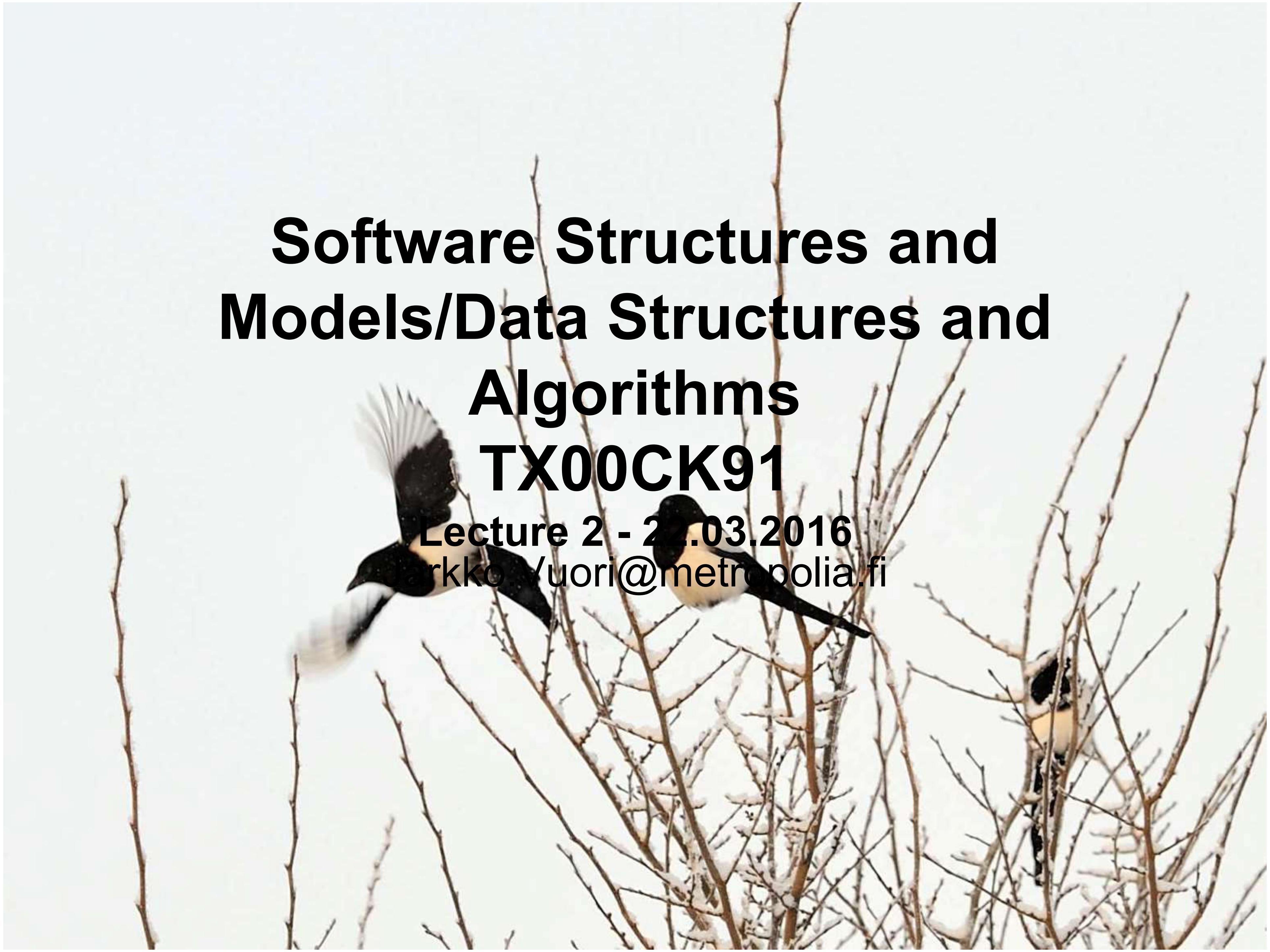


Software Structures and Models/Data Structures and Algorithms TX00CK91

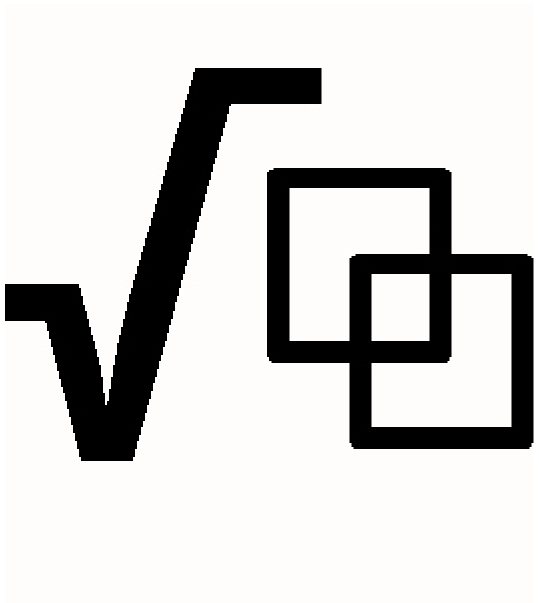
Lecture 2 - 22.03.2016

Jarkko.Vuori@metropolia.fi

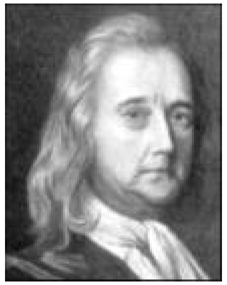
Using abstracted object

- Suppose you need to calculate a square root in your program
- What to do?
  1. Build your own square root function (time consuming, not very efficient)
  2. Use already defined abstract object, i.e. square root function sqrt()
- Note that the Java library function sqrt() has abstracted its internal implementation
  – You don't need to worry about it

```
int main() {
    double std_dev, variance;
    .
    .
    .
    // calculate the standard deviation
    std_dev = Math.sqrt(variance);
    .
    .
    .
}
```



Abstraction of the operations, square root $\sqrt{x}$


Isaac Newton (1643-1727)

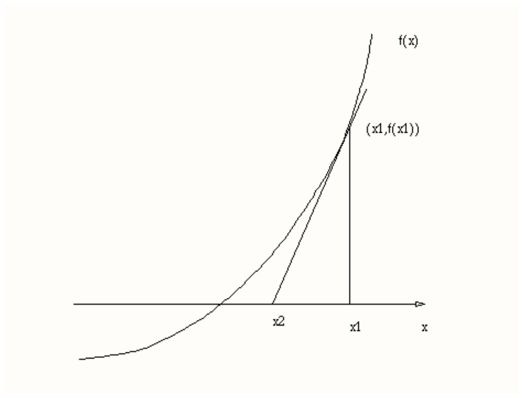
- Calculating square root using Newton's method

This equation is an abstraction for the more specific description presented on the right side $\longrightarrow$

$$x_{i+1} = \left(\frac{a}{x_i} + x_i \right) \Big/ 2, \quad i = 0,1,2,\ldots$$

where *a* is the argument and x_i *i*:th approximation for the root. Approximation x_{i+1} can be accepted as a root, if

$$\left| x_i^{\,2} - a \right| < \varepsilon$$



- Newton's method tries to find the zero point of the given function, f(x)=0, using derivative of the same function

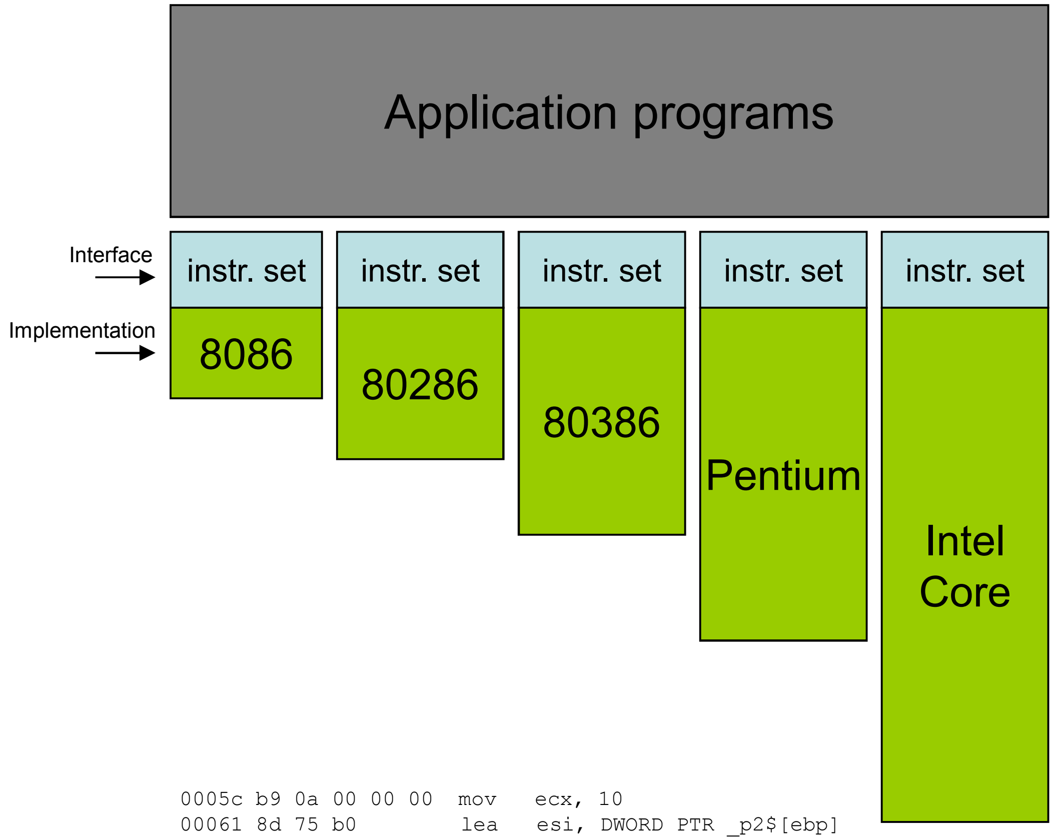
$$y = f(x)$$
$$y - f(x_0) = f'(x_0)(x - x_0)$$
$$\text{set } y = 0 \Rightarrow x_0 - \frac{f(x_0)}{f'(x_0)} = x$$

- For a square root

$$x = \sqrt{a} \Leftrightarrow x^2 = a$$
$$f(x) = x^2 - a \text{ we find the zero of this function}$$
$$\Rightarrow x = x_0 - \frac{x^2 - a}{2x} \Rightarrow x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n} \right)$$

Intel architecture is also an abstraction

- Intel’s 32/64-bit base architecture is called as Intel® 64 (Intel 64-bit Architecture)
  - Architecture means the logical structure of the processor and its instruction set
- Intel 64 architecture is based on 16-bit Intel 8086 –microprocessor which was introduced on 1978
  - 8086 was based on 8-bit 8080 microprocessor, introduced on 1974
  - For this reason, there are many inconsistencies in the instruction set (register binding, messy addressing modes, etc.)
  - On the other hand, all processors are upward compatible (e.g. programs made for 8086 will work intact in Intel Core processors)
- Instruction set can be seen as an abstraction of the processor structure
  - Internals of microprocessor has changed a lot during last 40 year, but the interface (instruction set) has been stayed the same

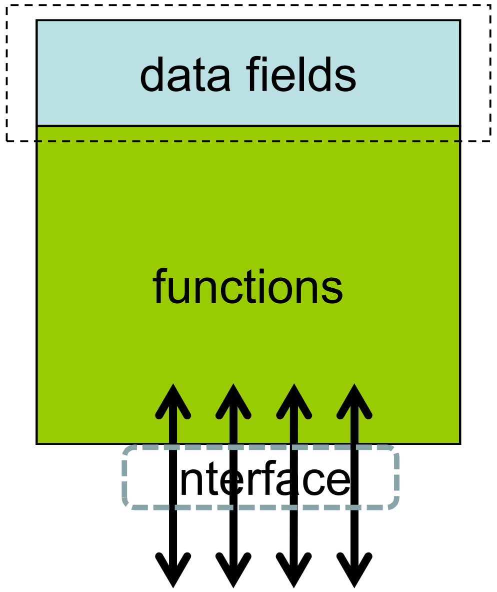


```
0005c b9 0a 00 00 00  mov   ecx, 10
00061 8d 75 b0         lea   esi, DWORD PTR _p2$[ebp]
00064 8d 7d d8         lea   edi, DWORD PTR _p1$[ebp]
00067 f3 a5            rep movsd
```

Instructions Assembler-code

Class

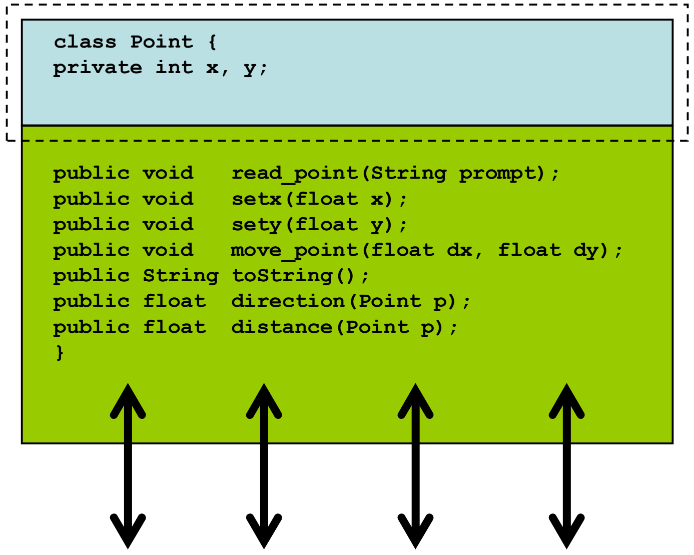
- Ttriangle (defined in the last lecture) datatypes and functions dedicated to handling it (toString, sides_equal, triangle_area) together form a class Triangle
  - Compared to function, class also includes data
- Definition: Class is a data type which contains all the data fields and functions (operation functions) needed to handle some specific task
  - We don't refer directly to the data fields of the class, but by using operation functions instead
    - Detail concealment (information hiding)
    - Operation functions are declared as function prototypes (in Java language) and data type and variable definitions (Class interface)
    - To the user, this interface is the name of the class
    - User (of the class) does not need to know anything about the internal structure of the class
      - Black box thinking



```
int      i;                   // int is a data type
Triangle t = new Triangle();  // Triangle is a class, but
                              // looks like a data type
```

Class

- Combining the datatypes, variables and its operations together forms the class
  - Operation of the program is easier to understand when we directly see what the program is doing (instead just how it is doing)
  - Program can be developed with step-by-step –method, and subparts can be tested separately
  - Higher-level parts of the program are isolated from the lower lever implementation details (lower level module can be changed without modifications to the higher-level)
  - Class can be used with other kind of applications than orienteering program



```
class Point {
private int x, y;

public void   read_point(String prompt);
public void   setx(float x);
public void   sety(float y);
public void   move_point(float dx, float dy);
public String toString();
public float  direction(Point p);
public float  distance(Point p);
}
```

```
void main(void) {
  Point p1 = new Point(),
        p2 = new Point();
  float  direction, distance;

  p1.read_point(); p2.read_point();

  distance  = p1.distance(p2);
  direction = p1.direction(p2);

  print_results(direction, distance);
}
```

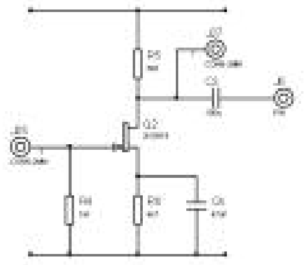
Class

- Class is like an instrument panel
  - Instrument panel is the user interface of the machine
  - Buttons, knobs, meters and lights are the interface to the complex system
    - Buttons and knobs are operation functions of the class
    - Lights and meters are value returning operation functions of the class
  - When you use those buttons you don’t need to know how the operation is exactly implemented, you only need to know the effect
  - With this kind of abstraction it is possible to master complex things relatively easy (like flying a helicopter)

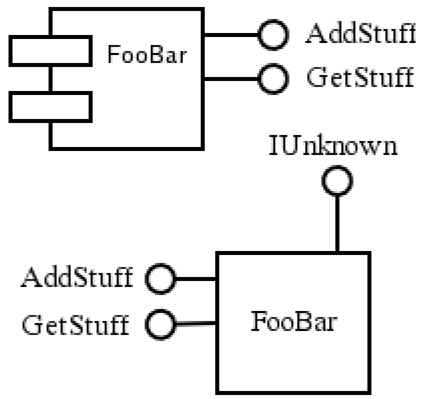
Bell 222 helicopter cockpit

Software component

- Class is a Software Component
  - It is generic in a sense that we can use similar component in different kinds of applications, i.e. component can be used in many kinds of applications without modifying the component
    - Point can be used as such in ”orientation” application, graphic object calculation applications (triangles, circles,…), game applications (position on the screen) or in route calculation example
    - Component must be well enough designed, it must contain enough operation functions
  - We can plug a new implementation of a component (a better or otherwise different) in our application without modifying the application itself (providing that interface specification is identical)
    - We need no modification to main function (or more generally upper levels in program module hierarchy) when point representation is changed into polar coordinates (e.g. due to the efficiency requirements)
    - This requires that we obey good programming habits, i.e. no references directly to the internal datastructures
  - We can use it with other components (only ”interface” is needed)
  - Encapsulated (Component is compact, black box)
    - Program is subdivided, components and the main part of the program is compiled separately (easier to change a component)
  - Can be further developed independently
- Class Point obeys all those requirements, it is a software component



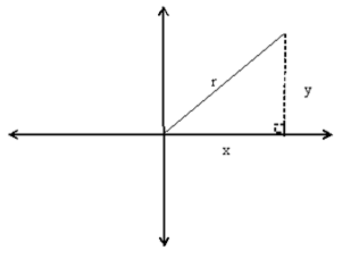
Drawing symbols of electronic components



Upper picture is UML symbol lower picture is Microsoft COM symbol of the software component

How to build a reusable software component

1. The operation functions should be well designed and we should have a good selection of them
2. Separate the interface and the implementation, compile the implementation
3. We have to obey the rule of good programming practice in writing the application
   - Local variables should be kept private (point.x is not allowed in applications)
   - Only operation functions are used to access data type/variables

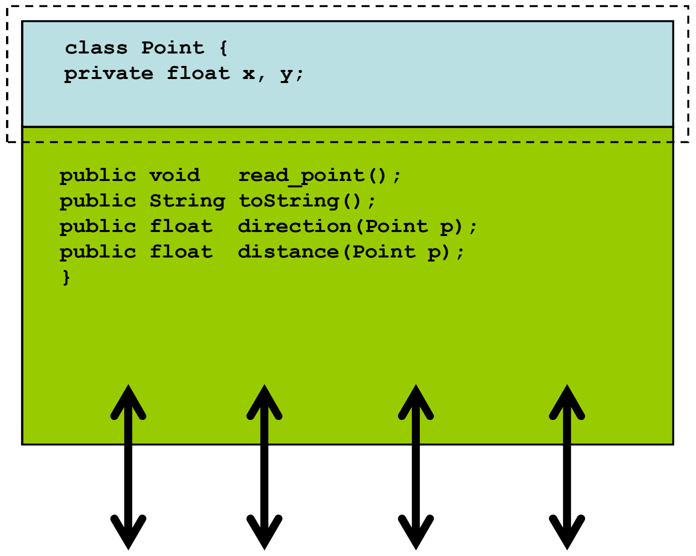


Class point, method 1

Rectangular coordinates

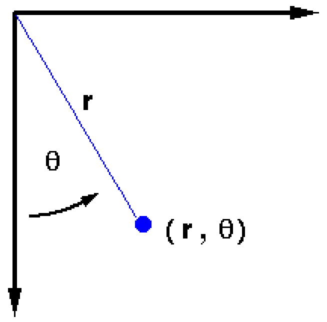
```
public void read_point() {
    Scanner in = new Scanner(System.in);
    x = in.nextDouble();
    y = in.nextDouble();
}

public String toString() {
  return x + "," + y;
}

public float distance(Point p) {
  return (Math.sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
}

public direction(Point p) {
  return (Math.atan(p.y-y, p.x-x));
}
```

Interface



```
class Point {
private float x, y;

public void   read_point();
public String toString();
public float  direction(Point p);
public float  distance(Point p);
}
```

```
void main(void) {
  Point p1 = new Point(),
        p2 = new Point();
  float  direction, distance;

  p1.read_point(); p2.read_point();

  distance  = p1.distance(p2);
  direction = p1.direction(p2);

  print_results(direction, distance);
}
```

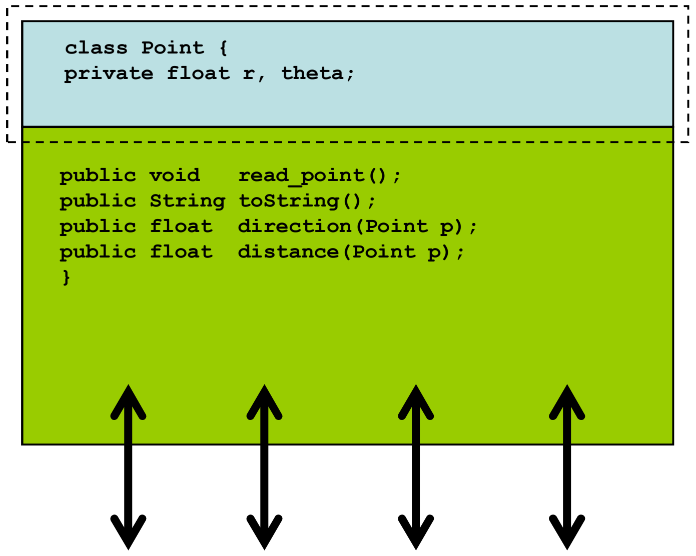


Class point, method 2

Polar coordinates

```
public void read_point() {
    Scanner in = new Scanner(System.in);
    x = in.nextDouble();
    y = in.nextDouble();
}

public String toString() {
  return x + "," + y;
}

public float distance(Point p) {
  return (Math.sqrt(r*r+p.r*p.r-2*r*p.r*Math.cos(p.theta-theta)));
}

public float direction(Point p) {
  return (3.1415/2 - theta - p.theta);
}
```

Interface



```
class Point {
private float r, theta;

public void   read_point();
public String toString();
public float  direction(Point p);
public float  distance(Point p);
}
```

```
void main(void) {
  Point p1 = new Point(),
        p2 = new Point();
  float  direction, distance;

  p1.read_point(); p2.read_point();

  distance  = p1.distance(p2);
  direction = p1.direction(p2);

  print_results(direction, distance);
}
```

Component programming

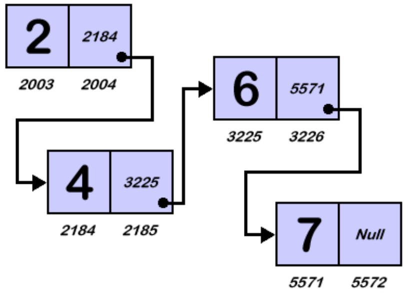
- Building programs using preprogrammed components is not a new idea
  - The idea was first presented by Douglas McIlroy in NATO's software technology conference in Germany 1968
    - Conference was created to answer the demands of software crisis
    - First results of "component thinking" were pipes and filters in Unix operating system
- The basic idea of component programming is to produce new application by combining ready made software modules
  - Object oriented programming stresses also reusability, but in addition the natural modeling of objects (e.g. using descriptive verbs and nouns when naming objects, fields and methods)
- Designing reusable component is quite demanding, component must be
  - Well documented
  - Thoroughly tested
  - Robust with the input parameters
  - Give sensible errors if needed
  - Build to stand unpredicted use
- Programming work in the future will be mainly just combining ready-made components together and parameterizing those components
  - It is quite easy to make modifications to the application (build in this way)
    - Specifying those modifications required needs more work that the actual modification
    - This kind of fine tuning of the application cannot be outsourced, because outsourcing need always good specifications
- Well known component programming environments
  - VBX, OCX/ActiveX/COM and DCOM (Microsoft)
  - XPCOM (Mozilla Foundation)
  - VCL ja CLX (Borland)
  - Enterprise Java Beans (Sun Microsystems)
  - UNO (OpenOffice.org office suite)
  - Eiffel ja Oberon programming languages

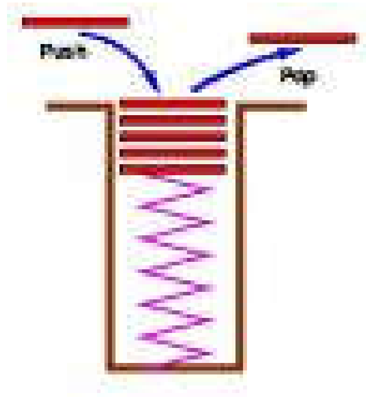
Some benefits using class

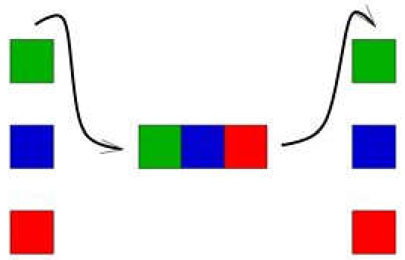
1. Operation functions can be grouped together with data fields (= class definition)
2. Good programming habits can be enforced by using keyword `private`
3. Operation functions for different objects could have the same name, e.g. initialize()
4. Operators can be overloaded (easier use of operators)
5. Function calls are more like natural languages, e.g. if (matti.older(pekka))
6. Initialization of objects can be done with constructors
7. We can use inheritance, it is possible to alter the properties of a software component (even without the source code)
8. Give it possible to implement abstraction/specialization relationship
9. Implements polyformism (e.g. array may have elements with different types)

Universal abstract datatypes

- List (lista), stack (pino) and queue (jono) are abstract datatypes which have universal usage
  - They (or very similar structures) are used in nearly every program
    - user list, queue between two processes, stack in the processor, etc.
  - In software world they are counterparts of hardware components
    - Same components can be used as building blocks in many different equipments
- Programmer's one task is to find general purpose (or universal) solutions to the problems
  - Because we don't want to solve and solve again the same problems
  - General purpose, abstract datatypes are one solution







Copying the code

- Our goal is to have the maximum amount of reusable components available to us when we are building applications
  - Language structures and keywords are lower level building blocks in software science
    - You don’t need to reimplement them every time you use them (they are already implemented in the compiler by the compiler’s designer)
  - If all the applications were done from the beginning, we have to recreate parts that have already been done
    - This means that we are doing the same thing again and again …
    - The simplest method to avoid this is to copy parts already done to the new application
      - Quite difficult if this has not taken in mind when those software units were created (variable names must be changed, parameter passing could be a problem, …)

Using subroutines (functions)

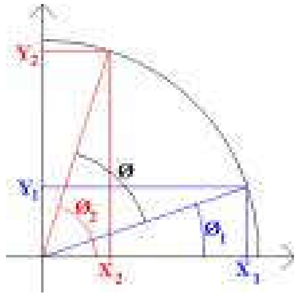
- The next step, when trying to reuse code, is to divide the program to multiple subroutines (or functions)
  - And these functions can be reused
    - Parameters of functions are central concept of this reusing
    - You can call the same function in different places with different variables in function call (= reusing the function in many places)
  - Also new applications can then reuse these simple functions
    - Square root calculation, calculating the distance between two points
  - Benefit (compared to simple code copying) is that source code of functions doesn't need to be taken into the source code of the new application
    - Already compiled functions can be linked later to this new application
  - Functions can be later switched to better ones (more rapid execution, less memory usage, etc.)

    And this does not influence the source code of the application (if the prototype of the function remains the same)
      - We can switch the Newton's iteration to the CORDIC algorithm when calculating the square root

```
// calculates the median of the given arguments
float median(float x1, float x2, float x3) {
    float largest, smallest;

    largest  = max(x1, x2);
    smallest = min(x1, x2);

    // find median
    if      (x3 < smallest) return smallest;
    else if (x3 > largest)  return largest;
    else                    return x3;
}
```

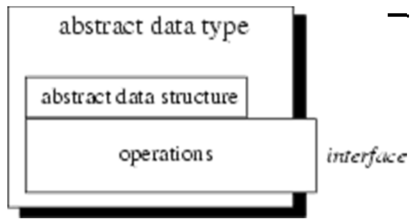
```
        .
        .
        .
t = median(s[n], s[n-1], s[n-2]);
        .
        .
        .



        .
        .
        .
k = median(previous, current, future);
        .
        .
        .
```

We can call the same function with different parameters



Using abstract datatype

- We group all variables (which belongs together) to form a single datatype, and all operations to this datatype are programmed to functions
  - This group is called an Abstract Data Type (ADT)
    - It is a mathematical abstraction
  - Abstraction level for reusability raises (compared to simple subroutine reusage), because the ADT method includes also the data (with operations, e.g. functions)
  - Modifications are easier to made because we can change the (internal) data representation (as functions) also
  - E.g., it is possible to change the algorithm to use double floating point values to improve the accuracy of the algorithm without any modifications to the application
    - Interface should be the same (name of the datatype and prototypes of the operation functions) with this modified ADT
      - So that we don't need any modification to the application that uses this ADT



Using of classes (OO languages)

- The Java class allows for the implementation of ADTs, with appropriate hiding of implementation details
- In addition we have other concepts
  - Inheritance
    - We can add new properties to the class without any modification to the original class
    - Represent specializing in one sense

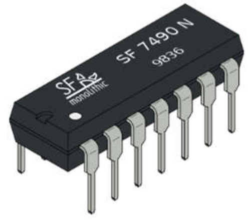
General purpose components

1. Application specific ADTs

2. General purpose ADTs (components)
   - Counter (laskurit)
   - Containers (säiliöt)
     - We can **store** data elements to them
     - Containers are general purpose ADTs
     - In principle, an array is an simple container because we can store multiple value to it (e.g. multiple measurement values)

- Some containers
  - List
  - Stack
  - Queue
  - Trees
    - Binary tree
    - Binary search tree
    - B+ -tree
    - Etc.
  - Hash table (jakautustaulukko)

```
class Counter {
private:
  int value;
public:
  void clr();
  void inc();
  void dec();
  void set(int value);
  int  read();
}
```



Introduction to generics

- In order to fulfill container generality requirement 3, we need generics
- Generics are based on the feature that type can be a kind of parameter in the function definition or class definition
- If only one or more types of function parameters or class data members differentiates functions or classes, we can write only one definition for it
  - This definition is not a real function or class definition. It is function or class generics
- The compiler can, using generics, generate (instantiate) real functions or real classes when they are needed
  - the compiler does type checking for generics function parameters and for member functions of generics class
- The generic definition is called method generic or class generic
- The method or class instantiated from generic is called generic method or generic class
- Generics save programmer's work, because he does not need to write many almost similar functions or classes

Method Generic

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (<E> in the next example)
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char)

```
public class GenericMethodTest {
   // generic method printArray
   public static <E> void printArray(E[] inputArray) {
      // Display array elements
      for (E element : inputArray){
          System.out.printf("%s ", element);
      }
      System.out.println();
    }

    public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println("Array integerArray contains:");
        printArray(intArray); // pass an Integer array

        System.out.println("\nArray doubleArray contains:");
        printArray(doubleArray); // pass a Double array

        System.out.println("\nArray characterArray contains:");
        printArray(charArray); // pass a Character array
    }
}
```

Bounded type parameter

- In order to restrict the kinds of types that are allowed to be passed to a type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound
- Comparable interface requires compareTo() method to be implemented

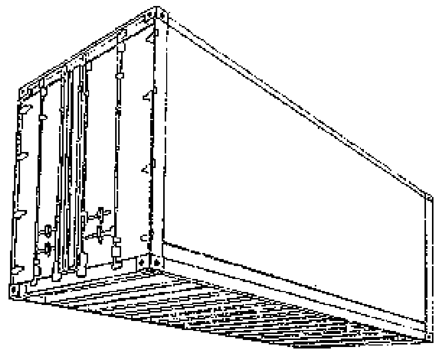
```
public class MaximumTest {
   // determines the largest of three Comparable objects
   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x; // assume x is initially the largest
      if (y.compareTo(max) > 0){
         max = y; // y is the largest so far
      }
      if (z.compareTo(max) > 0){
         max = z; // z is the largest now
      }
      return max; // returns the largest object
   }
   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
                   3, 4, 5, maximum(3, 4, 5));

      System.out.printf("Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
                   6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

Generic classes

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section
- The type parameter section of a generic class can have one or more type parameters separated by commas
- These classes are known as parameterized classes or parameterized types because they accept one or more parameters

```
public class Box<T> {
    private T t;
    public void add(T t) {
       this.t = t;
    }
    public T get() {
       return t;
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();
        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));
        System.out.printf("Integer Value :%d\n\n", integerBox.get());
        System.out.printf("String Value :%s\n", stringBox.get());
     }
 }
```

Container



- Container is a simple data storage
- Generality requirements for the container are
  1. We can reuse it in new applications (even in new application areas)
  2. Application which uses the container is independent from the implementation of the container
  3. Implementation of the container is independent of the type of the element to be stored (new requirement)
- Because container is an ADT it consist of
  - Data type (for example Container)
  - Operation functions (like insert_item, delete_item, …)
- It is operations that make a container unique (a certain type of container)

Linear list, definition 1

- Linear list is a container
- Linear list is a collection of elements. The following operations (between elements) are defined for the linear list
  1. initialize list
  2. test if empty
  3. find the length of the list
  4. retrieve *i*:th element (hae listasta *i*:s alkio)
  5. determine whether or not a given item is in list
  6. store a new value to the list
  7. delete an item at position *i*
- Linear sequence can be ordered by
  – entering order
  – ascending
  – descending
  – alphabetical
  – Some other order
- Because the order of the elements is defined, the first and the last element can be defined formally

Linear list, its usage

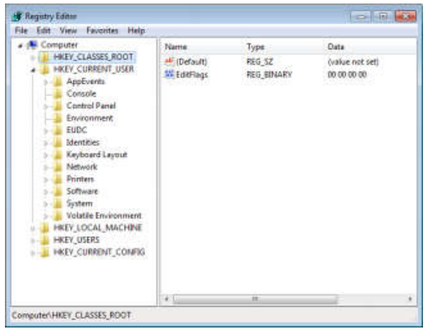
- We require that a List is an ADT, and implement it using a class
- First we declare class variable
  - Actual values are stored here
- Then we use class operation functions
  - Initialization of those internal data structures in a class variable are done automatically using a constructor

```
// declare a list object
List list = new List();

// use operation functions to use it
list.insert_to_list_end('a');
```

List items are stored there
in the list variable, but we
don't know how – it is abstracted out (using private fields)

Linear list, where it can be used

- Ranking list for a competition (order by the result)
- Results of an exam (alphabetical order)
- Bus timetable (list ordered by a bus line number)
- Phone book
- Component list (ordered by component type and identifier)
- Windows Registry



```
Computer
  HKEY_CLASSES_ROOT
  HKEY_CURRENT_USER
    AppEvents
    Console
    Control Panel
    Environment
    EUDC
    Identities
    Keyboard Layout
    Network
    Printers
    Software
    System
    Volatile Environment
  HKEY_LOCAL_MACHINE
  HKEY_USERS
  HKEY_CURRENT_CONFIG
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
EditFlags	REG_BINARY	00 00 00 00

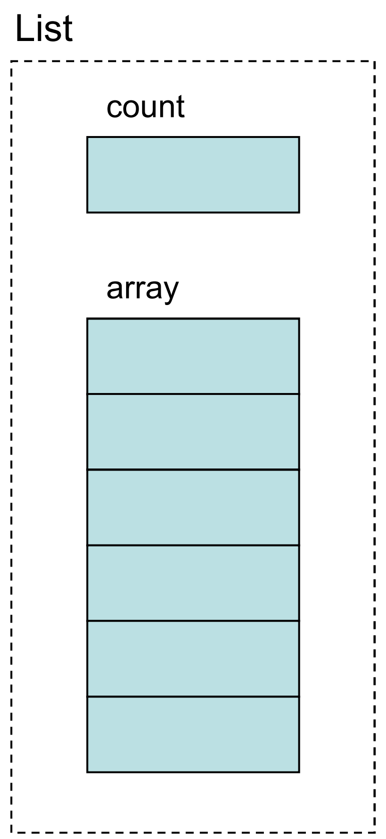
Computer\HKEY_CLASSES_ROOT

Linear list, an implementation

- Abstract datatype can be implemented in different ways, so that the current implementation does not affect applications which use it
- Linear list can be implemented in many different ways
  - Implementations vary in how the data is represented in the memory of computer (ns. tallennusrakenne)
  - Linear list must somehow represent the elements and their order
  - Most used implementation methods are
    - Table
    - Linked list
    - Dynamically linked list
    - Also variations of the previous methods are possible

Linear list, an array

- Elements of the list are represented as an ordinary array
- Order is defined by the location of the element in that array
  - Space for the elements is reserved element by element in a sequence
- In addition we need to implement also some details that give us variations
  - We should determine how the number of elements (current number of elements, and the maximum number of elements) is represented
  - Array can be a static or reside in a memory area reserved from dynamic memory
  - Operation functions are totally different depending the criterion of ordering (order of entering or order by elements value)
- In all those implementations, variables belonging to the list must be grouped together



Linear list, an array

```
public class simpleList<T extends Comparable<T>> {
    private static final int MAX = 10;
    private int count;
    private T[] array;

    private void compress(T[] array, int empty_slot, int n) {
        for (int i = empty_slot + 1; i < n; i++)
            array[i - 1] = array[i];
    }

    public simpleList() {
        count = 0;
        array = (T[])new Comparable[MAX];
    }

    public boolean empty() {
        return count == 0;
    }

    public int size() {
        return count;
    }

    public T at(int i) {
        return array[i];
    }
```

```
public boolean insert_to_end(T item) {
        if (count < MAX) {
            array[count++] = item;
            return true;
        } else
            return false;
    }

    public int find_pos(T item) {
        for (int i = 0; i < count; i++)
            if (array[i].compareTo(item) == 0)
                return i;

        return -1;
    }

    public boolean del(int orderNo) {
        if (orderNo >= 0 && orderNo < count) {
            compress(array, orderNo, count);
            count--;
            return true;
        }
        else
            return false;
    }
```

Linear list, an array

```
public static void main(String[] args) {
        simpleList<Character> list = new simpleList();
        Character             item;
        Scanner               s = new Scanner(System.in);
        int                   i = 0;

        try {
            System.out.print("Enter items? ");
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('.')) != 0) {
                list.insert_to_end(item);
                item = new Character(s.next().charAt(0));
            }
            //Print the contents of the list
            for (i = 1; i <= list.size(); i++)
                System.out.print("\n " + i + "th item was " + list.at(i-1));
            System.out.print("\nDelete list items ?");
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('.')) != 0) {
                i = list.find_pos(item);
                if (i >= 0) {
                    System.out.print("\nThe position of the item in list is " + i);
                    list.del(i);
                } else
                    System.out.print("\nItem not found");
                    item = new Character(s.next().charAt(0));
                }
             //Print the contents of the list
             for (i = 1; i <= list.size(); i++)
                 System.out.print("\n " + i + "th item was " + list.at(i-1));
             System.out.println();
        } catch(Exception e) {

        }
    }
}
```

Linear list, an array

- List implementation could be divided to parts
  - Interface and implementation of ADT list
  - (Test) Application
- Notice that the list interface and implementation must reside in the same file (because of compiler restrictions)
- In the example we used character list
- The genericity requirement now means that it should be easy to change the item to be stored in container
  - This can be easily done because we used templates

```
simpleList<Character> charList;
simpleList<Float>     floatList;
simpleList<Time>      timeList;
```

Linear list, an array

- The proposed implementation of the linear list fulfills all requirements (three requirements) for the container
  - Implementation of the list in source code level does not need to change, even if the type of the element is changed from char to person (Person)
  - There is no need to change the application (when the element type is changed), if the operation logic of the application remains the same
  - There is no need to modify the application in the case where the implementation of the list is changed (e.g. from array to linked list)
- In order to fulfill the generality requirement, elements must be processed in application level and in ADT by using element operation functions only and container must be handled in application level using container functions only
  - Abstraction layers should not be crossed
- In our list implementation, we have also ”private” functions (e.g. compress)
  - These are for the internal (implementation’s own use) use only

Very important!