МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра иностранных языков по техническим специальностям

Сборник аутентичных текстов на английском языке

COMPUTER TECHNOLOGIES

для студентов дневной формы обучения по специальностям
1-36 04 02 «Промышленная электроника», 1-40 02 01 «Вычислительные машины, системы и сети», 1-40 03 01 «Искусственный интеллект», 1-53 01 02 «Автоматизированные системы обработки информации»

Брест 2014

УДК 004.3'2(072)=111

Сборник аутентичных текстов на английском языке предназначен для студентов дневного обучения специальностей 1-36 04 02 «Промышленная электроника», 1-40 02 01 «Вычислительные машины, системы и сети», 1-40 03 01 «Искусственный интеллект» и 1-53 01 02 «Автоматизированные системы обработки информации», продолжающих изучение английского языка.

Основной целью сборника является совершенствование навыков чтения и понимания аутентичной научно-технической литературы по изучаемым специальностям, а также развитие навыков перевода.

Данный сборник включает три раздела, каждый из которых представлен несколькими текстами по специальности. Кроме того, сборник включает в себя тексты для дополнительного чтения и словарь.

Сборник аутентичных текстов на английском языке одобрен на заседании кафедры иностранных языков по техническим специальностям и рекомендован к изданию.

CONTENTS

Unit I. Computer Programming

Text 1. Computer Programming

Computer programming (often shortened to programming, scripting, or coding) is the process of designing, writing, testing, debugging, and maintaining the source code of computer programs. This source code is written in one or more programming languages (such as Java, C++, C#, Python, etc.). The purpose of programming is to create a set of instructions that computers use to perform specific operations or to exhibit desired behaviours. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

From the moment you turn on your computer, it is running programs, carrying out instructions, testing your RAM (Random-access memory), resetting all attached devices and loading the operating system from hard disk or CD-ROM.

Each and every operation that your computer performs has instructions that someone had to write in a programming language. These had to be created, compiled and tested- a long and complex task. An operating system like Microsoft's Windows Vista took millions of man hours to write and test the software.

There have been many attempts to automate this process, and have computers write computer programs but the complexity is such that for now, humans still write the best computer programs.

To write a program, software developers usually follow these steps:

1. First they try to understand the problem and define the purpose of the program.
2. They design a flowchart, a diagram which shows the successive logical steps of the program.
3. Next they write the instructions in a high-level language (Pascal, C, etc.). This is called coding. The program is then compiled.
4. When the program is written, they test it: they run the program to see if it works and use special tools to detect bugs, or errors. Any errors are corrected until it runs smoothly. This is called debugging, or bug fixing.
5. Finally, software companies write a detailed description of how the program works, called program documentation. They also have a maintenance program. They get reports from users about any errors found in the program. After it has been improved, it is published as an updated version.

Within software engineering, programming (the implementation) is regarded as one phase in a software development process.

As time has progressed, computers have made giant leaps in the area of processing power. This has brought about newer programming languages. Popular programming languages of the modern era include C++, C#, Visual Basic, SQL, HTML with PHP, Perl, Java, JavaScript, Python and dozens more. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code. However, high-level languages are still impractical for a few programs, such as those where low-level hardware control is necessary or where maximum processing speed is vital.

Computer programming has become a popular career in the developed world, particularly in the United States, Europe, Scandinavia, and Japan. Due to the high labour cost of programmers in these countries, some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities for programmers in less developed areas, particularly China and India.

Quality requirements. Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- ✓ Reliability: how often the results of a program are correct.
- ✓ Robustness: how well a program anticipates problems not due to programmer error.
- ✓ Usability: the ergonomics of a program: the ease with which a person can use the program for its intended purpose or in some cases even unanticipated purposes.
- ✓ Portability: the range of computer hardware and operating system platforms on which the source code of a program can be compiled / interpreted and run.
- ✓ Maintainability: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments.
- ✓ Efficiency/performance: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better.

Readability of source code. In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability.

Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code.

Methodologies. The first step in most formal software development processes is requirements analysis, followed by testing to determine value modelling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis. Nowadays many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the software development process.

Popular modelling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modelling Language (UML) is a notation used for both the OOAD and MDA.

A similar technique used for database design is Entity-Relationship Modelling (ER Modelling). Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Debugging. Debugging is a very important task in the software development process, because an incorrect program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static code analysis tool can help detect some possible problems.

Text 2. Programming Languages

Programming languages provide various ways of specifying programs for computers to run. Unlike natural languages, programming languages are designed to permit no ambiguity and to be concise. They are purely written languages and are often difficult to read aloud. They are generally either translated into machine code by a compiler or an assembler before being run, or translated directly at run time by an interpreter. Sometimes programs are executed by a hybrid method of the two techniques.

A programming language is used to write computer programs such as:

- Applications;
- Utilities;
- Servers;
- Systems Programs.

A program is written as a series of human understandable computer instructions that can be read by a compiler and linker, and translated into machine code so that a computer can understand and run it.

Different programming languages support different styles of programming (called programming paradigms). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute. Languages form an approximate spectrum from "low-level" to "high- level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones.

Low-level languages. Machine languages and the assembly languages that represent them (collectively termed low-level programming languages) tend to be unique to a particular type of computer. For instance, an ARM architecture computer (such as may be found in a PDA or a hand-held videogame) cannot understand the machine language of an Intel Pentium.

The only language a PC can directly execute is machine code, which consists of 1s and 0s. This language is difficult to write, so we use symbolic languages that are easier to understand. For example, assembly languages use abbreviations such as ADD, SUB, and MPY to represent instructions. The program is then translated into machine code by software called an assembler. Machine code and assembly languages are called low-level languages because they are closer to the hardware.

Higher-level languages. Though considerably easier than in machine language, writing long programs in assembly language is often difficult and is also error prone. Therefore, most practical programs are written in more abstract high-level programming languages that are able to express the needs of the programmer more conveniently (and thereby help reduce programmer error). High level languages are usually "compiled" into machine language (or sometimes into assembly language and then into machine language) using another computer program called a compiler. High level languages are less related to the workings of the target computer than assembly language, and more related to the language and structure of the problem(s) to be solved by the final program. It is therefore often possible to use different

compilers to translate the same high level language program into the machine language of many different types of computer. This is part of the means by which software like video games may be made available for different computer architectures such as personal computers and various video game consoles.

High-level languages, however, are closer to human languages; they use forms resembling English, which makes programming easier. The program is translated into machine code by software called a compiler. Some examples are:

FORTRAN – used for scientific and mathematical applications;

COBOL – popular for business applications;

BASIC – used as a teaching language; Visual BASIC is now used to create Windows applications;

C – used to write system software, graphics and commercial programs;

Java – designed to run on the Web; Java applets are small programs that run automatically on web pages and let you watch animated characters, and play music and games.

The languages used to create Web documents are called mark-up languages; they use instructions (mark-ups) to format and link text files. Examples are:

HTML – the code used to create Web pages

VoiceXML – it makes Internet content accessible via speech recognition and phone. Instead of using a web browser on a PC, you use a telephone to access voice-equipped web sites. You just dial the phone number of the web site and then give spoken instructions, commands, and get the required information.

More advanced techniques, for instance using Objects, Generics, Multi-threading mean that the modern programming languages are far more powerful.

As computers get faster, have more RAM, applications will get more complex and it is likely that more development will shift from C++ to the higher level languages such as Java and C#. Microsoft has put a lot of faith in C# as its answer to Java and has the financial leverage to continue plugging it for a very long time. We expect both Java and C# to become the two dominant programming languages.

Text 3. Drupal as a CMS

Ever since its advent in the web arena, Drupal CMS has evolved constantly to emerge as one of the most complex content management system. Its acceptability and popularity can be mapped by the fact that several top notch sites are using the CMS as their support system- The Economist, The White House, MIT, Harvard, Popular Science and Sony Music are to name a few. As a matter of fact, today 538,813 people are using Drupal in 228 countries across the world in 182 available languages.

For those who are new to Drupal but have been using Wordpress for a while will find some striking similarities between the two content management systems. Though, because of its interactive and sophisticated programming interface, Drupal is considered more than a simple CMS tool – it is viewed as an end-to-end development solution. The system in Drupal showcases advanced admin options that are supported by in-depth report generating tools. The basic use of the platform doesn't require any technical whereabouts, but it is put more into use as you hire Drupal developers and advanced level enterprises.

While Drupal has emerged as one of the most user-friendly CMS platforms, it can be too much to handle for some users. This post explores the various advantages and disadvantages that Drupal poses as a content management system.

Advantages:

✓ Drupal incorporates all basic features that you may need to manage content effectively. The Core version of Drupal includes basic utility features such as user registration, page layout customization, menu management and system administration.

✓ Drupal offers you great modules that can be used as the building blocks of your web site, enabling you create and manage content and rules quickly. Using the in-built themes and templates offered by Drupal, you need not start building a site from scratch, irrespective of how intricate your development venture may be.

✓ Drupal facilitates easy organization of content. It lets you create custom lists, organize content through path URLs, create defaults and associate content. This makes organizing, searching, resusing and managing content simplified.

✓ The 7000+ plugins of Drupal allow you to add new capabilities to your development forte. The plugins are free to download and use, which means you, can use as many plugins as you feel like to enhance the functionalities of your Drupal portal.

Disadvantages:

× Drupal might offer great power and functionalities but it is not as user-friendly as other CMS available in the web arena, because of its high learning curve. CMS like Wordpress and Joomla may offer better usability but they lack the power of Drupal.

× In terms of performance, Drupal lags behind Wordpress in two aspects- scalability and loading. The expanded breadth of Drupal functionalities is to be blamed for the slower speed of the platform. Moreover, the significantly large learning curve is quite time consuming at times.

× The backward compatibility feature is lacking in the Drupal CMS. This means, if you are used to some other content system or program, then Drupal might not be the right CMS choice.

× Considering both the advantages and disadvantages of the Drupal CMS, it can be said that the CMS is a perfect choice for those looking for powerful features, to enhance the functioning and usability of your next content-driven web site.

Text 4. Program Planning

The programming process begins with a problem statement that helps you clearly define the purpose of a computer program. In the context of programming, a problem statement defines certain elements that must be manipulated to achieve a result or goal. A good problem statement for a computer program has three characteristics:

1. It specifies any assumptions that define the scope of the problem.
2. It clearly specifies the known information.
3. It specifies when the problem has been solved.

In a problem statement an assumption is something you accept as true in order to proceed with program planning. The "known information" is the information that you supply to the computer to help it solve a problem. There are also variables (values that can change) and constants (factors that remain the same) in computer programs.

Formulating a problem statement provides a minimal amount of planning, which is sufficient for only the simplest programs. A typical commercial application requires far more extensive planning, which includes detailed program outlines, job assignments, and schedules. To some extent, program planning depends on the language and paradigm used to code a computer program. The phrase programming paradigm refers to a way of conceptualizing and structuring the tasks a

computer performs. For example, whereas one programmer might focus on the steps required to complete a specific computation, another one might focus on the data that forms the basis for the computation. Quite a number of programming paradigms exist, and a programmer might use techniques from multiple paradigms while planning and coding a program.

There are different program planning tools, such as flowcharts, structured English, pseudo code, UML diagrams, and decision tables, which are used to provide sufficient planning.

Regardless of the tools used, when planning is complete, programmers can begin coding, testing, and documenting. The process of coding a computer program depends on programming language you use, the programming tools you select, and the programming paradigm that best fits the problem you are trying to solve. Programmers typically use a text editor, a program editor, or a VDE to code computer programs.

A text editor is any word processor that can be used for basic editing tasks, such as writing e-mail, creating documents, or coding computer programs. When using a text editor to code a computer program, you simply type in each instruction.

A program editor is a type of text editor specially designed for entering code for computer programs.

A VDE (visual development environment) provides programmers with tools to build substantial sections of a program by pointing and clicking rather than typing lines of code. A typical VDE is based on a form design grid that a programmer manipulates to design the user interface for a program. By using various tools provided by the VDE, a programmer can add objects, such as controls and graphics, to the form design grid. In the context of a VDE, a control is a screen-based object whose behaviour can be defined by a programmer.

In visual development environment, each control comes with predefined set of events. Within the context of programming, an event is defined as an action, such as click, drag, or key press, associated with the form or control. A programmer can select the events that apply to each control. An event usually requires the computer to make some response. Programmers write event-handling code for the procedures that specify how the computer responds to each event.

A programmer's choice of development tools depends on what is available for a particular programming language and the nature of the programming project. Text editors and program editors provide a fine tool set for programs with minimal user interfaces. A visual development environment is a powerful tool for programming software applications for GUI environments, such as Windows. Most GUI applications are "event-driven", which means that when launched, the program's interface appears on the screen and waits for the user to initiate an event.

A computer program must be tested to ensure that it works correctly. Testing often consists of running the program and entering test data to see whether the program produces correct results.

When a program doesn't work correctly, it is usually the result of an error made by the programmer. A syntax error occurs when an instruction doesn't follow the syntax rules, or grammar of the programming language. Syntax errors are easy to make, but they are usually also easy to detect and correct.

Another type of program bug is a runtime error, which, as its name indicates, shows up when you run a program. Some runtime errors result from instructions that the computer can't execute.

Some runtime errors are classified as logic errors. A logic error is an error in the logic or design of a program. It can be caused by an inadequate definition of the problem or an incorrect formula for a calculation, and they are usually more difficult to identify than syntax errors.

Programmers can locate errors in a program by reading through lines of code, much like a proof-reader. They can also use a tool called debugger to step through a program and monitor the status of variables, input, and output. A debugger is sometimes packaged with a programming language or can be obtained as an add-on.

Anyone who uses computers is familiar with program documentation in the form of user manuals and help files. Programmers also insert documentation called remarks or "comments" into the programming code. Remarks are identified by language-specific symbols.

A well-documented program contains initial remarks that explain its purpose and additional remarks in any sections of a program where the purpose of the code is not immediately clear.

Text 5. Procedural Programming

The traditional approach to programming uses a procedural paradigm (sometimes called "imperative paradigm") to conceptualize the solution to a problem as a sequence of steps. A program written in a procedural language typically consists of self-contained instructions in a sequence that indicates how a task is to be performed or a problem is to be solved.

A programming language that supports the procedural paradigm is called a procedural language. Procedural languages are well suited for problems that can be easily solved with a linear, or step-by-step, algorithm. Programs created with procedural languages have a starting point and an ending point. The flow of execution from the beginning to the end of the program is essentially linear – that is, the computer begins at the first instruction and carries out the prescribed series of instructions until it reaches the end of the program.

An algorithm is a set of steps for carrying out a task that can be written down and implemented. An algorithm for a computer program is a set of steps that explains how to begin with known information specified in a problem statement and how to manipulate that information to arrive a solution. In a later phase of the software development process, the algorithm is coded into instructions written in a programming language so that a computer can implement it.

To design an algorithm, you might begin by recording the steps you take to solve the problem manually. The computer also needs the initial information, so the part of your algorithm must specify how the computer gets it. Next, your algorithm should also specify how to manipulate this information and, finally, how the computer decides what to display as the solution.

You can express an algorithm in several different ways, including structured English, pseudo code, and flowcharts. These tools are not programming languages, and they cannot be processed by a computer. Their purpose is to give you a way to document your ideas for program design.

Structured English is a subset of the English language with a limited selection of sentence structures that reflects processing activities. Another way to express an algorithm is with pseudo code. Pseudo code is a notational system for algorithms that has been described as a mixture of English and your favourite programming language.

A third way to express an algorithm is to use a flowchart. A flowchart is a graphical representation of the way a computer should progress from one instruction to the next when it performs a task.

Before finalizing the algorithm for a computer program, you should perform a walkthrough to verify that your algorithm works. To perform a walkthrough for a simple program, you can use a calculator, paper, and pencil to step through a sample problem using realistic "test" data.
For more complex programs, a walkthrough might consist of a verbal presentation to a group of programmers who can help identify logical errors in the algorithm and suggest ways to make the algorithm more efficient.

The algorithm specifies the order in which program instructions are performed by the computer. Unless you do otherwise, sequential execution is the normal pattern of program execution. During sequential execution, the computer performs each instruction in the order it appears – the first instruction in the program is executed first, then the second instruction, and so on, to the last instruction in the program.

Some algorithms specify that a program must execute instructions in an order different from the sequence in which they are listed, skip some instructions under certain circumstances, or repeat instructions. Control structures are instructions that specify the sequence in which program is executed. Most programming languages have three types of control structures: sequence controls, selection controls, and repetition controls.

A sequence control structure changes the order in which instructions are carried out by directing the computer to execute an instruction elsewhere in the program. A sequence control structure directs the computer to the statements they contain, but when these statements have been executed, the computer neatly returns to the main program.

A selection control structure, also referred to as a "decision structure" or "branch", tells a computer what to do, based on whether a condition is true or false. A simple example of a selection control structure is the IF...THEN...ELSE command.

A repetition control structure directs the computer to repeat one or more instructions until certain condition is met. The section of code that repeats is usually referred to as a loop or "iteration". Some of the most frequently used repetition commands are FOR...NEXT, DO...WHILE, DO...UNTIL, and WHILE...WEND (which means "while ends").

All the first programming languages were procedural. The first widely used standardized computer language, FORTRAN, with its procedural paradigm set the pattern for other popular procedural languages, such as COBOL, APL, ALGOL, PL/1, PASCAL, C, ADA, and BASIC.

The procedural approach is best suited for problems that can be solved by following a step-by-step algorithm. It has been widely used for transaction processing, which is characterized by the use of a single algorithm applied to many different sets of data. For example, in banking industry, the algorithm for calculating checking account balances is the same, regardless of the amounts deposited and withdrawn. Many problems in math and science also lend themselves to the procedural approach.

The procedural approach and procedural languages tend to produce programs that run quickly and use system resources efficiently. It is a classic approach understood by many programmers, software engineers, and system analysts. The procedural paradigm is quite flexible and powerful, which allows programmers to apply it to many types of problems.

The downside of the procedural paradigm is that it does not fit gracefully with certain types of problems – those that are unstructured or those with very complex algorithms. The procedural paradigm has also been criticized because it forces programmers to view problems as a series of steps, whereas some problems might better be visualized as interacting objects or as interrelated words, concepts, and ideas.

Text 6. Object-oriented Programming

One of the principal motivations for using OOP is to handle multimedia applications in which such diverse data types as sound and video can be packaged together into executable modules. Another is writing program code that's more intuitive and reusable; in other words, code that shortens program-development time.

Perhaps the key feature of OOP is encapsulation – bundling data and program instructions into modules called 'objects'. Here's an example of how objects work. An icon on a display screen might be called 'Triangles'. When the user selects the Triangles Icon – which is an object composed of the properties of triangles and other data and Instructions – a menu might appear on the screen offering several choices. The choices may be (1) create a new triangle and (2) fetch a triangle already in storage. The menu, too, is an object, as are the choices on it. Each time a user selects an object, instructions inside the object are executed with whatever properties or data the object holds, to get to the next step. For instance, when the user wants to create a triangle, the application might execute a set of instructions that displays several types of triangles – right, equilateral, isosceles, and so on.

Many industry observers feel that the encapsulation feature of OOP is the natural tool for complex applications in which speech and moving images are integrated with text and graphics. With moving images and voice built into the objects themselves, program developers avoid the sticky problem of deciding how each separate type of data is to be integrated and synchronized into a working whole.

A second key feature of OOP is inheritance. This allows OOP developers to define one class of objects, say "Rectangles", and a specific instance of this class, say "Squares" (a rectangle with equal sides). Thus, all properties of rectangles – "Has 4 sides" and "Contains 4 right angles" are the two shown here – are automatically inherited by Squares. Inheritance is a useful property in rapidly processing business data. For instance, consider a business that has a class called "Employees at the Dearborn Plant" and a specific instance of this class, "Welders". If employees at the Dearborn plant are eligible for a specific benefits package, welders automatically qualify for the package. If a welder named John Smith is later relocated from Dearborn to Birmingham, Alabama, where a different benefits package is available, revision is simple. An icon representing John Smith – such as John Smith's face – can be selected on the screen and dragged with a mouse to an icon representing the Birmingham plant. He then automatically "inherits" the Birmingham benefit package.

A third principle behind OOP is polymorphism. This means that different objects can receive the same instructions but deal with them in different ways. For instance, consider again the triangles example. If the user right clicks the mouse on "Right triangle", a voice clip might explain the properties of right triangles. However, if the mouse is right clicked on "Equilateral triangle" the voice instead explains properties of equilateral triangles.

The combination of encapsulation, inheritance and polymorphism leads to code reusability. "Reusable code" means that new programs can easily be copied and pasted together from old programs. All one has to do is access a library of objects and stitch them into a working whole. This eliminates the need to write code from scratch and then debug it. Code reusability makes both program development and program maintenance faster.

Text 7. Object-Oriented Languages and Applications

Computer historians believe that SIMULA (SIMUlation LAnguage) was the first computer language to work with objects, classes, inheritance, and methods. SIMULA was developed in 1962 by two Norwegian computer scientists for the purpose of programming simulations and models. SIMULA laid the foundation for the object-oriented paradigm, which was later incorporated into other programming languages, such as Eiffel, Smalltalk, C++, and Java.

The second major development in object-oriented languages came in 1972 when Alan Kaye began work on the Dynabook project at the Xerox Palo Alto Research Center (PARC). Dynabook was a prototype for a notebook-sized personal computer, intended to handle all the information needs of adults and children. Kaye developed a programming language called Smalltalk for the Dynabook that could be easily used to create programs based on real-world objects. Dynabook never became a commercial product, but Smalltalk survived and is still in use today. Smalltalk is regarded as a classic object-oriented language, which encourages programmers to take a "pure" OO approach to the programming process.

As the object-oriented paradigm gained popularity, several existing programming languages were modified to allow programmers to work with objects, classes, inheritance, and polymorphism. The concept for the Ada programming language originated in 1978 at the U. S. Department of Defense. The first versions of Ada were procedural, but in 1995, the language was modified to incorporate object-oriented features. A similar transformation took place with the C language in 1983, except that the object-oriented version earned a new name — C++. Hybrid languages, such as Ada95, C++, Visual Basic, and C#, give programmers the option of using procedural and object-oriented techniques.

Java is one of the newest additions to the collection of object-oriented languages. Originally planned as a programming language for consumer electronics, such as interactive cable television boxes, Java evolved into an object-oriented programming platform for developing Web applications. Java was officially launched by Sun Microsystems in 1995 and has many of the characteristics of C++, from which it derives much of its syntax. Like C++, Java can also be used for procedural programming, so it is sometimes classified as a hybrid language.

The object-oriented paradigm can be applied to a wide range of programming problems. Basically, if you can envision a problem as a set of objects that pass messages back and forth, the problem is suitable for the OO approach.

The object-oriented paradigm is cognitively similar to the way human beings perceive the real world. Using the object-oriented approach, programmers might be able to visualize the solutions to problems more easily. Facets of the object-oriented paradigm can also increase a programmer's efficiency because encapsulation allows objects to be adapted and reused in a variety of different programs. Encapsulation refers to the process of hiding the internal details of objects and their methods. After an object is coded, it becomes a "black box," which essentially hides its details from other objects and allows the data to be accessed using methods.

A potential disadvantage of object-oriented programs is runtime efficiency. Object-oriented programs tend to require more memory and processing resources than procedural programs. Programmers, software engineers, and system analysts can work together to weigh the trade-offs between the OO approach and runtime efficiency.

Text 8. Java

Java is a high-level language with which to write programs that can execute on a variety of platforms. So are C, C++, Fortran and Cobol, among many others. So the concept of a portable execution vehicle is not new. Why, then, has the emergence of Java been trumpeted so widely in the technical and popular press?

Why is Java different from other languages? Part of Java's novelty arises from its new approach to portability. In previous High- level languages, the portable element was the source program. Once the source program is compiled into executable form for a specific instruction

set architecture (ISA) and bound to a library of hardware-dependent I/O, timing and related operating system (OS) services, portability is lost. The resultant executable form of the program runs only on platforms having that specific ISA and OS. Thus, if a program is to run on several different platforms, it has to be recompiled and re-linked for each platform. And if a program is sent to a remote target for execution, the sender must know in advance the exact details of the target to be able to send the correct version.

With Java, source statements can be compiled into machine-independent, "virtual instructions" that are interpreted at execution time. Ideally, the same virtual code runs in the same way on any platform for which there is an interpreter and OS that can provide that interpreter with certain multithreading, file, graphical and similar support services. With portability moved to the executable form of the program, the same code can be sent over the net to be run without prior knowledge of the hardware characteristics of the target. Executable programs in the Java world are universal.

In principle, portability could have been achieved in the C or C++ world by sending the source program over the net and then having the compilation and linkage done as a pre-step to execution. However, this approach would require that the target system have sufficient CPU speed and disk capacity to run the sophisticated compilers and linkers required. In the future, network platforms may not have the facilities to run even a simple compiler.

Is that all? Java is not just a new concept in portability. The Java language evolved from C and C++ by locating and eliminating many of the major sources of program error and instability. For example, C has an element known-as a pointer that is supposed to contain the address at which a specific type of information is stored. However, the pointer can be set to literally any address value, and by «casting» a programmer can trick the compiler into storing any type of information at the arbitrary pointer address. This is convenient if you write error-free code and a snake pit if you don't. Java does not have pointers.

Equally important, Java has built-in support for multiprogramming. C and its immediate descendant C++ were designed to express a single thread of computing activity. There was no inherent support for multiple program threads executing simultaneously (on multiple CPUs), or in parallel (timesharing a single CPU). Any such facilities had to be supplied by an external multitasking operating system. There are several good programs of this type readily available, such as MTOS-UX from

Industrial Programming. However, the services provided are all vendor-specific. Neither ANSI nor any of the various committees set up to hammer out a universal set of OS services ever produced a single, universally-accepted standard. There are in fact, several proposed standards, so there is no standard. Java bypasses the problem by building multithreading and the data synchronization it requires directly into the source program. You still need an OS to make this happen, but, the semantic meaning of the OS actions is standardized at the source level.

A standard at last. Java has all of the technical requisites to become the standard programming language for programs to be distributed over the net. And with a well-supported campaign spearheaded by Sun Microsystems, Java is becoming the de facto working standard. Will Java supersede C as the language of choice for new programs in general? With network programming likely to play an increasingly larger part in the overall programming field, I think so.

Java for embedded systems. Embedded or real-time systems include all those in which timing constrains imposed by the world outside of the computer play a critical role in the design

and implementation of the system. Common areas for embedded systems are machine and process control, medical instruments, telephony, and data acquisition. A primary source of input for embedded systems – are random, short-lived, external signals. When such signals arrive, the processor must interrupt whatever else it is doing to capture the data or it will be lost. Thus, an embedded program is most often organized as a set of individual, but cooperating threads of execution. Some threads capture new data, some analyze the new data and integrate it with past inputs some generate the outgoing signals and displays that are the products of the system. Currently, most embedded programs are coded in C, with critical parts possibly in assembler.

Putting the issue of execution efficiency aside, some of the major problems of C for embedded systems are:

- The permissiveness of C operations, which can lead to undisciplined coding practices and ultimately to unstable execution.
- The absence of universal standards for multithreading, shared data protection, and intra-thread communication and coordination, which can make the program hard to transfer to alternate platforms. But, these are jus t the problems that Java solves. Since many programmers will have to learn Java because of its importance to the net, it will be natural for Java to supplant C in the embedded world. The use of Java may be different, however. We anticipate that Java programs for embedded applications will differ from net applets in at least five major ways.

Embedded applications will be:

- compiled into the native ISA for the target hardware;
- capable of running in the absence of a hard or floppy disk, and a network connection;
- supported by highly tailored, thus relatively small run-time packages;
- able to execute on multiple processors, if needed for capacity expansion;
- contain significant amounts of legacy C code, at least during the transition from C to Java.

Mixed systems: multiple languages, multiple CPUs. While we expect Java to supersede C as the primary programming language for embedded systems in the near future, there is still an enormous number of lines of C code in operation. Companies will have to work with that code for many years as the transition to Java runs its course. Many systems will have to be a mixture of legacy C code and Java enhancements.

It is not trivial to integrate an overall application with some components written in Java and others in C or assembler. Part of the problem arises from security issues: How can Java guarantee the security of the system if execution disappears into "unknown" regions of code? Furthermore, the danger is compounded if the non-Java code were to make OS support service calls, especially calls that alter the application's threading and data-protection aspects. Java expects to be the sole master of such matters.

Thus we see that mixed language systems may have to exist, but this is not going to be easy. Similarly, there may be problems with multiple CPUs. Current CPUs are fast, and get faster with each new generation. Yet, there are some embedded applications for which a single CPU still does not have enough power to keep up with a worst-case burst of external input. Such systems require multiple CPUs working together to complete the required processing. Even if the system can handle current work loads, the next version may not.

Do we have a problem? When you combine the des ire to write in Java, with the need to execute on unique, system-specific hardware, possibly with mixed source languages and multiple CPUs, you introduce a major obstacle. You are not likely to get an off-the-shelf Java OS from Sun Microsystems.

Many companies that have previously offered their own proprietary real-time OS are now developing a Java OS, or are seriously considering such an offering. My own company, Industrial Programming, is currently using its experience with embedded multithreading / multiprocessor operating systems to create a new system that will handle applications written in both Java and C. And as the case with its traditional product, MTOS-UX, the OS is transparent to the number of tightly-coupled CPUs that are executing the application code. If one CPU is not enough, you can add more without altering the application.

Unit II. Data Security

Text 1. Data Security (Part I)

There are a variety of different crimes that can be committed in computing, including:

Computer Crime	Description
Spreading viruses	Distributing programs that can reproduce themselves and written with the purpose of causing a computer to behave in an usual way.
Hacking	Gaining unauthorised access to network system.
Salami shaving	Manipulating programs or data so that small amounts of money are deducted from a large number of transactions or accounts and accumulated elsewhere. The victims are often unaware of the crime because the amount taken from any individual is so small.
Denial of service attack	Swamping a server with large numbers of requests.
Trojan horse	A technique that involves adding concealed instructions to a computer program so that it will still work but will also perform prohibited duties. In order words, it appears to do something destructive in the background.
Trapdoors	A technique that involves leaving, within a completed program, an illicit program that allows unauthorised and unknown entry.
Mall bombing	Inundating an email address with thousands of messages, slowing or even crashing the server.
Software piracy	Unauthorized copying of a program for sale or distributing to other users.
Piggybacking	Using another person's identification code or using that person's files before he or she has logged off (disconnected from a network account).
Spoofing	Tricking a user into revealing confidential information such as an access code or a credit-card number.
Defacing	Changing the information shown on another person's website.
Hijacking	Redirecting anyone trying to visit a certain site elsewhere.

A computer virus is a program that can reproduce itself and is written with the purpose of causing damage or causing a computer to behave in an unusual way. It infects other programs i.e. it attaches itself to other programs known as host programs and therefore reproduces itself. It operates by replacing the first instruction in the host program with a JUMP command. This is a command that changes the normal instruction sequence in a program causing the virus instructions to be executed (processed by the processor) before the host program instructions. When the virus has been executed, the host program is executed in the normal way.

When it attaches to operating systems programs to integrate itself with the operating system (the set of programs that control the basic functions of a computer and provide communication between the applications programs and the hardware), it is said to have patched the operating

system. Viruses normally attach themselves to programs that have a COM extension (e.g. command.com) that are known as command files or COM files, or to programs that have an EXE extension (e.g. explorer.exe) that are known as executable files or EXE files. A virus is loaded into memory (copied from the storage media into memory) when a program that it has attached itself to is run or executed (processed by the processor). It then becomes memory resident i.e. it stays in the memory until the computer is switched off. When a virus is triggered by a predetermined event, it operates the payload (the part of the virus that causes the damage). Although a virus is the term used to describe any program that can reproduce itself, viruses usually have four main parts:

1. *misdirection routine that it enables it to hide itself;*
2. reproduction routine that allows it to copy itself to other programs;
3. trigger that causes the payload to be activated at a particular time or when a particular event takes place;
4. payload that may be a harmless joke or may be very destructive.

A program that has a payload but does not have a reproduction routine is known as a Trojan.

Each virus is given a name and can be classified as a particular type of virus. Virus types include: logic bombs that destroy data when triggered; boot sector viruses that stores themselves in the boot sector of a disk (the part of a disk containing the programs used to start up the computer).

File viruses that attach themselves to COM files. Macro viruses that are small macro programs that attach themselves to word processor files and use the macro programming facilities provided in some word processing programs.

Text 2. Data Security (Part II)

There are a variety of security measures that can be used to protect hardware (the physical components of a computer system) including:

1. Controlling physical access to hardware and software.
2. Backing up data and programs (storing a copy of files on a storage device to keep them safe).
3. Implementing network controls such as:
   - ✓ using passwords (a secret code used to control access to a network system);
   - ✓ installing a firewall (a combination of hardware and software used to control the data going into and out of a network. It is used to prevent unauthorised access to the network by hackers);
   - ✓ encrypting data (protecting data by putting it in a form only authorised users can understand;
   - ✓ installing a call-back system (a system that automatically disconnects a telephone line after receiving a call and then dials the telephone number of the system that made the call, to reconnect the line. It is used in remote access system to make sure that connections can only be made from permitted telephone numbers);
   - ✓ using signature verification or biometric security devices (security devices that measure some aspect of a living being e. g. a fingerprint reader or an eye scanner).
4. Separating and rotating the computing functions carried out by employees and carrying out periodic audits of the system i.e. observing and recording events on the network systematically.

5. Protecting against natural disasters by installing uninterruptible power supplies (battery backup system that automatically provide power to a computer when the normal electricity source fails) and surge protectors (electronic devices that protect equipment from damage due to a sudden surge in a power supply).
6. Protecting against viruses by using antivirus programs (computer programs or sets of programs used to detect, identify and remove viruses from a computer system) and ensuring that all software is free of viruses before it is installed. Particular care must be taken when using public domain software (free software) and shareware (software that is free to try out but must be paid for if it is used after the trial period).

A smart card is a plastic card containing a processor and memory chip. It can be used to store large amounts of confidential data including coded data that can be used as digital cash (electronic currency that is used for making electronic purchases over the Internet). It can also be used as a security device to prevent or allow access to a system and allow a user to withdraw cash from a bank ATM (automatic teller machine – a type of machine used by banks for enabling customers to withdraw money from their bank accounts). A smart card reader is a device used for reading smart cards by detecting radio signals emitted from a radio antenna (aerial) in the form of a small coil inside the smart card.

An anti-virus program is a program that checks files for virus coding instructions inside another program and can be used for removing any virus coding instructions detected.

A backup program is a program that stores a copy of data on a storage device to keep it safe. There are different kinds of backup, including:
1. Incremental backup which copies all the selected files that have created or changed since the last full, differential or incremental backup. These files are identified by the fact that their archive bit would be on. The archive bit is a digital bit stored with a file indicating if the has been backed up since it was last edited. The archive bit is switched off when the file is backed up using a full or incremental backup.
2. Differential backup which copies all the files created or modified since the last full backup. The archive bit is not set to "off" by a differential backup.
3. Full backup which copies all the selected files on a system, whether or not they have been edited or backed up before.

A series of incremental backups and a full backup, or the most recent differential backup and a full backup, is known as a backup set.

Text 3. The ex-hacker

A hacker is a person who attempts to gain unauthorized access to a network system. They are often young teenagers although they you usually fairly skilled programmers (people who write computer programs). Sometimes, the type of person who becomes a hacker is referred to as a 'geek' (an expert lacking in social skill), or as an 'anorak' (a slang term for an eccentric, socially inept person with little or no fashion sense and having an obsessive interest in a hobby or subject). Although 'geek' was originally derogatory term it is now used in computing to mean a dedicated expert. Although it is illegal, people become hackers for different reasons including: making money, criminal purposes, or to expose political information. But often people hack (break into a computer system) just because it is an exciting challenge. Parents are often unaware that their children are hacking into computer system although they usually receive very large telephone bills.

Since hacking (attempting to gain unauthorized access to a network system) is illegal, hackers want to keep their true identity secret but they often like to call themselves by special names such as "the Analyser". The Internet has made hacking more common and hackers are found throughout the world. They sometimes form the hacking groups or teams that work together and exchange ideas. These groups also like to be known by names such as "Hackers Unite".

Hackers like to attack and penetrate computer systems belonging to large, important organizations such as the Pentagon's computer systems, computer systems belonging to US military bases and Hotmail, the free email service provided by the Microsoft Corporation. In fact, hackers compete with each other to be the first to hack into really powerful systems. Often, breaking into a system is done gradually, with the hacking gaining entry to a system then planting passwords in the system, allowing them to gain access to the system more easily in the future.

When a hacker gains access to a system they don't usually break into the system using the Internet and steal all the data on the system, as is often portrayed in the cinema. In fact, most hacks (break-ins) are done by company staff misusing the company network system. Hackers have been known to do a variety of things to computer systems, including:

1. Downloading files (copying files from a server computer) and leaking confidential information. Posting information is the term used for making information available to a large number of users in a newsgroup (an Internet discussion group that uses a restricted area on a server computer to display messages about a common interest) or on a bulletin board (an electronic notice board system that enables users to display messages for other users to read).
2. Exposing email (electronic mail) correspondence managed by well known email services, causing the service to be shut down while the exposed weakness in the system is repaired.
3. Programming email server computers to reroute email (send to a different email address than the one it was originally sent to).
4. Hijacking websites by redirecting the Web address (URL) to point to another website.
5. Defacing websites by changing the text and graphics on the web pages, sometimes leaving very rude messages on the system.
6. Blackmailing the owners of websites by threatening to damage their systems by doing something like releasing a virus onto their system, although such a threat often turns out to be nothing more than a hoax.

Sometimes, young hackers put their experience and knowledge to good use when they become older. Many former hackers have been hired by large companies as security experts. They are employed to test out the company systems by trying to hack into them to find any weaknesses in the systems. Cyberspace is the combination of all the data on all the computer networks throughout the world, accessed using the Internet. A person who uses their skills to make cyberspace safer is referred to as a "white hat" hacker.

A computer system can be hacked (broken into) in various ways including:

- Guessing somebody's password
- Finding a bug (a fault in a system) that allows certain passwords to access information they are not supposed to access.
- Phoning a company, pretending to be a company employee and asking for a password. People tend to be too trusting.

Connecting to a computer network involves logging in (sometimes referred to as logging on) by typing a username or ID (identification username) and a password. Usernames that are often used on networks systems include "guest", "demo" and "help".

To avoid a computer system being hacked into, the people managing the system must work hard to keep ahead of the hackers. There are different ways of avoiding being hacked into including:

- ✓ Installing a firewall (a combination of hardware and software used to control the data going into and out of a network).
- ✓ Using a call-back system (a system that automatically disconnects a telephone line after receiving a call and then dials the telephone number of the system that made the call, to reconnect the line. It is used in remote access systems to make sure that connections can only be made from permitted telephone numbers).
- ✓ Having really secure passwords – don't use common names or dictionary words.
- ✓ Auditing the system regularly (checking the system regularly using event logs to find failed access attempts).

Some people do not like to give out their credit card numbers on the Internet. Hackers have been known to get databases (applications programs used for storing information so that it can be easily searched and sorted) of credit card numbers by hacking computer systems. However, in the opinion of the ex-hacker in this unit, using your credit card on the Internet is no more dangerous than giving your credit card number on the telephone or throwing away a credit card receipt. There are various things you can do to avoid credit card theft on the Internet including:

- ✓ Using a separate credit card for Internet purchases;
- ✓ Having a small credit limit on the credit card you use;
- ✓ Buying a pre-paid charge card for small purchases.

In the future, smart cards (plastic cards containing a processor and memory chip that can be used to store large amounts of confidential data) will be used instead of credit cards. This will require smart card readers (devices used for reading smart cards) to be attached to computers.

Text 4. Data Theft: How Big Is a Problem?

Data theft is, quite simply, the unauthorized copying or removal of confidential information from a business or other large enterprise. It can take the form of ID-related theft or the theft of a company's proprietary information or intellectual property. ID-related data theft occurs when customer records are stolen or illegally copied. The information stolen typically includes customers' names, addresses, phone numbers, usernames, passwords and PINs, account and credit card numbers, and, in some instances, Social Security numbers. When transmitted or sold to lower-level criminals, this information can be used to commit all manner of identity fraud. A single data theft can affect large numbers of individual victims.

Non-ID data theft occurs when an employee makes one or more copies of a company's confidential information, and then uses that information either for his own personal use or transmits that information to a competitor for the competitor's use. However it's done, this is a theft of the business' intellectual property, every bit as harmful as a theft of money or equipment. A company's confidential information includes its employee records, contracts with other firms, financial reports, marketing plans, new product specifications, and so on. Imagine you're

a competitor who gets hold of a company's plans for an upcoming product launch; with knowledge beforehand you can create your own counter-launch to blunt the impact of the other company's new product. A little inside information can be extremely valuable and damaging for the company from which it was stolen.

Data theft can be a virtual theft (hacking into a company's systems and transmitting stolen data over the Internet) or, more often, a physical theft (stealing the data tapes or discs). In many ways, it's easier for a thief to physically steal a company's data than it is to hack into the company's network for the same purpose. Most companies give a lot of attention to Internet-based security, but less attention is typically paid to the individuals who have physical access to the same information.

One would expect data theft to be somewhat widespread. And it probably is if we truly knew all the numbers. The problem with trying to size the data theft issue is twofold. First, many companies do not report data theft to the police or do not publicize such thefts; they're trying to avoid bad publicity. And even when data theft is reported, the dollar impact of such theft is difficult to ascertain.

Whichever number is correct, that's a lot of stolen data. Add to that the immeasurable cost of intellectual property data theft, and you get a sense of the size of the problem — it's big and it's getting bigger.

Unfortunately, there's little you as an individual can do to prevent data theft; the onus is all on the company holding the data. You could reduce your risk by limiting the number of companies with which you do business, but that may not be practical. Being alert is your only defence against this type of large-scale theft.

Text 5. What is Malicious Code?

Malicious code is any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system. Though the problem of malicious code has a long history, a number of recent, widely publicized attacks and certain economic trends suggest that malicious code is rapidly becoming a critical problem for industry, government, and individuals.

Traditional examples of malicious code include viruses, worms, Trojan Horses, and attack scripts, while more modern examples include Java attack applets and dangerous ActiveX controls.

Viruses are pieces of malicious code that attach to host programs and propagate when an infected program is executed.

Worms are particular to networked computers. Instead of attaching themselves to a host program, worms carry out programmed attacks to jump from machine to machine across the network.

Trojan Horses, like viruses, hide malicious intent inside a host program that appears to do something useful (e. g., a program that captures passwords by masquerading as the login daemon.)

Attack scripts are programs written by experts that exploit security weaknesses, usually across the network, to carry out an attack. Attack scripts exploiting buffer overflows by "smashing the stack" are the most commonly encountered variety.

Java attack applets are programs embedded in Web pages that achieve foothold through a Web browser.

Dangerous ActiveX controls are program components that allow a malicious code fragment to control applications or the operating system.

Recently, the distinctions between malicious code categories have been bleeding together, and so classification has become difficult.

Any computing system is susceptible to malicious code.

The growing connectivity of computers through the Internet has increased both the number of attack vectors, and the ease with which an attack can be made. More and more computers ranging from home PCs to systems that control critical infrastructures (e.g., the power grid) are being connected to the Internet. Furthermore, people, businesses, and governments are increasingly dependent upon network-enabled communication such as e-mail or Web pages provided by information systems. Unfortunately, as these systems are connected to the Internet, they become vulnerable to attacks from distant sources. Put simply, it is no longer the case that an attacker needs physical access to a system to install or propagate malicious code.

A second trend that has enabled widespread propagation of malicious code is the size and complexity of modern information systems. Complex devices, by their very nature, introduce the risk that malicious functionality may be added (either during creation or afterwards) that extends the original device past its primary intended design. An unfortunate side effect of inherent complexity is that it allows malicious subsystems to remain invisible to unsuspecting users until it is too late.

A third trend enabling malicious code is the degree to which systems have become extensible. From an economic standpoint, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. Unfortunately, the very nature of extensible systems makes it hard to prevent malicious code from slipping in as an unwanted extension.

Text 6. Defence against Malicious Code

Creating malicious code is not hard. In fact, it is as simple as writing a program or downloading and configuring a set of easily customized components. It is becoming increasingly easy to hide ill-intentioned code inside otherwise innocuous objects, including Web pages and e-mail messages. This makes detecting and stopping malicious code before it can do any damage extremely hard.

To make matters worse, our traditional tools for ensuring the security and integrity of hosts have not kept pace with the ever-changing suite of applications. For example, traditional security mechanisms for access control reside within an operating system kernel and protect relatively primitive objects (e.g. files); but increasingly, attacks such as the Melissa virus happen at the application level where the kernel has no opportunity to intervene.

In general, when a computational agent arrives at a host, there are four approaches that the host can take to protect itself.

1. Analyze the code and reject it if there is the potential that executing it will cause harm.
2. Rewrite the code before executing it so that it can do no harm.
3. Monitor the code while its executing and stop it before it does harm, or
4. Audit the code during executing and take policing action if it did some harm.

Analysis includes simple techniques, such as scanning a file and rejecting it if it contains any known virus, as well as more sophisticated techniques from compilers, such as dataflow analysis, that can determine previously unseen malicious code. Analysis can also be used to find bugs (e.g. potential buffer overruns) that malicious code can use to gain a foothold in a system. However, static analysis is necessarily limited, because determining if code will mis-

behave is as hard as the halting problem. Consequently, any analysis will either be too conservative (and reject some perfectly good code) or too permissive (and let some bad code in) or more likely, both. Furthermore, software engineers working on their own systems often neglect to apply any bug-finding analyses.

Code rewriting is a less pervasive approach to the problem, but may become more important. With this approach, a rewriting tool inserts extra code to perform dynamic checks that ensure bad things cannot happen.

Monitoring programs, using a reference monitor, is the traditional approach used to ensure programs don't do anything bad. For instance, an operating system uses the page-translation hardware to monitor the set of addresses that an application attempts to read, write, or execute.

If the application attempts to access memory outside of its address space, then the kernel takes action (e.g. by signalling a segmentation fault). If malicious code does damage, recovery is only possible if the damage can be properly assessed and addressed. Creating an audit trail that captures program behaviour is an essential step. Several program auditing tools are commercially available.

Each of the basic approaches, analysis, rewriting, monitoring, and auditing, has its strengths and weaknesses, but fortunately, these approaches are not mutually exclusive and may be used in concert.

Text 7. Authentication, Authorization, and Accounting

Whether a security system serves the purposes of information asset protection or provides for general security outside the scope of IT, it is common to have three main security processes working together to provide access to assets in a controlled manner. These processes are: authentication, authorization, and accounting.

Identification and Authentication. The process of authentication is often considered to consist of two distinct phases: (1) identification and (2) (actual) authentication. Identification provides user identity to the security system. This identity is typically provided in the form of a user ID. The security system will typically search through all the abstract objects that it knows about and find the specific one for the privileges of which the actual user is currently applying. Once this is complete, the user has been identified. Authentication is the process of validating user identity. The fact that the user claims to be represented by a specific abstract object (identified by its user ID) does not necessarily mean that this is true. To ascertain that an actual user can be mapped to a specific abstract user object in the system, and therefore be granted user rights and permissions specific to the abstract user object, the user must provide evidence to prove his identity to the system. Authentication is the process of ascertaining claimed user identity by verifying user-provided evidence.

The evidence provided by a user in the process of user authentication is called a credential. Different systems may require different types of credentials to ascertain user identity, and may even require more than one credential. In computer systems, the credential very often takes the form of a user password, which is a secret known only to the individual and the system. Credentials may take other forms, however, including PIN numbers, certificates, tickets, etc.

User identification and authentication are typically the responsibility of the operating system. Before being allowed to create even a single process on a computer, the individual must authenticate to the operating system. Applications and services may or may not honour authentication provided by the operating system, and may or may not require additional authentication upon access to them.

There are typically three components involved in the process of user authentication:

Supplicant. The party in the authentication process that will provide its identity, and evidence for it and as a result will be authenticated. This party may also be referred to as the authenticating user, or the client.

Authenticator. The party in the authentication process that is providing resources to the client (the supplicant) and needs to ascertain user identity to authorize and audit user access to resources. The authenticator can also be referred to as the server.

Security authority/database. A storage or mechanism to check user credentials. This can be as simple as a flat file, or a server on the network providing for centralized user authentication, or a set of distributed authentication servers that provide for user authentication within the enterprise or on the Internet.

In a simple scenario, the supplicant, authenticator, and security database may reside on the same computer. It is also possible and somewhat common for network applications to have the supplicant on one computer and the authenticator and security database collocated on another computer. It is also possible to have the three components geographically distributed on multiple computers.

It is important to understand that the three parties can communicate independently with one another. Depending on the authentication mechanism used, some of the communication channels might not be used — at least not by an actual dialogue over the network. The type of communication and whether or not it is used depends on the authentication mechanism and the model of trust that it implements.

Authorization. Authorization is the process of determining whether an already identified and authenticated user is allowed to access information resources in a specific way. Authorization is often the responsibility of the service providing access to a resource.

Before authorization takes place, the user must be identified and authenticated. Authorization relies on identification information to maintain access control lists for each service.

User Logon Process. Authentication and authorization work very closely together, and it is often difficult to distinguish where authentication finishes and where authorization starts. In theory, authentication is only supposed to ascertain the identity of the user. Authorization, on the other hand, is only responsible for determining whether or not the user should be allowed access.

To provide for the logical interdependence between authentication and authorization, operating systems and applications typically implement the so-called user logon process (or login process, also sign-in process). The logon process provides for user identification; it initiates an authentication dialogue between the user and the system, and generates an operating system or application-specific structure for the user, referred to as an access token. This access token is then attached to every process launched by the user, and is used in the process of authorization to determine whether the user has or has not been granted access. The access token structure sits in between user authentication and authorization. The access token contains user authorization information but this information is typically provided as part of the user identification and authentication process.

The logon process can also perform non-security-related tasks. For instance, the process can set up the user work environment by applying specific settings and user preferences at the time of logon.

Accounting. Users are responsible for their actions in a computer system. Users can be authorized to access a resource; and if they access it, the operating system or application

needs to provide an audit trail that gives historical data on when and how a user accessed a resource. On the other hand, if a user tries to access a resource and is not allowed to do so, an audit trail is still required to determine an attempt to violate system authorization and, in some cases, authentication policies.

Accounting is the process of maintaining an audit trail for user actions on the system. Accounting may be useful from a security perspective to determine authorized or unauthorized actions; it may also provide information for successful and unsuccessful authentication to the system.

Accounting should be provided, regardless of whether or not successful authentication or authorization has already taken place. A user may or may not have been able to authenticate to the system, and accounting should provide an audit trail of both successful and unsuccessful attempts.

Furthermore, if a user has managed to authenticate successfully and tries to access a resource, both successful and unsuccessful attempts should be monitored by the system; access attempts and their status should appear in the audit trail files. If authorization to access a resource was successful, the user ID of the user who accessed the resource should be provided in the audit trail to allow system administrators to track access.

Text 8. Understanding Denial of Service

A denial-of-service attack is different in goal, form, and effect than most of the attacks that are launched at networks and computers. Most attackers involved in cybercrime seek to break into a system, extract its secrets, or fool it into providing a service that they should not be allowed to use. Attackers commonly try to steal credit card numbers or proprietary information, gain control of machines to install their software or save their data, deface Web pages, or alter important content on victim machines. Frequently, compromised machines are valued by attackers as resources that can be turned to whatever purpose they currently deem important.

In denial-of-service attacks, breaking into a large number of computers and gaining malicious control of them is just the first step. The attacker then moves on to the denial-of-service attack itself, which has a different goal — to prevent victim machines or networks from offering service to their legitimate users. No data is stolen, nothing is altered on the victim machines, and no unauthorized access occurs. The victim simply stops offering service to normal clients because it is preoccupied with handling the attack traffic. While no unauthorized access to the victim of the denial-of-service flood occurs, a large number of other hosts have previously been compromised and controlled by the attacker, who uses them as attack weapons. In most cases, this is unauthorized access, by the legal definition of that term.

While the denial-of-service effect on the victim may sound relatively benign, especially when one considers that it usually lasts only as long as the attack is active, for many network users it can be devastating. Use of Internet services has become an important part of our daily lives. Following are some examples of the damaging effects of denial-of-service attacks.

• Sites that offer services to users through online orders make money only when users can access those services. For example, a large book-selling site cannot sell books to its customers if they cannot browse the site's Web pages and order products online. A denial-of-service attack on such sites means a severe loss of revenue for as long as the attack lasts. Prolonged or frequent attacks also inflict long-lasting damage to a site's reputation — customers who were unable to access the desired service are likely to take their business to the competition. Sites whose reputations were damaged may have trouble attracting new customers or investor funding in the future.

• Large news sites and search engines are paid by marketers to present their advertisements to the public. The revenue depends on the number of users that view the site's Web page. A denial-of-service attack on such a site means a direct loss of revenue from the marketers, and may have the long-lasting effect of driving the customers to more easily accessible sites. Loss of popularity translates to a direct loss of advertisers' business.

• Numerous businesses have come to depend on the Internet for critical daily activities. A denial-of-service attack may interrupt an important videoconference meeting or a large customer order.

• The Internet is increasingly being used to facilitate management of public services, such as water, power, and sewage, and to deliver critical information for important activities, such as weather and traffic reports for docking ships. A denial-of-service attack that disrupts these critical services will directly affect even people whose activities are not related to computers or the Internet. It may even endanger human lives.

• A vast number of people use the Internet on a daily basis for entertainment or for communicating with friends and family. While a denial-of-service attack that disrupts these activities may not cause them any serious damage, it is certainly an unpleasant experience that they wish to avoid. If such disruptions occur frequently, people are likely to stop using the Internet for these purposes, in favour of more reliable technologies.

Text 9. Information Warfare

In the past decade we have witnessed phenomenal growth in the capabilities of information management systems. National security implications of these capabilities are only now beginning to be understood by national leadership. There is no doubt IW is a concept the modern military officer should be familiar with, for advancements in computer technology have significant potential to dramatically change the face of military command and control.

Information warfare is an orchestrated effort to achieve victory by subverting or neutralizing an enemy command and control (C2) system, while protecting use of C2 systems to coordinate the actions of friendly forces. A successful IW campaign seizes initiative from an enemy commander; the IW campaign allows allied forces to operate at a much higher tempo than an enemy can react to.

The concept of an "OODA Loop" is often used to illustrate information warfare. OODA stands for the steps in a commander's decision making cycle — Observe, Orient, Decide and Act. Based on the premise that information is a strategic asset, a portion of IW doctrine seeks to disrupt or deny access to information in order to seize initiative from an adversary. The other half of IW doctrine seeks to maintain the integrity of our information gathering and distribution infrastructure.

Applying Information Warfare. Most modern political and military C2 systems are based on high speed communications and computers. It follows that this information infrastructure, also known as an "info-sphere", will be the arena in which information warfare is waged. Any system or person who participates in the C2 process will be a potential target in an IW campaign.

An IW campaign will focus against the enemy info-sphere. It will be necessary to isolate, identify and analyze each element of an enemy info-sphere in order to determine portions which can affect the OODA loop's size. Once these areas of the enemy info-sphere are identified, an attack against critical nodes would deny access to information, destroy the information, or render it useless to the adversary forces. Even more damaging, information warriors could

alter data in a network, causing the adversary to use false information in his decision making process and follow a game plan of the friendly commander's design.

Fighting the Information War. One development with implications for the military is the appearance of "hackers" and "phreakers" — persons who gain unauthorized access to computer and telephone systems, respectively. A computer network or telephone system is designed to transmit information. Much of that information will form an excellent intelligence picture of an adversary. Computer networks can be monitored through telephone modems, peripheral equipment, power lines, human agents and other means. If a system can be monitored remotely, it might also be accessed remotely. A program could be installed to record and relay computer access codes to a remote location. Employing computers as a weapon system will introduce a new glossary of terminology. Computer war fighting weapons can be divided into four categories: software, hardware, electromagnetic systems and other assets.

Software consists of programs designed to collect information on, inhibit, alter, deny use of, or destroy the enemy info-sphere. The examples of software war fighting assets have exotic, computer hacker names: "knowbot", "demons", "sniffers", "viruses", "Trojan horses", "worms" or "logic bombs".

A knowbot. (knowledge robot) is a program which moves from machine to machine, possibly cloning itself. KNOWBOTs can communicate with one another, with various servers in a network, and with users. The KNOWBOT could even be programmed to relocate or erase itself to prevent discovery of espionage activity. KNOWBOTs could seek out, alter or destroy critical nodes of an enemy C2 system. Demon. A program which, when introduced into a system, records all commands entered into the system. Similar to the demon is the "sniffer". A sniffer records the first 128 bits of data on a given program. Logon information and passwords are usually contained in this portion of any data stream. Because they merely read and record data, such programs are very difficult to detect.

Virus. A program which, upon introduction, attaches itself to resident files or tables on a machine or network. The virus spreads itself to other files as it comes into contact with them. It may reproduce without doing any actual damage, or it may erase files via the file allocation table.

Trap Door. A back door into a system, written in by a programmer to bypass future security codes.

Trojan Horse. A code which remains hidden within a computer system or network until it emerges to perform a desired function. A Trojan Horse can authorize access to the system, alter, deny or destroy data, or slow down system function.

Worm. A nuisance file which grows within an information storage system. It can alter files, take up memory space, or displace and overwrite valuable information.

Logic Bomb. This instruction remains dormant until a pre-determined condition occurs. Logic bombs are usually undetectable before they are activated. The logic bomb can alter, deny or destroy data and inhibit system function.

Hardware. The primary purpose of a hardware asset is to bring software assets into contact with an enemy computer system. Any piece of equipment connected to a computer, be it a fibre-optic or telephone cable, facsimile machine or printer, is capable of transmitting information to that computer. Therefore it is a potential avenue for gaining access to the info-sphere.

Electromagnetic Systems. Any mechanisms using the electromagnetic spectrum to subvert, disrupt or destroy enemy command and control are electromagnetic systems. Electromagnetic pulse simply shorts-out electronic equipment.

Other assets. This catch-all category makes an important point. Information warfare is not limited to electronic systems. Simply put, non-computer assets can compliment use of computer hardware and software assets, or can act unilaterally. Their goal is to achieve the desired effect upon the enemy C2 network in pursuit of strategic, operational or tactical objectives. Successful employment of IW assets could theoretically end a war before the first shot is fired. IW doctrine has significant implications for modern military theory. IW will focus on preventing the enemy soldier from talking to his commander. Without coordinated action, an enemy force becomes an unwieldy mob, and a battle devolves to a crowd-control issue. In the not too distant future, computer weapon systems will conduct "software strikes" against the enemy infosphere to disrupt command and control. Targets will be chosen for military, political or economic significance. IW opens new doors throughout the spectrum of conflict to achieve tactical, operational and strategic objectives.

Information warfare is a concept which is only now beginning to make its way through governmental and military circles. The technology currently exists with which to conduct an IW campaign. National leaders must reflect on the implications of this new technology in order to develop coherent policy and rules of engagement.

Text 10. Information Warfare: Its Application in Military and Civilian Contexts

The lexicon of information warfare (IW), or cyber war, to use a common variant, has been around for more than two decades, but for most of that time it has remained the preserve of the defence community. The privileging of military thinking is myopic. Information warfare concepts deserve to be liberated from their military associations and introduced into other discourse communities concerned with understanding the social consequences of pervasive computing. Already, the principles and practices of information warfare are being exhibited, more or less wittingly, in a variety of civilian contexts, and there are good grounds for assuming that this trend will intensify, causing potentially serious social problems and creating novel challenges for the criminal justice system. To paraphrase a well-worn cliché, information warfare is too important to be left to the military.

The term "information warfare" is still popularly associated with high-technology weapons and broadcast images of Cruise missiles seeking out Iraqi or other military targets with apparently unerring accuracy. The media's early focus on smart bombs and intelligent battle systems masked the potentially deeper societal implications of virtual warfare strategies. That, however, is beginning to change, as journalists and pundits foreground computer hacking and data corruption as pivotal information warfare techniques. Simplifications and confusions notwithstanding, an axial assumption of information age warfare is that brains matter more than brawn. In tomorrow's battlefield, be it military or civilian, information technology will act as a force multiplier. Traditional notions about the bases of superiority existing between attacker and target may thus require redefinition.

Pandemic access to digital networks creates a downward adjustment of established power differentials at all levels of society.

The principles and practice of information warfare have potentially much wider implications for society at large in a networked age. We consider four spheres of activity in which information warfare may very soon become relatively commonplace: military, corporate/economic, community/social, and personal.

The Military Context. The term "information warfare" is widely used within the defence community. Information warfare implies a range of measures or actions intended to protect,

exploit, corrupt, deny, or destroy information or information resources in order to achieve a significant advantage, objective, or victory over an adversary. A typical goal of conventional warfare is to destroy or degrade the enemy's physical resources, whereas the aim of IW is to target information assets and infrastructure, such that the resultant damage may not be immediately visible or detectable to the untrained eye. These strikes are called soft kills. In practical terms, cyber warfare means infiltrating, degrading, or subverting the target's information systems using logic bombs or computer viruses. But it also extends traditional notions of psychological warfare. An IW goal may be silent penetration of the target's information and communications system in order to shape community perceptions, foster deception, or seed uncertainty. In the battle for hearts and minds, the control of broadcast technologies has been a prime objective. In one sense, nothing much has changed, but the picture has become more complicated with the emergence of the Internet and World Wide Web, which give voice to the most unlikely individuals and groups, their multidirectional communication properties affording access to audiences that, under monopolistic or oligopolistic broadcasting conditions, would have remained permanently out of reach. In the information age, the silent enemy can easily acquire a voice and quickly amplify its dissident message.

Of course, IW constitutes a double-edged sword for information-intensive nations. The greater the military's reliance on complex networks and smart weaponry, the greater is its potential vulnerability to stealth attack by materially much weaker enemies blessed with networking savvy, be they foreign agents or corrupted insiders. And it is this aspect of IW — resource "asymmetricality" — that has attracted so much attention among both military planners and media analysts and shifted much of the discussion from offensive to defensive information warfare strategy.

Corporate/Economic Information Warfare. The similarities between the military and business world grow each day. Both involve competition between adversaries with various assets, motives, and goals. With the progressive globalization of trade and internationalization of business, the parallels will intensify. Business would thus seem to be an obvious site to appropriate the discourse of information warfare. Given the growing dependence of companies on sophisticated information systems, and, more particularly, the rapid growth of Web-based electronic commerce, it is reasonable to conclude that information warfare theory will soon establish a curricular foothold in leading business schools. In the age of economic and corporate information warfare, proactive intelligence management systems become essential requirements for high-performing companies.

Community/Social Information Warfare. The rise of the networked society has resulted in an intensification of debate on a vast array of social issues. What makes distributed computing, or more specifically the Internet, so attractive to individuals or groups interested in having their opinions heard or waging word warfare with others is the lack of restraints. Gate keeping, particularly in the public sphere, is not a new concept. Government control of the Internet, however, has not followed that of newspapers, radio, and television. Consequently, the Internet remains a communications arena where discourse, positive and negative, rational and irrational, flourishes freely. With the advent of the World Wide Web and graphical user interfaces or browsers such as Netscape, social and political activists have an even richer agora in which to debate, pontificate, or castigate. It is clear that de-layering and disintermediation — the loss of intervening controls — have created a climate conductive to information warfare and cyber-terrorism.

Personal Information Warfare. Ordinary citizens are vulnerable to various kinds of overt and covert attack by cyber-terrorists acting alone or in concert, whether the motivation is ostensibly ludic or demonstrably criminal. Hacker culture may dismiss electronic break-inns and impersonation as puckishly acceptable behaviours, but the victim will probably view matters differently. The sense of violation and loss of sanctuary can have long-lasting psychological effects. The reconstitution of trust and salvaging of reputations in the wake of virtual vilification campaigns will likely pose major challenges for targeted individuals and collectivities. As with military or business resources, an individual's information assets and online identity are potentially highly degradable by a determined hacker — which isn't to say that anything other than a minority of individuals will ever be targeted in systematic fashion by information warriors/terrorists. Ontological warfare is thus a novel option within the digital battle space.

It's clear that IW thinking need not be bounded by the discourse of the military community. The principles of information warfare and net terrorism are being instantiated in a diverse set of social contexts, though the range of motivations and practices varies greatly. Decoupled from their military roots, the language and principles of information warfare have enormously wide applicability. Internetworking technologies and emergence of complex computational communities provide the conditions to support multidimensional information warfare and net terrorism.

Unit III. Artificial Intelligence

Text 1. Introduction to Artificial Intelligence

Humankind has given itself the scientific name "homo sapiens" – man the wise – because our mental capacities are so important to our everyday lives and our sense of self. The field of artificial intelligence, or AI, attempts to understand intelligent entities. Thus, one reason to study it is to learn more about ourselves. But unlike philosophy and psychology, which are also concerned with intelligence, AI strives to build intelligent entities as well as understand them. Another reason to study AI is that these constructed intelligent entities are interesting and useful in their own right. AI has produced many significant and impressive products even at this early stage in its development. Although no one can predict the future in detail, it is clear that computers with human-level intelligence (or better) would have a huge impact on our everyday lives and on the future course of civilization.

AI addresses one of the ultimate puzzles. How is it possible for a slow, tiny brain, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself? How do we go about making something with those properties? These are hard questions, but unlike the search for faster-than-light travel or an antigravity device, the researcher in AI has solid evidence that the quest is possible. All the researcher has to do is look in the mirror to see an example of an intelligent system.

AI is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. Along with modern genetics, it is regularly cited as the "field I would most like to be in" by scientists in other disciplines. A student in physics might reasonably feel that all the good ideas have already been taken by Galileo, Newton, Einstein, and the rest, and that it takes many years of study before one can contribute new ideas. AI, on the other hand, still has openings for a full-time Einstein.

The study of intelligence is also one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done. AI has turned out to be more difficult than many at first imagined and modern ideas are much richer, more subtle, and more interesting as a result.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavour. In this sense, it is truly a universal field.

We have now explained why AI is exciting, but we have not said what it is. We could just say, "Well, it has to do with smart programs, so let's get on and write some". But the history of science shows that it is helpful to aim at the right goals. Early alchemists, looking for a potion for eternal life and a method to turn lead into gold, were probably off on the wrong foot. Only when the aim changed, to that of finding explicit theories that gave accurate predictions of the terrestrial world, in the same way that early astronomy predicted the apparent motions of the stars and planets, could the scientific method emerge and productive science take place. Definitions of artificial intelligence according to eight recent textbooks are shown in the table below. These definitions vary along two main dimensions. The ones on top are concerned with thought processes and reasoning, whereas the ones on the bottom address behaviour. The definitions on the left measure success in terms of human performance, whereas the ones on the right measure against an ideal concept of intelligence, which we will call rationality. A system is rational if it does the right thing.

"The exciting new effort to make computers think...machines with minds, in the full and literal sense". (Haugeland, 1985) "The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning..." (Bellman, 1978)	"The study of mental faculties through the use of computational models". (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act". (Winston, 1992)
«The art of creating machines that perform functions that require intelligence when performed by people". (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes". (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behaviour" (Luger and Stubblefield, 1993)

This gives us four possible goals to pursue in artificial intelligence:

Systems that think like humans.	Systems that think rationally.
Systems that act like humans	Systems that act rationally

Historically, all four approaches have been followed. As one might expect, a tension exists between approaches centred around humans and approaches centred around rationality. We should point out that by distinguishing between human and rational behaviour, we are not suggesting that humans are necessarily "irrational" in the sense of "emotionally unstable" or "insane". One merely need note that we often make mistakes; we are not all chess grandmasters even though we may know all the rules of chess; and unfortunately, not everyone gets the best mark on the exam. A human-centred approach must be an empirical science, involving hypothesis and experimental confirmation. A rationalist approach involves a combination of mathematics and engineering. People in each group sometimes cast aspersions on work done in the other groups, but the truth is that each direction has yielded valuable insights. Let us look at each in more detail.

Text 2. Approaches to AI (Part I). Acting humanly: The Turing Test approach

The Turing Test, proposed by Alan Turing, was designed to provide a satisfactory opera tional definition of intelligence. Turing defined intelligent behaviour as the ability to achiev human-level performance in all cognitive tasks, sufficient to fool an interrogator. Roughl speaking, the test he proposed is that the computer should be interrogated by a human via teletype, and passes the test if the interrogator cannot tell if there is a computer or a human a the other end. For now, programming a computer to pass the test provides plenty to work on The computer would need to possess the following capabilities:

1. Natural language processing to enable it to communicate successfully in English (or som other human language);
2. Knowledge representation to store information provided before or during the interrogation;
3. Automated reasoning to use the stored information to answer questions and to draw nev conclusions;
4. Machine learning to adapt to new circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator an the computer, because physical simulation of a person is unnecessary for intelligence. How ever, the so-called total Turing Test includes a video signal so that the interrogator can test th subject's perceptual abilities, as well as the opportunity for the interrogator to pass physica objects "through the hatch". To pass the total Turing Test, the computer will need computc vision to perceive objects, and robotics to move them about.

Within AI, there has not been a big effort to try to pass the Turing Test. The issue of actin like a human comes up primarily when AI programs have to interact with people, as when a expert system explains how it came to its diagnosis, or a natural language processing syste has a dialogue with a user. These programs must behave according to certain normal conven tions of human interaction in order to make themselves understood. The underlying represen tation and reasoning in such a system may or may not be based on a human model.

Thinking humanly: The cognitive modelling approach

If we are going to say that a given program thinks like a human, we must have some way c determining how humans think. We need to get inside the actual workings of human minds There are two ways to do this: through introspection – trying to catch our own thoughts as they go by – or through psychological experiments. Once we have a sufficiently precise theory c the mind, it becomes possible to express the theory as a computer program. If the program's input/output and timing behaviour matches human behaviour, that is evidence that some of th program's mechanisms may also be operating in humans. For example, Newell and Simon who developed the "General Problem Solver" (1961), were not content to have their progran correctly solve problems. They were more concerned with comparing the trace of its reasonin steps to traces of human subjects solving the same problems. This is in contrast to other re searchers of the same time such as Wang (1960), who were concerned with getting the righ answers regardless of how humans might do it.

The interdisciplinary field of cognitive science brings together computer models from AI and ex perimental techniques from psychology to try to construct precise and testable theories of th workings of the human mind. Real cognitive science, however, is necessarily based on experimen tal investigation of actual humans or animals, and we assume that the reader only has access to computer for experimentation. We will simply note that AI and cognitive science continue to fer tilize each other, especially in the areas of vision, natural language, and learning.

Text 3. Approaches to AI (Part II). Thinking rationally: The laws of thought approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking", that is, irrefutable reasoning processes. His famous syllogisms provided patterns for argument structures that always gave correct conclusions given correct premises. For example, "Socrates is a man; all men are mortal; therefore Socrates is mortal". These laws of thought were supposed to govern the operation of the mind, and initiated the field of logic.

The development of formal logic in the late nineteenth and early twentieth centuries provided a precise notation for statements about all kinds of things in the world and the relations between them. Contrast this with ordinary arithmetic notation, which provides mainly for equality and inequality statements about numbers. By 1965, programs existed that could, given enough time and memory, take a description of a problem in logical notation and find the solution to the problem, if one exists. If there is no solution, the program might never stop looking for it. The so-called "logicist" tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem "in principle" and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to any attempt to build computational reasoning systems, they appeared first in the "logicist" tradition because the power of the representation and reasoning systems are well-defined and fairly well understood.

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals. An agent is just something that perceives and acts. This may be an unusual use of the word, but you will get used to it. In this approach, AI is viewed as the study and construction of rational agents.

In the "laws of thought" approach to AI, the whole emphasis was on correct inferences. Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals, and then to act on that conclusion. On the other hand, correct inference is not all of rationality because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also ways of acting rationally that cannot be reasonably said to involve inference. For example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the "cognitive skills" needed for the Turing Test are there to allow rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception not just because seeing is fun, but in order to get a better idea of what an action might achieve – for example, being able to see a tasty morsel helps one to move toward it.

The study of AI as rational agent design therefore has two advantages. First, it is more general than the "laws of thought" approach, because correct inference is only a useful mecha-

nism for achieving rationality, and not a necessary one. Second, it is more amenable to scientific development than approaches based on human behaviour or human thought, because the standard of rationality is clearly defined and completely general. Human behaviour, on the other hand, is well-adapted for one specific environment and is the product, in part, of a complicated and largely unknown evolutionary process that still may be far from achieving perfection. We will see that despite the apparent simplicity with which the problem can be stated, an enormous variety of issues come up when we try to solve it.

Text 4. Artificial Intelligence and Expert Systems

There is a class of computer programs, known as expert systems that aim to mimic human reasoning. The methods and techniques used to build these programs are the outcome of efforts in a field of computer science known as Artificial Intelligence (AI). Expert systems have been built to diagnose disease, translate natural languages, and solve complex mathematical problems.

In conventional computer programs, problem-solving knowledge is encoded in program logic and program-resident data structures. Expert systems differ from conventional programs both in the way problem knowledge is stored and used.

An expert system is a computer program, with a set of rules encapsulating knowledge about a particular problem domain (i.e. medicine, chemistry, finance, flight and so on). These rules prescribe actions to take when certain conditions hold, and define the effect of the action on deductions or data. The expert system, seemingly, uses reasoning capabilities to reach conclusions or to perform analytical tasks. Expert systems that record the knowledge needed to solve a problem as a collection of rules stored in a knowledge-base are called rule-based systems.

Expert systems are especially important to organizations that rely on people who possess specialized knowledge of some problem domain, especially if this knowledge and experience cannot be easily transferred. Artificial intelligence methods and techniques have been applied to a broad range of problems and disciplines, some of which are esoteric and others which are extremely practical.

Expert systems are used by the investments, banking, and telecommunications industries. They are essential in robotics, natural language processing, theorem proving, and the intelligent retrieval of information from databases. They are used in many other human endeavours which might be considered more practical. Rule-based systems have been used to monitor and control traffic, to aid in the development of flight systems.

A rule-based, expert system maintains a separation between its knowledge base and that part of the system that executes rules, often referred to as the expert system shell. The system shell is indifferent to the rules it executes. This is an important distinction, because it means that the expert system shell can be applied to many different problem domains with little or no change. It also means that adding or modifying rules to an expert system can effect changes in program behaviour without affecting the controlling component, the system shell.

The language used to express a rule is closely related to the language subject matter experts use to describe problem solutions. When the subject matter expert composes a rule using this language, he is, at the same time, creating a written record of problem knowledge, which can then be shared with others. Thus, the creation of a rule kills two birds with one stone; the rule adds functionality or changes program behaviour, and records essential information about the problem domain in a human-readable form. Knowledge captured and main-

tained by these systems ensures continuity of operations even as subject matter experts (i.e. mathematicians, accountants, physicians) retire or transfer.

Furthermore, changes to the knowledge-base can be made easily by subject matter experts without programmer intervention, thereby reducing the cost of software maintenance and helping to ensure that changes are made in the way they were intended. Rules are added to the knowledge-base by subject matter experts using text or graphical editors that are integral to the system shell.

Finally, the expert system never forgets, can store and retrieve more knowledge than any single human being can remember, and makes no errors, provided the rules created by the subject matter experts accurately model the problem at hand.

Text 5. Artificial Neural Networks (Part I). What Are Artificial Neural Networks?

A neural network is an attempt to simulate the brain. Neural network theory revolves around the idea that certain key properties of biological neurons can be extracted and applied to simulations, thus creating a simulated (and very much simplified) brain. The first important thing to understand is that the components of an artificial neural network are an attempt to recreate the computing potential of the brain. The second important thing to understand, however, is that no one has ever claimed to simulate anything as complex as an actual brain. Whereas the human brain is estimated to have something on the order of ten to a hundred billion neurons, a typical artificial neural network (ANN) is not likely to have more than 1,000 artificial neurons.

Before discussing the specifics of artificial neural nets though, let us examine what makes real neural nets-brains function the way they do. Perhaps the single most important concept in neural net research is the idea of connection strength. Neuroscience has given us good evidence for the idea that connection strengths – that is, how strongly one neuron influences those neurons connected to it – are the real information holders in the brain. Learning, repetition of a task, even exposure to a new or continuing stimulus can cause the brain's connection strengths to change, some synaptic connections becoming reinforced and new ones are being created, others weakening or in some cases disappearing altogether. The second essential element of neural connectivity is the excitation/inhibition distinction. In human brains, each neuron is either excitatory or inhibitory, which is to say that its activation will either increase the firing rates of connected neurons, or decrease the rate, respectively. The amount of excitation or inhibition produced is of course, dependent on the connection strength – a stronger connection means more inhibition or excitation, a weaker connection means less. The third important component in determining a neuron's response is called the transfer function. Without getting into more technical detail, the transfer function describes how a neuron's firing rate varies with the input it receives. A very sensitive neuron may fire with very little input, for example. A neuron may have a threshold, and fire rarely below threshold, and vigorously above it. A neuron may have a bell-curve style firing pattern, increasing its firing rate up to a maximum, and then levelling off or decreasing when over-stimulated. A neuron may sum its inputs, or average them, or something entirely more complicated. Each of these behaviours can be represented mathematically, and that representation is called the transfer function. It is often convenient to forget the transfer function, and think of the neurons as being simple addition machines, more activity in equals more activity out. This is not really accurate though, and to develop a good understanding of an artificial neural network, the transfer function must be taken into account.

Armed with these three concepts: Connection Strength, Inhibition/Excitation, and the Transfer Function, we can now look at how artificial neural nets are constructed. In theory, an artificial neuron (often called a "node") captures all the important elements of a biological one. Nodes are connected to each other and the strength of that connection is normally given a numeric value between –1.0 for maximum inhibition, to +1.0 for maximum excitation. All values between the two are acceptable, with higher magnitude values indicating stronger connection strength. The transfer function in artificial neurons whether in a computer simulation, or actual microchips wired together is typically built right into the nodes' design.

Perhaps the most significant difference between artificial and biological neural nets is their organization. While many types of artificial neural nets exist, most are organized according to the same basic structure. There are three components to this organization: a set of input nodes, one or more layers of "hidden" nodes, and a set of output nodes. The input nodes take in information, and are akin to sensory organs. Whether the information is in the form of a digitised picture, or a series of stock values, or just about any other form that can be numerically expressed, this is where the net gets its initial data. The information is supplied as activation values, that is, each node is given a number, higher numbers representing greater activation. This is just like human neurons except that rather than conveying their activation level by firing more frequently, as biological neurons do, artificial neurons indicate activation by passing this activation value to connected nodes. After receiving this initial activation, information is then passed through the network. Connection strengths, inhibition/excitation conditions, and transfer functions determine how much of the activation value is passed on to the next node. Each node sums the activation values it receives, arrives at its own activation value, and then passes that along to the next nodes in the network (after modifying its activation level according to its transfer function). Thus the activation flows through the net in one direction, from input nodes, through the hidden layers, until eventually the output nodes are activated. If a network is properly trained, this output should reflect the input in some meaningful way. For instance, gender recognition net might be presented with a picture of a man or woman at its input nodes and must set an output node to 0.0 if the picture depicts a man or 1.0 for a woman. In this way, the network communicates its knowledge to the outside world.

Text 6. Artificial Neural Networks (Part II). How Do Artificial Neural Networks Learn?

Having explained that connection strengths are storehouses of knowledge in neural net architectures, it should come as no surprise that learning in neural nets is primarily a process of adjusting connection strengths. In neural nets of the type described so far, the most popular method of learning is called back-propagation. To begin, the network is initialised, all the connection strengths are set randomly, and the network sits as a blank slate. The network is then presented with some information; let us suppose that we are designing the "gender detector" mentioned earlier, and that the input nodes are receiving a digitised version of a photograph. The activation flows through the net (albeit haphazardly since we have not yet set the connection strengths to anything but random values). And eventually the output node registers an activation level. However, since the net has not yet been trained, its responses will initially be random. This is where back-propagation steps in. The net's response is compared with the correct response for that picture (i.e. 0.0 for male, 1.0 for female). Then working backwards from the output node, each connection strength is adjusted so that next time it's shown that picture, its answer will be closer to the desired one.

This whole process: input, processing, comparing output with correct answer and adjusting connection strengths is called one 'back-propagation cycle", or often "iteration". The net is then presented with another picture and its answer is compared with the correct answer, the connection strengths adjusted where needed. This process can often take hundreds or thousands of iterations. Eventually, the net should become fairly proficient at identifying males and females. There is always a risk however, that the net has not learned to discriminate males from females, but rather that it has effectively memorized the response for each picture. To test for this, the pictures (or whatever input is being used) should be divided into two groups: the training set, and the transfer set. The training set is used during back-propagation cycles, and the transfer set is used once learning is complete. If the net performs as well on the novel transfer stimuli as it did on the training set, then we conclude that learning has occurred.

It's fine in theory to talk about neural nets that tell males from females, but if that was all they were useful for, they would be a sad project indeed. In fact, neural nets have been enjoying growing success in a number of fields, and significantly: their successes tend to be in fields that posed large difficulties for symbolic AI. Neural networks are, by design, pattern processors – they can identify trends and important features, even in relatively complex information. What's more, they can work with less-than-perfect information, such as blurry or static-filled pictures, which has been an insurmountable difficulty for symbolic AI systems. Discerning patterns allows neural nets to read handwriting, detect potential sites for new mining and oil extraction, predict the stock market, and even learn to drive.

Interestingly, neural nets seem to be good at the same things we are, and struggle with the same things we struggle with. Symbolic AI is very good at producing machines that play grandmaster-level chess, that deduce logic theorems, and that compute complex mathematical functions. But Symbolic AI has enormous difficulty with things like processing a visual scene, dealing with noisy or imperfect data, and adapting to change. Neural nets are almost the exact reverse – their strength lies in the complex, fault-tolerant, parallel processing involved in vision, and their weaknesses are in formal reasoning and rule-following. Although humans are capable of both forms of intellectual functioning, it is generally thought that humans possess exceptional pattern recognition ability. In contrast, the limited capacity of human information processing systems often makes us less-than-perfect in tasks requiring abstract reasoning and logic.

Critics charge that a neural net's inability to learn something like logic, which has distinct and unbreakable rules, proves that neural nets cannot be an explanation of how the mind works. Neural net advocates have countered that a large part of the problem is that abstract rule-following ability requires many more nodes than current artificial neural nets implement. Some attempts are now being made at producing larger networks, but the computational load increases dramatically as nodes are added, making larger networks very difficult. Another set of critics charge that neural nets are too simplistic to be considered accurate models of human brain function. While artificial neural networks do contain some neuron-like attributes (connection strengths, inhibition/excitation, etc.) they overlook many other factors which may be significant to the brain's functioning. The nervous system uses many different neurotransmitters, for instance, and artificial neural nets do not account for those differences. Different neurons have different conduction velocities, different energy supplies, even different spatial locations, which may be significant. Moreover, brains do not start as a jumbled, randomised set of connection strengths, there is a great deal of organization present. Any or all of these can be seen as absolutely essential to the functioning of the brain, and without their inclusion in the artificial neural network models, it is possible that the models end up oversimplified.

One of the fundamental objections that has been raised towards back-propagation style networks like the ones discussed here is that humans seem to learn even in the absence of an explicit 'teacher' which corrects our outputs and models the response. For neural networks to succeed as a model of cognition, it is imperative that they produce a more biologically (or psychologically) plausible simulation of learning. In fact, research is being conducted with a new type of neural net, known as an 'Unsupervised Neural Net', which appears to successfully learn in the absence of an external teacher.

Supplementary Reading

Text. Artificial Intelligence in Education

For decades, science fiction authors, futurists, and movie makers have been predicting the amazing changes that will arise with the advent of widespread artificial intelligence. So far, AI hasn't made any such extreme waves, and in many ways has quietly become ubiquitous in numerous aspects of our daily lives. From the intelligent sensors that help us take perfect pictures, to the automatic parking features in cars, to the sometimes frustrating personal assistants in smart phones, artificial intelligence of one kind or another is all around us all the time. One place where artificial intelligence is poised to make big changes (and in some cases already is) is in education.

While we may not see humanoid robots acting as teachers within the next decade, there are many projects already in the works that use computer intelligence to help students and teachers get more out of the educational experience. Here are just a few of the ways those tools, will shape and define the educational experience of the future.

Artificial intelligence can automate basic activities in education, like grading. In college, grading homework and tests for large lecture courses can be tedious work. Even in lower grades, teachers often find that grading takes up a significant amount of time, time that could be used to interact with students, prepare for class, or work on professional development.

While AI may not ever be able to truly replace human grading, it's getting pretty close. It's now possible for teachers to automate grading for nearly all kinds of multiple choice and fill-in-the-blank testing and automated grading of student writing may not be far behind. Today, essay-grading software is still in its infancy, yet it can (and will) improve over the coming years, allowing teachers to focus more on in-class activities and student interaction than grading.

Educational software can be adapted to student needs. From kindergarten to graduate school, one of the key ways artificial intelligence will impact education is through the application of greater levels of individualized learning. Some of this is already happening through growing numbers of adaptive learning programs, games, and software. These systems respond to the needs of the student, putting greater emphasis on certain topics, repeating things that students haven't mastered, and generally helping students to work at their own pace, whatever that may be.

This kind of custom tailored education could be a machine-assisted solution to helping students at different levels work together in one classroom, with teachers facilitating the learning and offering help and support when needed. Adaptive learning has already had a huge impact on education, and as AI advances in the coming decades adaptive programs will likely only improve and expand.

Students could get additional support from AI tutors. While there are obviously things that human tutors can offer that machines can't, at least not yet, the future could see more

students being tutored by tutors that only exist "in zeros and ones". Some tutoring programs based on artificial intelligence already exist and can help students through basic mathematics, writing, and other subjects.

These programs can teach students fundamentals, but so far aren't ideal for helping students learn high-order thinking and creativity, something that real-world teachers are still required to facilitate. Yet that shouldn't rule out the possibility of AI tutors being able to do these things in the future. With the rapid pace of technological advancement that has marked the past few decades, advanced tutoring systems may not be a dream.

AI-driven programs can give students and educators helpful feedback. AI can not only help teachers and students to craft courses that are customized to their needs, but it can also provide feedback to both about the success of the course as a whole. Some schools, especially those with online offerings, are using AI systems to monitor student progress and to alert professors when there might be an issue with student performance.

These kinds of AI systems allow students to get the support they need and for professors to find areas where they can improve instruction for students who may struggle with the subject matter. AI programs at these schools aren't just offering advice on individual courses, however. Some are working to develop systems that can help students to choose majors based on areas where they succeed and struggle. While students don't have to take the advice, it could mark a brave new world of college major selection for future students.

It could change the role of teachers. There will always be a role for teachers in education, but what that role is and what it entails may change due to new technology in the form of intelligent computing systems. As we've already discussed, AI can take over tasks like grading, can help students improve learning, and may even be a substitute for real-world tutoring. Yet AI could be adapted to many other aspects of teaching as well. AI systems could be programmed to provide expertise, serving as a place for students to ask questions and find information or could even potentially take the place of teachers for very basic course materials. In most cases, however, AI will shift the role of the teacher to that of facilitator.

Teachers will supplement AI lessons, assist students who are struggling, and provide human interaction and hands-on experiences for students. In many ways, technology is already driving some of these changes in the classroom, especially in schools that are online or embrace the classroom model.

AI can make trial-and-error learning less intimidating. Trial and error is a critical part of learning, but for many students, the idea of failing, or even not knowing the answer, is paralyzing. Some simply don't like being put on the spot in front of their peers or authority figures like a teacher. An intelligent computer system, designed to help students to learn, is a much less daunting way to deal with trial and error. Artificial intelligence could offer students a way to experiment and learn in a relatively judgment-free environment, especially when AI tutors can offer solutions for improvement. In fact, AI is the perfect format for supporting this kind of learning, as AI systems themselves often learn by a trial-and-error method.

Data powered by AI can change how schools find, teach, and support students. Smart data gathering, powered by intelligent computer systems, is already making changes to how colleges interact with prospective and current students. From recruiting to helping students choose the best courses, intelligent computer systems are helping make every part of the college experience more closely tailored to student needs and goals.

Data mining systems are already playing an integral role in today's higher-education landscape, but artificial intelligence could further alter higher education. Initiatives are already underway at some schools to offer students AI-guided training that can ease the transition between college and high school.

AI may change where students learn, who teaches them, and how they acquire basic skills. While major changes may still be a few decades in the future, the reality is that artificial intelligence has the potential to radically change just about everything we take for granted about education.

Using AI systems, software, and support, students can learn from anywhere in the world at any time, and with these kinds of programs taking the place of certain types of classroom instruction, AI may just replace teachers in some instances (for better or worse). Educational programs powered by AI are already helping students to learn basic skills, but as these programs grow and as developers learn more, they will likely offer students a much wider range of services. What is the result? Education could look a whole lot different a few decades from now.

Text. Top 10 Most Popular Programming Languages

Java uses a compiler, and is an object-oriented language released in 1995 by Sun Microsystems. Java is the number one programming language today for many reasons. First, it is a well-organized language with a strong library of reusable software components. Second, programs written in Java can run on many different computer architectures and operating systems because of the use of the JVM (Java virtual machine). Sometimes this is referred to as code portability or even WORA (write once, run anywhere). Third, Java is the language most likely to be taught in university computer science classes. A lot of computer science theory books written in the past decade use Java in the code examples. So learning Java syntax is a good idea even if you never actually code in it.

JavaScript is an interpreted, multi-paradigm language. Despite its name, it has nothing whatsoever to do with Java. You will rarely, if ever, see this language outside of a web browser. It is basically a language meant to script behaviours in web browsers and used for things such as web form validation and AJAX style web applications. The trend in the future seems to be building more and more complex applications in JavaScript, even simple online games and office suites. The success of this trend will depend upon advancements in the speed of a browser's JavaScript interpreter. If you want to be correct, the real name of this programming language is ECMAscript, although almost nobody actually calls it this.

C is a compiled, procedural language developed in 1972 by Dennis Ritchie for use in the UNIX operating system. Although designed to be portable in nature, C programs must be specifically compiled for computers with different architectures and operating systems. This helps make them lightning fast. Although C is a relatively old language, it is still widely used for system programming, writing other programming languages, and in embedded systems.

C++ is a compiled, multi-paradigm language written as an update to C in 1979 by Bjarne Stroustrup. It attempts to be backwards-compatible with C and brings object-orientation, which helps in larger projects. Despite its age, C++ is used to create a wide array of applications from games to office suites.

C# is a compiled, object-oriented language written by Microsoft. It is an open specification, but rarely seen on any non-Windows platform. C# was conceived as Microsoft's premium language in its.NET Framework. It is very similar to Java in both syntax and nature.

PHP uses a run-time interpreter, and is a multi-paradigm language originally developed in 1996 by Rasmus Lerdorf to create dynamic web pages. At first it was not even a real programming language, but over time it eventually grew into a fully featured object-oriented programming language. Although PHP has been much criticized in the past for being a bit sloppy and insecure, it's been pretty good since version 5 came out in 2004. It's hard to argue with success. Today, PHP is the most popular language used to write web applications.

Visual Basic is an interpreted, multi-paradigm language developed by Microsoft Corporation for the Windows platform. It has been evolving over the years and is seen as a direct descendant of Microsoft's old BASIC from the 1970's. Visual Basic is a good language for scripting Windows applications that do not need the power and speed of C#.

Python is an interpreted, multi-paradigm programming language written by Guido van Rossum in the late 1980's and intended for general programming purposes. Python was not named after the snake but actually after the Monty Python comedy group. Python is characterized by its use of indentation for readability, and its encouragement for elegant code by making developers do similar things in similar ways. Python is used as the main programming choice of both Google and Ubuntu.

Perl is an interpreted, multi-paradigm language written by Larry Wall in 1986. It is characterized by a somewhat disorganized and scary-looking syntax which only makes sense to other PERL programmers ;) However, a lot of veteran programmers love it and use if every day as their primary language. 10 years ago, Perl was more popular than it is today. What happened? A lot of newer programmers and even old Perl programmers (such as myself) have switched to other languages such as PHP, Python, and Ruby. Perl is perhaps still the best language for text processing and system administration scripting. I personally do not recommend it however as a primary programming language.

Ruby is an interpreted, object-oriented language written by Yukihiro Matsumoto around 1995. It is one of the most object-oriented languages in the world. Everything is an object in Ruby, even letters and numbers can have method calls. It's a great language to learn if you love objects. The only negative is that its love of object-orientation makes it a bit slow, even for an interpreted language.

Text. Object-Oriented Programming

The abbreviation "OO", which stands for object oriented, is used to describe a programming paradigm as well as a variety of computer programming languages.

Objects and classes. The object-oriented paradigm is based on the idea that the solution for a problem can be visualized in terms of objects that interact with each other. In the context of this paradigm, an object is a unit of data that represents an abstract or a real world entity, such as a person, place, or thing. For example, an object can represent a $10.99 small pepperoni pizza. Another one can represent a pizza delivery guy named Jack Flash. Yet another object can be a customer living at 22 Pointe Rd.

The real world contains lots of pizzas, customers, and delivery guys. These objects can be defined in a general way by using classes. Whereas an object is a single instance of an entity, a class is a template for a group of objects with similar characteristics. For example, a Pizza class defines a group of gooey Italian snacks that are made in a variety of sizes, crafted into rectangular or round shapes, and sold for various prices. A class can produce any number of unique objects.

When taking the object-oriented approach to a problem, one of the first steps is to identify the objects that pertain to a solution. As you might expect, the solution to the pizza problem requires some pizza objects. Certain characteristics of pizzas provide information necessary to solve the problem. This information – the price, size, and shape of a pizza – provides the structure for the Pizza class. A class is defined by attributes and methods. A class attribute defines the characteristics of a set of objects.

Each class attribute typically has a name, scope and data type. One class attribute of the Pizza class might be named "pizzaPrice". Its scope can be defined as public or private. A public attribute is available for use by any routine in the program. A private attribute can be accessed only from the routine in which it is defined. The pizzaPrice attribute's data type can be defined as "double", which means that it can be any decimal number. OO programmers often use UML (Unified Modeling Language) diagrams to plan the classes for a program. Although a programmer completes the overall program plan before coding, jump ahead to take a quick look at the Java code for the attributes in the Pizza class. The first line of code defines the name of the class. Each subsequent line defines the scope, data type, and name of an attribute. The curly brackets simply define the start and end of the class.

```
Class Pizza
{
        public string pizzaShape;
        public double pizzaPrice;
        public double pizzaSize;
}
```

Inheritance. The object-oriented paradigm endows classes with quite a bit of flexibility. For the pizza program, objects and classes make it easy to compare round pizzas to rectangular pizzas rather than just to square pizzas.

Suppose you want to compare a 10-inch round pizza to a rectangular pizza that has a length of 11 inches and a width of 8 inches. The Pizza class holds only one measurement for each pizza—pizzaSize. This single attribute won't work for rectangular pizzas, which might have a different length and width. Should you modify the class definition to add attributes for pizzaLength and pizzaWidth? No, because these attributes are necessary only for rectangular pizzas, not for round pizzas. An OO feature called "inheritance" provides flexibility to deal with objects' unique characteristics.

In object-oriented jargon, inheritance refers to passing certain characteristics from one class to other classes. For example, to solve the pizza problem, a programmer might decide to add a RoundPizza class and a RectanglePizza class. These two new classes can inherit attributes from the Pizza class, such as pizzaShape and pizzaPrice. You can then add specialized characteristics to the new classes. The RectanglePizza class can have attributes for length and width, and the RoundPizza class can have an attribute for diameter.

The process of producing new classes with inherited attributes creates a superclass and subclasses. A superclass, such as Pizza, is any class from which attributes can be inherited. A subclass (or "derived class"), such as RoundPizza or RectanglePizza, is any class that inherits attributes from a superclass. The set of superclasses and subclasses that are related to each other is referred to as a class hierarchy. Java uses the "extends" command to link a subclass to a superclass. The statement class RectanglePizza extends Pizza means "create a class called RectanglePizza that's derived from the superclass called Pizza".

```
class RectanglePizza extends Pizza
{
double pizzaLength;
double pizzaWidth;
}
```

Methods and messages. An OO program can use objects in a variety of ways. A basic way to use objects is to manipulate them with methods. A method is a segment of code that defines an action. The names of methods usually end in a set of parentheses, such as compare() or getArea().

A method can perform a variety of tasks, such as collecting input, performing calculations, making comparisons, executing decisions, and producing output. For example, the pizza program can use a method named compare () to compare the square-inch prices of two pizzas and display a message indicating the best pizza.

A method begins with a line that names the method and can include a description of its scope and data type. The scope—public or private—specifies which parts of the program can access the method. The data type specifies the kind of data, if any, that the method produces. The initial line of code is followed by one or more lines that specify the calculation, comparison, or routine that the method performs.

A method is activated by a message, which is included as a line of program code, sometimes referred to as a "call". In the object-oriented world, objects often interact to solve a problem by sending and receiving messages. For example, a pizza object might receive a message asking for the pizza's area or price per square inch.

Polymorphism, sometimes called "overloading", is the ability to redefine a method in a subclass. It allows programmers to create a single, generic name for a procedure that behaves in unique ways for different classes. Polymorphism provides OO programs with easy extensibility and can help simplify program code.

A

access token – маркер доступа
add-on – дополнение
advent – появление
adversary – противник
albeit – хотя (и)
alert – бдительный, настороже, осторожный
alert – уведомление, предупреждение
allied forces – союзники
alter – изменять
altering – изменение
ambiguity – неопределённость, неясность; двусмысленность
amenable to – поддающийся
artificial neural network – искусственная нейронная сеть
ascertain – устанавливать, выяснять
ascertain – установить, удостовериться
aspersion – клевета
assembly language – язык ассемблера
assess – определять, оценивать
asset – ресурс, имущество
assets – активы, ресурсы
assumption – предположение, допущение
attempt – пытаться, стараться, стремиться, пробовать, делать попытку
audit – проверять
audit trail – контрольный журнал, файл регистрации сетевых событий
audit –проверка, ревизия
aware – сознающий; знающий, осведомленный

B

back-propagation – обратная передача ошибки обучения
backup – резервирование, дублирование, копирование
backward compatibility – обратная совместимость, полная совместимость с предыдущими версиями
bandwidth – полоса; полоса пропускания
behaviour – поведение
benign – неопасный, безвредный
blunt – ослабить
blurry – неясный; расплывчатый, смазанный
buffer overflow – переполнение буфера
bundling – объединение, комплектация
burden – груз; бремя

C

capture – захватывать, перехватывать; собирать
capture – захватывать, перехватывать; собирать

cautious – осторожный, предусмотрительный
circumstance – обстоятельство; условие
cite – ссылаться
CMS (content management system) – система управления контентом
cognitive – познавательный
coherent policy and rules – согласованная политика и правовые нормы
coil – спираль
collocate – располагать
compiler – компилятор
comprehensible – понятный, постижимый, ясный
concise – краткий; сжатый; лаконичный
conclusion – умозаключение, вывод
convention – правило поведения
credentials – полномочия, имя пользователя и пароль
customization – настройка или изготовление продукта под требования

D

data structure – структура данных
debugger – отладчик
debugging – отладка
decision-making – принятие решения
deem – считать, думать
deface – искажать, портить
defence – защита
defense community – службы безопасности
definition – определение
deliberation – размышление, взвешивание, обдумывание
denial of service – отказ в обслуживании
depict – изображать
derive – получать, извлекать
descendant – наследник, потомок
detect – замечать, открывать, обнаруживать
devastating – разрушительный
differential backup – дифференциальное резервное копирование
digital battle space – цифровая среда моделирования боевых действий
disrupt – разрывать, разрушать
distinction – различение, распознавание; разграничение
distinguish – различать

E

electronic break-ins – электронные средства слежения
embedded – вложенный; встроенный
emergence – появление
emit – испускать, выделять
encapsulation – инкапсуляция
encode – кодировать, шифровать

encompass – заключать
endeavour – попытка, старание; стремление
end-to-end solution – комплексное решение
entity – суть, сущность
entity-relationship model – модель объект–отношение; модель сущность–связь
equilateral – равнобочный, равносторонний
erode – подрывать
esoteric – известный или понятный лишь посвящённым
eventually – в конечном счёте, в итоге, в конце концов
evidence – доказательство, основание
excitation – активизация, возбуждение
execute – осуществлять, выполнять
execution – выполнение, исполнение
expert system shell – экспертная система–оболочка
exploit – использовать в своих интересах
exploit – использовать, эксплуатировать

F

facilitate – содействовать, способствовать, облегчать
fake – ложный, фиктивный
fault – повреждение, ошибка
fertilize – обогащать, усиливать
fighting the information war – приемы ведения информационной войны
firewall – брандмауэр
firing rate – кпд [доля работающих нейронов] нейронной сети
flowchart – блок-схема, структурная схема
force multiplier – многократное усиление боеспособности
fraud – обман, мошенничество
friendly forces – союзники

G

game plan – план операции
grant – давать, предоставлять
GUI (graphical user interface) – графический интерфейс пользователя

H

hacking – неавторизованный доступ [к компьютерным данным], проникновение [в систему], взлом программ, "хакерство"
halting problem – проблема остановки
haphazardly – бессистемно; бесцельно; случайно
harm – вред, ущерб
harmless – безвредный, безобидный, невинный
hide – скрывать, прятать
hook – ловить, поймать
human agent – агент
hybrid language – комбинированный (составной) язык

I

immeasurable – неизмеримый, безмерный; большой, громадный
impact – влияние; воздействие
impact – воздействие, влияние
imperative language – императивный язык
implement – осуществлять, реализовывать
implementation – реализация, внедрение, ввод в действие, ввод в эксплуатацию
in the background – в фоновом режиме
inadvertently – ненамеренно, неумышленно
incorporate – содержать в себе
inference – [логический] вывод
inflict – наносить, причинять
information warfare – информационная война
inherent – присущий, свойственный
inhibition – торможение, подавление, угнетение
initialise – инициализировать например, задать начальное значение переменной
innocuous – безвредный, безобидный
innocuous – безвредный; безобидный, безопасный
insurmountable – неодолимый, непреодолимый, непобедимый, несокрушимый
integrity – целостность, сохранность
integrity – целостность, сохранность
intelligent battle system – интеллектуальная боевая система управления
intent – намерение, цель
interact – взаимодействовать
interdisciplinary – междисциплинарный
interrogator – допрашивающий
intervene – помешать, вмешиваться
introspection – самоанализ, самонаблюдение
investigation – расследование, следствие
invisible – невидимый
irrefutable – неопровержимый, неоспоримый, бесспорный, несомненный
isosceles – равнобедренный
iteration – повторение, шаг (в итеративном процессе), цикл

K

kernel – ядро
knowbot – программа глобального поиска

L

lack – отсутствие, недостаток
leaking – утечка, течь
learning curve – кривая обучения
legitimate – законный
leverage – средство для достижения цели
logical notation – логическое обозначение
lure – завлекать, приманивать

M

maintainability – удобство обслуживания, сопровождения
malicious – враждебный, злонамеренный
malicious – враждебный, злонамеренный
malicious – злоумышленный, злонамеренный
management system – интеллектуального
masquerade – выдавать себя за кого–л.; нелегально проникать
mental capacities – умственные способности
military planners – военные планирующие организации
misdirection – неправильный курс, направление
morsel – маленький кусочек (пищи)
multidimensional information – многомерная информационная
multiple – многочисленный
multithreading – многопотоковый режим, многопотоковость

N

neglect – пренебрегать
net applet – сетевое приложение
neuroscience – неврология
neurotransmitter – нейромедиатор
node – узел
novelty – новизна, инновация
numeric value – числовое значение

O

obstacle – помеха, преграда, препятствие
obstacle – помеха, преграда, препятствие
overflow – переполнение
overrun – перегрузка, переполнение

P

paste – вставлять
payload – оружие
perceive – воспринимать, понимать, осознавать; постигать
perceptual – относящийся к восприятию
permissiveness – вседозволенность
pervasive – всеобъемлющий, распространенный
pervasive computing – распределенные вычисления
phreaker – злоумышленник, взламывающий телефонные сети
portability – транспортабельность; мобильность, переносимость
potion – снадобье
predefined – предопределённый
preference – предпочтение, преимущество
premise – предположение
proactive intelligence – упреждающая система

procedural programming – процедурное программирование
program editor – программный редактор
program resident – резидентная часть программы
programming paradigm – принцип программирования, парадигма программирования
prone – склонный, предрасположенный
proof-reader – корректор
propagate – распространяться, передаваться
property – собственность
proprietary – частный, патентованный, оригинальный
pseudo code – псевдокод, символический код
publicize – разглашать; оглашать; извещать, сообщать, уведомлять
puzzle – вопрос, ставящий в тупик; головоломка, загадка

Q

quest – поиск(и)

R

randomly – случайно
ransfer function – функция преобразования
rationality – разумность, рациональность
readability – чёткость, разборчивость
reasoning – рассуждение, умозаключение
reasoning system – система, реализующая механизм рассуждений; разумная система; система, способная к рассуждениям
recovery – восстановление, исправление
rectangle – прямоугольник
regardless – безотносительно к чему-либо
regardless – независимо
reinforced – укрепленный, усиленный, утолщенный
reject – отказывать, отбрасывать
remote access – удалённый доступ
reroute – изменять маршрут
responsibility – ответственность, обязанность
reusable – повторно используемый; многократного пользования
robustness – робастность, устойчивость к нарушениям
rule-based system – система на основе продукционных правил
runtime error – ошибка во время выполнения

S

salami shaving – последовательное похищение небольших сумм со счетов при электронных банковских операциях
satellite communities – спутниковые системы
scam – афера, жульничество
scripting – написание сценария
seek to do smth. (sought, sought) – пытаться что-л. сделать
seemingly – по-видимому

showcase – выставлять, демонстрировать, показывать
simultaneously – одновременно
slip – проскользнуть, вкрасться
soft killers – способы уничтожения программного обеспечения
software strikes – активация атак с помощью ПО
solve – решать, разрешать (проблему); находить выход
source program исходная программа, программа на входном языке
spoof – обманывать
steal – красть
storehouse – хранилище
striking – поразительный, изумительный, необыкновенный; выдающийся, замечательный
subject matter – тема, предмет обсуждения
subset – подмножество
subvert – нарушать, разрушать
supersede – заменять; замещать, смещать
surge protector – устройство защиты от перенапряжений
susceptible – восприимчивый, поддающийся
syllogism – *лог.* силлогизм; тонкий, хитрый ход (для подтверждения или доказательства чего–л.)
syntax error – синтаксическая ошибка

T

terrestrial – земной
text editor – текстовый редактор
theft – воровство, кража
thief – вор
threshold – порог, пороговая величина
top notch – высшее качество
trade-off – выбор оптимального соотношения; компромиссное решение
trick into – обманом заставить что–л. сделать
triggered – пустить в ход, привести в движение
twofold – двукратный, удвоенный

U

UML (unified modeling language) – унифицированный язык моделирования
unauthorized access – несанкционированный доступ
unerring accuracy – точные координаты
unsuspecting – доверчивый, неподозревающий
utility – утилита, сервисная программа

V

VDE (visual development environment) визуальная среда разработки
victim – жертва
vigilant – бдительный

vigorously – решительно
violate – нарушать
visual perception – зрительное восприятие
vulnerable – уязвимый

W

walkthrough – сквозной контроль, разбор (анализ) программы
warfare – война
web site – веб–сайт
widespread – широко распространённый
withdraw – извлекать, изымать

Учебное издание

Составители:
Новик Диана Владимировна
Щербакова Александра Александровна

Сборник аутентичных текстов на английском языке

COMPUTER TECHNOLOGIES

для студентов дневной формы обучения по специальностям
1-36 04 02 «Промышленная электроника», 1-40 02 01 «Вычислительные машины, системы и сети», 1-40 03 01 «Искусственный интеллект», 1-53 01 02 «Автоматизированные системы обработки информации»

Ответственный за выпуск: Новик Д.В.
Редактор: Боровикова Е.А.
Компьютерная верстка: Боровикова Е.А.
Корректор: Новик Д.В., Щербакова А.А.

Подписано к печати 28.08.2014 г. Бумага «Снегурочка». Формат 60x84 1/16.
Гарнитура Arial Narrow. Усл. печ. л. 3,0. Уч. изд. л. 3,25.
Заказ № 529. Тираж 50 экз. Отпечатано на ризографе Учреждения образования
«Брестский государственный технический университет»
224017, г. Брест, ул. Московская, 267.