action used in the OOP vocabulary has evolved to be described by several different terms, often depending on the author(s) who wrote the particular book your are happy to be reading. For instance, we can cause an object to *change its state* by:

1. *sending it a message*
2. *invoking its methods*
3. *calling its member functions*.

3 The Characteristics of an OOP Program

Having explained the main ideas of an object-oriented program, let's have a look at the question in detail.

3.1 What is An Object?

Let's consider the three general questions.

- What are objects in OOP?
- What should be the objects in the design of an object-oriented program?
- What is it about OOP that sets it apart from and possibly makes it better than traditional procedural programming?

Simply stated, and taking a *very liberal view:*

> An object is an **instance** of a data **type**.

The following Java code fragment declares two objects.

The first object is an instance of a simple *int*eger variable named **tired**. Some computer scientists may not agree that this is an object. Purists think an object should not be intrinsic to the language itself. This might be correct in general, but we are talking very liberally.

The second object is an instance of an *abstract data type* where the object is named **man** and the abstract data type is named **Person**.

```
...
 int tired;  // an instance of an int type
 Person man; // an instance of abstract data type
             // identified as Person
...}
```

As you know, or can surmise, the integer data type comes from types which are intrinsic in the language and named *primitive* types.

It is supposed that the abstract data type named **Person** has been invented by you and declared somewhere. From the language point of view, this is *a new data type.*

3.2 Abstraction

Abstraction is the specification of an abstract data type and includes a specification of the type's *data representation* and *behaviour*. It shows, what kind of data can be held in the new type of data, and all ways of manipulation of that data. An abstract data type is not intrinsic to the language, so compiler knows nothing about how to manipulate it until it is specified by the programmer in an appropriate manner.

Java programmers define the *data representation* and the *behaviour* of a new type (present the specification to the compiler) using the keyword **class**. It means, the keyword **class** is used to convert the specification of a new type into something that the compiler can understand and work with.

Once the new type is defined, one or more objects of that type can be put into existing state, from *abstraction* to reality. In other words, object of such kind can be *instantiated*.

Once instantiated, the object is said to have *state* and *behaviour*. The *state* of an object is determined by the current values of its data (instance variables) and the *behaviour* of an object is determined by its methods (member functions or instance methods).

A popular example is a button, as an element of a GUI. If a button is viewed as an object, we can visualised its state and behaviour easily. It has a number of different states like size, position, caption, etc. Each of these states is determined by data stored in the *instance variables* of the *button object* at any given point in time. The combination of one or more *instance variables* for the particular *state* of the object named a *property* of the object.

Similarly, when you click it with a mouse, that usually causes some *action* defined for the button.

Each individual button object has *instance variables*, the values of which define the *state* of the button at any given time, from one side, and has certain fundamental behaviour to respond to a click etc to use with some action.

3.3 Sample Class Definition

Now let's create a real example. We will try to create the program for modelling a human being. The Human class will describe it as triggering between two common states: working, and resting. The class should inform its current state, fit for work or be sleeping or celebrating. As *instance variables* of the class, we will use integer value `tired` and two strings, one to hold `name` and another for `origin`.

The corresponding line looks like this:

```
class Human {
  // code of the class
      ...
}
```

The key word in the definition is **class**, and its name is **Human**.

The *behaviour* of the new type is defined by three *instance methods*. One can be used to store a data in an object of the new type, it is named **setPerson**. The other one is named **getHumanInfo** can be used to retrieve a stored data from the object. The last one is named **Work** and implements the changing of the object **tired**, that is the member of a class **Human**. The code is not shown, we shall see it later.

The corresponding lines look like these:

```
// instance method to store data
    void setPerson (int state, String na, String orig) {
    ...
 }
// instance method to display info of a human
    String getHumanInfo() {
   ...
 }
// instance method Work
    String Work() {
   ...
 }
```

This new type we can expand by incorporating other behaviours by means of setting up additional methods.

Having defined the new type, we may create instances of this type, i.e. objects, and deal with those objects similar we would deal with any other variables created from the primitive data types.

4 Encapsulation

The first of the three major principles of an object-oriented program is encapsulation. On an abstract level we think in terms of independent agents working together. Then we *encapsulate* the *representation of the data* and *behaviour* into a *class*, thereby defining its *implementation* and *interface* .

According to good object-oriented programming practice, an encapsulated design usually *hides its implementation* from the class user and *reveals only its interface*. A seed of *clemantine* was created in such manner. Realisation of all its behaviour are hidden from us, we know only how to grow it, we know *the interface*.

God did not provide detail documentation on how seeds are implemented, but from the experience of human beings we know their interface and have the

class documentation about it's interface. In technique, an ordinary user does not usually need to know how the steering mechanism of a car is implemented, and do not know implementation details. A designer of *class* should have documentation about *implementation* to develop and possibly change the user-level documentation.

By the way, Java has a special mechanism to produce documentation that will be described in the next lectures.

To control access the members of a class, Java uses the keywords **public**, **private**, and **protected**. It is easy to surmise what the first two means. The last word, **protected**, is used to provide inherited classes with special access privileges to the members of their base classes.

Usually in a properly designed class the *user cannot modify* the values in the instance variables without going through the interface.

Do you remember the fragments of the code? Now let's have a look at the full source.

```
 1  class Human {
 2
 3      int tired;      // instance variables
 4      String name;    // of the class
 5      String origin;  //
 6
 7      // instance method to store data
 8      void setPerson (int state, String na, String orig) {
 9  tired = state;
10  name = na;
11  origin = orig;
12      } // end method setPerson
13
14      // instance method to display info of a human
15      String getHumanInfo() {
16  String info;
17          info = ("My name is " +
18  name + " and I am " +
19  origin + " ");
20          if (tired > 1)
21              info = info +
22  ("I am very tired.");
23          else
24              info = info +
25  ("and I am ready to work.");
26  return info;
27      } // end method showHumanInfo
28
29      void drinking(){
30       System.out.println(
```

```
31                "I like drink with friends" );
32     }//end drinking()
33
34      // instance method Work
35      String Work() {
36          if (tired < 2) {
37              tired ++;
38              return
39  ("Ouff... working again...");
40          }
41          else
42              return
43  ("Sorry can't work, being too exhausted.");
44      } // end method Work
45
46      String rest() {
47          if (tired < 1)
48              return
49  ("Not very tired, so going to a party.");
50          else {
51              tired --;
52              return
53  ("Sleeping, hhhharrarrrdrrhhhh...");
54          }
55      }
56  } // end class Human definition
57
58  // Driver program
59
60  class ex1 { // defining the controlling class
61      public static void main(String[] args) { // define main
62
63  Human obj = new  Human ();
64  obj.setPerson(0, "Olexiy", "Ukrainian"); //store data
65  // Get info about Olexiy
66  System.out.println ( obj.getHumanInfo() );
67      } //end main
68  } //end ex1 class
```

In general, to be visible to a user, the class consists of the *public* methods. In our program all methods as well as classes are **public** by default. The class user stores, reads and modifies values by invoking those methods. In the program above we used method **setPerson** to input information about me, and than **getHumanInfo** to get information about me.

The program we are examining has no *hidden the implementation*. From the OOP point of view, this is not a well-designed program, but it is too simple!

Note, in our program we used methods **setPerson** and **getHumanInfo**. There

is a special note about names of the methods started with **set** and **get**:

> *Methods whose names begin with set and get have a special meaning in Java. In particular, the introspection capability of the Java Beans API considers these names to represent design patterns for manipulating the properties of an object.*

An object-oriented design is *not a good design by default*. In an attempt to produce good designs, there are some general agreements on certain design standards for classes. For instance, the data members (instance variables) are usually **private**. The interface usually consists only of methods and includes few if any data members. This is, of course, a way of hiding the implementation.

However, there is one exception to this general rule: the data members which are going to be used as symbolic constants are made public and defined to disallow modifying their values.

The methods in the interface should control access to, or provide a pathway to, the private instance variables. The interface should be generic as possible, in that it is not bound to any particular implementation. It means, from the practical point of view, that the arguments of the method should have the same meaning. If for some reasons changing implementation is needed, it should be done in a such way to avoid changing the interface.

In our program we have only instance variables and instance methods. Let us have a look at class variables and class methods.

4.1 Instance and Class Variables

In our program we have only instance variables and instance methods. We have to discuss about *class variables* and *class methods*. What are they? Why we need them?

Instance variables are defined such that each instance of the class, or object has its own set of variables. These variables are fully independent and stored in their own memory space. To get access to an instance variable the joining operator is used, in Java it is a simple period called *dot operator*. E.g., in our program we can access the **name** of the object **obj** as **obj.name**.

It is possible to surmise:

> *Class variables are shared among all objects of the class. They are very similar to global variables. Only one copy of a class variable exists in memory and all objects of the class can access them.*

A very important characteristic of class variables is, that they can also be accessed even if no object of the type was instantiated. You can access them with *dot operator*, joining class name and the variable.

4.2 Instance and Class Methods

The methods of a class come in two varieties:

1. Instance methods;
2. Class methods.

Knowing the situation with instance and class variables, you can guess easily: methods designated *static* are *class* methods, and non-static are *instance* ones.

An instance method can only be invoked with an object of the class, so it is bound to an object. If we invoke an instance method on a particular object, the method will access the instance variables belonging to the object on which it was invoked. It is very important to know, that the methods of a class have direct access to the member variables of the same class, paying no attention to their control access like *public*, *private*, *protected*.

Class methods can only access other class members (class variables or other class methods). They cannot access instance variables or instance methods.

The most important thing about class methods is that they can be accessed using the name of the class without a requirement to instantiate an object of the class. As with class variables, class methods can be accessed by joining the name of the class to the name of the method using the appropriate joining operator.

4.2.1 Sample Program

Much of what we have been discussing can probably be better understood when seen in the context of an actual program. The program will produce the following output on the standard output device:

```
---------------
This cat has child(ren): 100
This cat has legs: 4
---------------
This cat has child(ren): 1220
This cat has legs: 4
```

Before we take a look at the listing of the program, lets examine some of the *interesting code fragments* that make up the program.

The first interesting code fragment shows the declaration of two member variables of the class. One is a class variable named **child** and the other is an instance variable named **legs**.

```
int child;     // declare an instance variable 'child'
static int legs; // declare a class variable 'legs'
```

These are typical variable declaration statements in Java: consisting of the name of the type followed by the name of the variable. The important thing to note in the context of this discussion is the use of the `static` keyword in the declaration of the class variable.

The next code fragment shows the definitions of two methods (with the bodies of the methods deleted for brevity). One of these methods is a class method named **classMethod** and the other is an instance method named **instanceMethod**.

```
  void { instanceMethod() { // define an instance method
 ........  // body of method deleted for brevity
 } //end instanceMethod()

 static void classMethod() {   //define} a class method
 ........ //body of method deleted for brevity
 } //end classMethod()
```

These are typical method or member function definitions in Java, consisting of the name of the return type (*void* means nothing is returned) followed by the name of the method, followed by the formal argument list (is empty in this case). The body of the method is then enclosed within a matching pair of curly braces { }.

Now you can guess easily: the important thing to note is the use of the `static` keyword in the definition of the class method.

The next code fragment is a single statement taken from the body of one of the methods. This statement causes output to be displayed on the standard output device.

This single statement incorporates classes, class variables, instance methods, and overloaded operators, and illustrates some of the syntactical complexity in an object-oriented program.

```
System.out.println("This cat has legs: "
                   + legs);
```

This statement has three elements joined with periods.

The first element is the word `System` which is the name of one of the classes in the standard Java class library. As background information, the **System** class is loaded automatically whenever a Java application is started.

The name of the `System` class is joined to the word `out` using a period. The word `out` is the name of a member variable of the **System** class.

The member variable named `out` is a public class variable. This makes it possible to access the variable using the name of the class and the name of the variable joined by the period.

Note that the class variable named **out** is also joined to the word `println` using the period as the joining operator. The variable `out` is not only a *class* variable, it is also a *reference* variable (as opposed to a *primitive* variable) and it contains a reference to an *object* of the **PrintStream** class.

The **PrintStream** class has an *instance method* named **println()**. In fact, there are ten overloaded versions of the **println()** method in the **PrintStream** class. The behaviour of the version used here is to cause its string argument to be displayed on the standard output device.

Now consider the string argument to the **println** method as shown below:

```
("This cat has child(ren): "
                          + child)
```

In Java, as in C, literal strings are enclosed in quotation marks. The plus sign is *overloaded* in Java in this way to being used as an arithmetic addition operator, and also to concatenate strings. Even more, the behaviour of the overloaded *plus* operator also includes the ability to coerce its right operand into a string representation if it isn't already a string. In this case, the right operand is not a string, but rather is the instance variable named **instanceVariable**. Thus the behaviour of the overloaded *plus* operator is to first convert the value of **instanceVariable** to a string representation and then to concatenate it to the left operand.

Now let's take another look at the same two methods as before, this time preserving the bodies of the methods for further examination.

```
    // instance method
    void instanceMethod ( ) {
        System.out.println("This cat has child(ren): "
                           + child);
    } // end instanceMethod
    static  void classMethod ( ) {
        System.out.println("This cat has legs: "
                           + legs);
    } // end classMethod
```

Here we see the code in the body of the methods accessing the member variables of the class. Recall that one of the member variables is an instance variable named **child** and the other member variable is a class variable named **legs**.

The instance method named **child** is able to access and display both the instance variable and the class variable while the class method named **classMethod** is only allowed to access and display the class variable. Class methods cannot access instance variables.

```
class ex2 { // defining the controlling class
    public static void main(String[] args) { // define main
     ..........  // code
    } //end main
} //end ex2 class
```

Now consider the contents of the **main** method as shown below. Java applications (but not applets) require a `main` method or function as the controlling method of the application. In our simple application, we will use code in the **ex2** method to instantiate an object and to access both the instance method and the class method.

Recall that in order to access an instance method, it is necessary to access it via an object of the class. The next code fragment is the code in the **main** method, that instantiates an object named **mour** of the class named `cat`.

```
 cat mour  = new cat();
```

This is a typical Java statement for instantiating an object.

The **new** operator requests the operating system provide memory "from the heap" to store one copy of an object of type `cat`.

If the required memory is successfully allocated, the address of that block of memory will be assigned to the *reference variable* named **mour**. If unsuccessful, the Java runtime system will *throw an exception.* This is a type of exception which can either be ignored, or can be caught and processed by the program. If ignored, it will cause the runtime system to shut down the program. Exception handling is discussed later.

Once we have access to an object of the class (or more correctly access to a reference variable which refers to an object of the class), we can use that reference variable to access the public member variables and to invoke the public methods of the class. This is illustrated in the following code fragment.

The two statements in the following code fragment use the reference variable named **mour** along with the period to access the instance variable and the instance method of the object. Recall that the instance variables and the instance methods can be accessed only via an object of the class.

```
// access instance variable via objects:
        mour.child=100;
//access instance method via the object
        mour.instanceMethod();
```

Equally important is the fact that the class variable and the class method can be accessed without the requirement to use an object of the class. The two statements in the following code fragment simply use the name of the class to access the class variable and the class method of the class.

```
        // access class variable via the class
        mour.legs = 4;
        // access class method via the class
        mour.classMethod();
```

Class variables and class methods can be accessed either via an object of the class, or via the name of the class alone. Note, we do not put *another* value for class variable "leg": this value is shared between two object of the class `cat`: `meow` and `mour`.

Finally, we put it all together in the Java application.

```
1  /* Copyright 2001, O.Ye. Tykhomyrov
2     The second example, ex2.java
3     Illustrates instance and class variables along with
4     instance and class methods.
5  */
6
7  class cat {
```

```
 8      int child;       // instance variables 'child'
 9      static int legs; // class variable 'legs'
10
11      // instance method
12      void instanceMethod ( ) {
13  System.out.println("This cat has child(ren): "
14     + child);
15      } // end instanceMethod
16      static  void classMethod ( ) {
17  System.out.println("This cat has legs: "
18     + legs);
19      } // end classMethod
20  }
21  // Driver program
22
23  class ex2 { // defining the controlling class
24      public static void main(String[] args) { // define main
25
26  // instantiate an object of the class cat
27  cat mour  = new cat();
28  cat meow  = new cat();
29  // access instance variable via objects:
30  mour.child=100;
31  meow.child=1220;
32  // access class variable via the class
33  mour.legs = 4;
34
35  System.out.println("---------------");
36  // access instance method via the object
37  mour.instanceMethod();
38  // access class method via the class
39  mour.classMethod();
40  System.out.println("---------------");
41  meow.instanceMethod();
42  meow.classMethod();
43      } //end main
44  } //end ex2 class
```

4.3 Messages

Methods are sometimes called member functions.

A message is simply the invocation of a method or member function.

The program *sends a message* to an object telling it to invoke the method and sometimes provides parameters for the method to use.

Someone recently wrote that an object-oriented program consists simply of a

bunch of objects laying around sending messages to one another. This might be a slight exaggeration, but is not too far from the truth.

4.4 Constructors, Destructors, and Garbage Collection

The allocation, reclamation, and reuse of dynamic memory from the *heap* is an important aspect of most object-oriented programs, and some non-object-oriented programs as well.

Any particular language or language platform, like Java virtual machine, should do some operation in memory:

- to allocate memory when it is needed by the program,
- to reclaim that memory when it is no longer needed, and
- to reuse it as appropriate.

Failure to deal with this important issue results in a condition often referred to as "memory leakage."

Constructors are used to instantiate and possibly initialise an object:

```
// instantiate an object of the class cat
cat mour  = new cat();
```

Constructors can be *overloaded* just like other methods in Java. Overloading will be discussed later. Briefly, method overloading means that two or more methods can share the same name. Compiler determines version depends on the list of actual parameters.

In this particular statement, the **new** operator is used to allocate dynamic memory from the heap, and also as the constructor to construct the object in that memory space. The address of the memory containing the object is returned and assigned to the reference variable named `mour`. If the memory cannot be allocated, an *exception* will be thrown.

In Java you do not define a constructor when you define a new class, a default constructor that takes no parameters is defined on your behalf.

You can also define your own constructor with no argument needed. Defining a constructor is similar to defining a method, but must have the same name as the class, do not have a return type and must not have return statements.

The following code fragment shows the important parts of a Java program, similar to the previous one which has been modified to use a parameterised constructor.

```
...
   String name;
  // Parameterised constructor
  cat (String n) {
      name=n;}
...
      System.out.println("This cat named " +
                         name +
                         " has child(ren): "
                         + child);
...
 cat mour = new cat("Pirat");
...
```

Destructors A destructor is a special method typically used to perform cleanup after an object is no longer needed by the program. C++ supports destructors, but Java does not support destructors.

Java supports another mechanism for returning memory to the operating system when it is no longer needed by an object.

Garbage Collection The garbage collector is a part of the runtime system that runs in a low-priority thread reclaiming memory that is no longer needed by objects used by the program. An object becomes *eligible* for garbage collection in Java when there are no longer any reference variables that reference the object.

5 Inheritance

The first major characteristic of an object-oriented program is **encapsulation**. The second one is **inheritance**. Let's now have a look at it.

Having bought an old house to live, the person may try to *reconstruct* it in order to have it *extended* into another one, more modern and more comfortable without ruining the old version in general. Thus after reconstruction the house will be a *subclass* the house that already existed. The new house *inherits* from the existing one.

The same with the OO program: a class can normally inherit the attributes and characteristics of another class. This mechanism can be blocked by using different ways, though.

The original class is often called the *base class* or the *superclass*, and the new class is often called the *derived class* or the *subclass*. Inheritance is often referred to as *extending* the base class or superclass.

Inheritance is hierarchical. In other words, a class may be the subclass of one class and the superclass of another class.

The derived class inherits the data representation and behaviour of the base class except where the derived class modifies the behaviour by overriding methods. The derived class can also add new data representation and behaviour that is unique to its own purpose.

A program can usually instantiate objects of a base class as well as of a class which is derived from the base class. It is also possible to block instantiation of the base class in some cases by defining it as an *abstract base class.* If the base class is an *abstract base class* — one that exists only to be derived from – the program may not instantiate objects of the base class but can instantiate objects of classes derived from the base class.

The Java inheritance mechanism allows to build an orderly hierarchy of classes.

When several of your abstract data types have characteristics in common, you can design their commonalities into a single base class and separate their unique characteristics into unique derived classes. This is one of the purposes of inheritance.

Remember, we have created a class **Human**, but not all humans are the same, but different: we have different race, sex, culture, religion etc. Nonetheless we have some very common features: two legs, two arms, one head. We all can work, sleep, etc. Developed class **Human** has methods to indicate a human state concerning possibility of working. This is the common parameter therefore of the **Human** class we have built.

From this base class, we may derive a **Gentleman** class and a **Lady** class. They certainly have different rest: the **Lady** prefers to drink tea with her neighbours but the **Gentleman** largely his pint of beer in his favourite pub.

Objects of the classes will then be able to deal with all **Human** parameters as well as new ones.

You may have noticed that in this hierarchical class structure, inheritance causes the structure to grow in a direction from most general to less general. This is typical.

Here you are an example. In the next fragment of the code one line of the code is essential: it describes the **Lady** and **Gentleman** *subclass*es:

```
class Lady extends Human {
...
}
class Gentleman extends Human {
...
}
```

The keyword *extends* shows the **Lady** and **Gentleman** classes are *subclass*es of the **Human** class.

Now we might have a look at the next fragment of the code, that was replaced ....

```
    // a new method
    public void drinking(){
```

```
        System.out.println
            ("Drinking tea with my neighbors");
    }
```

In this part of the program we declared a new method, **drinking**. Of course, the very similar method the **Gentleman** has, but about beer!

Yet another sensitive fragment of the code shows manipulating with method declared in the base class, **Human**. The keyword **super** is used to access that method following by period, or *dot operator*.

```
   // changing mathod getHumanInfo
    public String getHumanInfo() {
        String info;
        info=super.getHumanInfo();
        info=info+"and I am a lady.";
        return info;
    }
```

The last fragment of the code shows how to use new classes. Nothing new, isn't it?

```
...
        Gentleman me   = new Gentleman();
        Lady      anne = new Lady();
        me.setPerson (0, "Olexiy", "Ukrainian"); //store data
        you.setPerson(0, "Anne", "Russian");  //store data
        System.out.println( me.getHumanInfo() );
        System.out.println( you.getHumanInfo() );
...
```

5.1 Single and Multiple Inheritance

Some object-oriented languages like C++ allow for so-called multiple inheritance. This means that a new class can be derived from *more than one* base class. This has advantages in some cases, but can lead to difficulties in other cases. Java does not support multiple inheritance but provides a special mechanism called an *interface* to provide the same result. It is mentioned simply to inform you that OOP language may, or may not provide mechanisms to achieve the same end result.

5.2 The is-a Relationship

You may hear sometimes people speak of the **is-a** relationship. The source of this terminology is more fundamental than you may at first suspect.

Object-oriented designers often strive to use inheritance to model relationships where a derived class *"is a kind of"* the base class. For example, a lady *"is a kind of"* human. A wife *"is a kind of"* lady etc.

This form of relationship is called the **is-a** relationship.

6 Polymorphism

The last required characteristic of an object-oriented language is **polymorphism**.

The word **polymorphism** is Greek by origin. It means something like "one thing, many forms". In OOP polymorphism represents the idea of "one interface, multiple methods" and means that functions or operators not implicitly recognised by the compiler. Java does not support operator overloading, but does support the overloading and overriding of methods.

6.1 Overloading Methods as a Form of Polymorphism

The best-known example of polymorphism is the ability to replace the three C functions,

- `abs()`
- `labs()`
- `fabs()`

by a single Java function called `ABS()`.

C requires three different functions which depends on the data types. In Java only one *name* of the function is needed for any data types, `int`, `long`, `float`. It is said in this situation that *function or method is overloaded*. This situation is recognised by a compiler, and polymorphism of this kind is called *compile-time polymorphism*. Some authors say also about *early binding* or *static binding* in this case. They usually do not consider function overloading to be a "true" polymorphism.

6.2 Overloading of Operators

Polymorphism of this kind is available in many non-OOP languages. In C, for instance, we use "+", plus operator to add *any* kind of data. This form of polymorphic behaviour has been *intrinsic to the language* and could not be modified or controlled by the programmer.

C++ provides the opportunity, and (with one or two exceptions) the responsibility for the programmer to define the behaviour of almost any operator that may be applied to an object of a new class.

Java does not support operator overloading although in many cases overloaded operators will provide a much cleaner and more intuitive syntax.

6.3 Overriding Methods as a Form of Polymorphism

The words *override* and *overload* write and sound differently from each other and have difference in meaning. *Overriding* a method is an entirely different thing from *overloading* a method.

The overloading methods and overloading operators is not the end of the story. If a derived class *customises* the behaviour of methods defined in the base class,

usually to meet the special requirements of the derived class, we say about *run-time polymorphism*.

Java supports the notion of *overriding a method* in a base class to cause it to behave differently relative to objects of the derived class. In other words, a method named **drinking()** that is defined in the base class and is overridden in the derived class would behave differently depending on whether it is invoked by an object of the base class or by an object of the derived class.

In Java reference variable of a base-class type can be used to reference an object of any class derived from that base class.

If an overridden method is invoked using that reference variable, the system will be able to determine, at runtime, which version of the method to use based on the true type of the object, and not on the type reference variable used to invoke the method. This fact is illustrated in the following Java program.

The first, an overridden method named **drinking()** as a base-class reference to a base-class object is invoked, then, as the second, the overridden method named **drinking()** is invoked on a derived-class reference to a derived-class object.

In the first case, the base-class version of **drinking()** is actually invoked. In the second case, the derived-class version of **drinking()** is invoked. This is trivial.

Then as the third, is invoked the overridden method named **drinking()** on a base-class reference which *refers to a derived-class object*. When this is done, it is the version of the method defined in the derived class and *not* the version defined in the base class that is actually invoked. **This is the essence of runtime polymorphism.**

```
 1  //  This program illustrates the run-time polymorphism.
 2  //   Output of the program should be as followed:
 3  //  My name is null and I am null and I am ready to work.
 4  //  I like drink with friends
 5  //  My name is Olexiy and I am Ukrainian and I am ready to work.
 6  //   and I am a Gentleman.
 7  //  I like drinking beer in my favorite pub
 8  //  My name is Olexiy and I am Ukrainian and I am ready to work.
 9  //   and I am a Gentleman.
10  //  I like drinking beer in my favorite pub
11
12  class ex4 {
13      public static void main(String args[]) {
14  // Human class ref to Human class object
15  Human     obj  = new Human();
16  // Gentleman class ref to Gentleman class object
17  Gentleman me   = new Gentleman();
18  // set information about me:
19  me.setPerson (0, "Olexiy", "Ukrainian");
20
21  // Display informatio about obj:
```

```
22  System.out.println( obj.getHumanInfo() );
23  // invoke method named drinking:
24  obj.drinking();
25
26  // Display information about me:
27  System.out.println ( me.getHumanInfo() );
28  // invoke method named drinking:
29  me.drinking();
30
31  // Human class ref to Gentleman class object
32  obj = me;
33
34  // Display information about obj:
35  System.out.println (obj.getHumanInfo() );
36  // invoke method named drinking:
37  obj.drinking();
38      }
39  }
```

Inheritance and *method overriding* are used in almost all Java programming. Even the well-known "Hello World" Java applet requires that the **Applet** class be extended and the **paint()** method be overridden.

7 Exception Handling

Although *exception handling* may not be considered as an OOP principle, Java operates and requires it. Therefore, it is useful to speak about it a little in a general sense.

We will attempt to look at the following topics briefly:

- What is an exception?
- How do we throw and catch exceptions?
- What do we do with an exception once we have caught it?
- How do we make use of the exception class hierarchy provided by the development environment?
- Will we have advantages with exception handling?

7.1 Exception Hierarchy

Exception can be defined by the following sentence:

> *An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.*

Exception-handling capability makes it possible to monitor exceptional conditions within a program, and to transfer control to special exception-handling code designed by a programmer whenever an exceptional condition is detected.

First, we **try** to execute the statements contained within a block surrounded by braces.

Then, if an exceptional condition is detected within that block, a programmer's code or the runtime system itself **throws** an exception object of a specific type. In Java that exception is an object derived, either directly, or indirectly, from the class **Throwable**.

The **Throwable** class in Java has two subclasses:

Error indicates that a *non-recoverable* error has occurred that *should not be caught*. Errors usually cause the Java interpreter to display a message and exit.

Exception indicates an abnormal condition that *must be properly handled* to *prevent program termination*.

Of all possible exceptions that Java can throw automatically, there is a subset for which catching and processing is optional. The compiler allows you to ignore the exceptions of this subset. If an actually ignoring exception occurs, it will be *caught* by the runtime system, and the program will be terminated.

The remaining exceptions, that can automatically be thrown in Java, must be *recognised* by program's code in order to compile your program.

Recognised means the program's code *catch* and either process the exception object using another piece of the code, or your code can pass it up to the next level in the method-invocation hierarchy for handling there.

Java program can then optionally execute a block of code designated by *finally* which is normally used to perform some type of cleanup which is needed whether or not an exception occurs.

If a method passes an exception up to the next level in the invocation hierarchy, this must be declared along with the method signature using the **throws** keyword.

If your code catches and processes an exception, the processing code can be as elaborate, or as simple as you want to make it. The fact is, simply ignoring it after you catch it will satisfy the compiler. This may, or may not be a good idea, depending on the type of the exception.

All exception objects inherit the methods of the **Throwable** Java class.

7.2 Advantages of Using Exception Handling

Exception handling provides the following advantages over "traditional" error management techniques:

separating Error Handling Code from "regular" one provides a way to separate the details of what to do when something out-of-the-ordinary happens from the normal logical flow of the program code;

propagating Errors Up the Call Stack lets the corrective action to be taken at a higher level. This allows the corrective action to be taken in the method that calling that one where an error occurs;

grouping Error Types and Error Differentiation allows to create similar hierarchical structure for exception handling so groups they in logical way.

8 Distributed Objects with CORBA and RMI

The same way we can build a house buying needed pieces from *different* vendors, it is possible to make software that works through the net using software written in *different languages* and running on *different platforms*. CORBA helps us to do that.

CORBA is the acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture. CORBA is open, vendor-independent specification for an architecture and infrastructure that computer applications use to work together over networks. It allows a CORBA-based program from any vendor on almost any operating system, programming language, and network, to interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

CORBA is useful in many situations. Because CORBA helps integrating machines from many vendors with different computation power, it is the common choice for large (and even not-large) enterprises. One of its most important strengths is the ability to handle large amount of clients which results in wide deployment on web servers. CORBA works for many of the world's largest websites. Specialised versions of CORBA can run real-time systems, and small embedded systems.

CORBA provides this way in general. Java has a similar feature called RMI to do more or less the same as CORBA.

RMI gives your objects the ability to communicate and exchange data over a network. So it becomes quite easy to setup a client/server architecture for a large distributed system. RMI stands for **R**emote **M**ethod **I**nvocation.

Communication via RMI and CORBA is most likely not very fast, so it is not applicable for time critical tasks, but for remote monitoring where the response time is not that critical, it would be a nice solution.

CORBA applications are composed of objects, individual units of running software that combine functionality and data, and that frequently (but not always) represent something in the real world.

So, the main idea of both CORBA and RMI is the physical independence of an object's *implementation* and *interface* while keeping the logical uniformity.

8.1 CORBA Versus RMI

Although RMI and CORBA are used to achieve more or less similar end results, they are not compatible!

With RMI the user application as well as remote object implementation **must** be implemented in Java. The interface definition is just a regular Java interface. RMI provides a lightweight, Java-specific mechanism for objects to interact

across the network **and** the tools to make them work together. The tools, classes, etc. are included within the Java virtual machine.

CORBA, on the other hand, is a specification and an architecture for the integration of networked objects. CORBA ensures interoperability of these distributed objects. Different vendors follow the CORBA specification to provide tools, libraries, etc., for multiple languages and platforms. CORBA implementation for Java is provided by *Sun Microsystems*.

9 Introduction to Object-Oriented Design

Object-oriented design (OOD) is a technology in its own right, often treated as being separate and apart from OOP, but better to think about OOP as about the implementation of OOD procedure.

Many books have been written describing a variety of methods for accomplishing OOD and here we are not going to discuss and cover all topics of that. As a physicist, I think about programming as *a way to help us in solving some particular problem*, not as a theory in general. Standard way, however, to show how to create an OO program is to explain, how to make *first pass* to identify *classes data members* and *methods* to build the OO program.

One common technique is to create a *narrative description* of the solution to the problem, then use *nouns* and the *verbs* to identify the *classes* , *data members* , and *methods* needed in the design.

Consider the following narrative description of the solution to an object-oriented design problem. We will use this description to design a simple object-oriented program. The nouns will give us hints about the *classes* and the *data members* that will be needed, and the *verbs* will give us an idea of the *methods* that will be needed.

Design a digital Counter which has two bits with the following abilities:

1. *Initialising* the Counter means to set its values to **00**;
2. *Incrementing* the Counter by one;
3. *Displaying* the Counter means to show its contents.

For each bit the followings abilities should be provided:

1. *to set* the value of the bit to a 0;
2. *to get* the value stored in the bit;
3. *to add* a 1 to the value stored in the bit and return the carry according to the truth-table that follows:
   0 + 0 = 0, c = 0
   0 + 1 = 1, c = 0
   1 + 0 = 1, c = 0

1 + 1 = 0, c = 1

A test program should instantiate a Counter object and exercise the Counter by incrementing it 4 times and displaying the result.

The output from the test program should be:
00
01
10
11

The program may look like this:

```
 1  /* Counter designing */
 2
 3  class Bit{
 4      int value;
 5      void set(){//method to set the value of the bit to 0
 6  value = 0;
 7      }//end set()
 8
 9      int get(){//method to get the value stored in the bit
10  return value;
11      }//end get()
12
13      //method to implement binary addition
14      // we provide the table here:
15      int add(int inValue){
16  int carry = 0;
17  if((value == 0) && (inValue == 0)){      //0+0=0,c=0
18      carry = 0;
19  }else if((value == 0) && (inValue == 1)){//0+1=1,c=0
20      value = 1;
21      carry = 0;
22  }else if((value == 1) && (inValue == 0)){//1+0=1,c=0
23      carry = 0;
24  }else if((value == 1) && (inValue == 1)){//1+1=0,c=1
25      value = 0;
26      carry = 1;
27  }//end if statement
28
29  return carry;
30      }//end add()
31  }//end class Bit
32
33  class Counter{
```

```
34      Bit fstbit = new Bit(); //instantiate two bit objects
35      Bit scndbit= new Bit();
36
37      //method to initialize the bit objects to 0
38      void initialize(){
39  fstbit.set();
40  scndbit.set();
41      }//end initialize()
42
43      //method to add 1 to lsb and have it ripple up
44      // through all three bits of the counter
45      void increment(){
46  scndbit.add(fstbit.add(1));
47      }//end increment()
48
49      //method to display the value of each bit in the counter
50      void show(){
51  System.out.println("" + scndbit.get() + fstbit.get() );
52      }//end show()
53
54  }//end class Counter
55
56  //controlling class required by Java application
57  class ex5 {
58      public static void main(String[] args){
59  //instantiate a counter object
60  Counter myCounter = new Counter();
61  myCounter.initialize();//initialize the counter object
62  myCounter.show(); //display contents of counter object
63  //increment and display counter object
64  for(int cnt = 0; cnt < 3; cnt++){
65      myCounter.increment();//increment it
66      myCounter.show();//display it
67  }//end for loop
68      }//end main()
69
70  }
71
```

The program includes a class named **Bit** with a data member named **value** as well as methods to set the instance variable, get the instance variable, and add a **1** to the value of the instance variable returning the carry from the binary addition. These items you can find as nouns or verbs in the description.

Also as you can see, the design resulted in a class named Counter which had three embedded Bit objects as instance variables or data members. This class has a method named initialise that is used to initialise the two-bit counter to **00**.

This class also has a method named increment which adds one to the least-

significant bit, adds the carry from the least significant bit to the middle bit, and adds the carry from the middle bit to the most-significant bit. This is the typical ripple pattern for a binary counter.

This class also has a show method that displays the values stored in each of the two Bit objects in the order from most-significant to least-significant.

This program also contains a class named **ex5** which is not represented by a noun in the narrative description. This is because all applications in Java require a controlling class, and in this case the controlling class is named **ex5**.

10 A few final remarks on OOP

OOP requires a new style of program design. Common behaviour and properties must be identified and described in several class hierarchies. Once the class hierarchies are defined, most of the work is done. New instances of these objects are created and their methods called.

Class Design Principles:

In the design of classes one is primarily concerned with information hiding and data access. Parna's Principle states that "a class should only reveal to its users, what it has to, and no more". This makes a class more secure, protects its data from misuse and simplifies its interface. The arrtibutes of a class must be hidden by declaring them as private. Access to the variables should be via suitable accessor and mutator methods. Methods that are not seen outside should also be hidden.

The advantages of OOP are:

- Code re-use and uniqueness by inheritance and encapsulation;
- Maintainability: changes in a superclass are seen by all subclasses;
- Independence of code by encapsulation. Only the object's interface is seen by the outside world, not the implementation details;
- High degree of organisation and modularity of the code. This is especially important for large projects;
- Makes you think before coding.

The disadvantages are:

- Compiled programs are usually larger because the inheritance mechanism must be implemented at runtime.
- Compiled programs are usually slower because inherited code must be looked up when called by sub-classes.

References

[1] Bruno R. Preiss (2000). *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.*ISBN: 0-471-34613-6, University of Waterloo

[2] Peter Coad & Edward Yourdan *Object Oriented Design (Yourdon Press Computing Series)* (1991).

[3] Mary Campione, Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet* (2nd Edition)

[4] *The Java tutorial. A practical guide for programmers*. Sun Microsystems. Available online at `http://java.sun.com`

[5] *The Java tutorial. A practical guide for programmers*. Borland USA. Available online at `http://www.borland.com/delphi/`

[6] *Richard Baldwin*, `http://www.dickbaldwin.com/index.html`

[7] *Paul Ramos* Available online at `http://ref.web.cern.ch/ref/CERN/Tutorial/java/basicoop_v2/` for CERN users