

Object Oriented Programming

Why ?

- New software is usually object-oriented.
- The software is written using an abstraction called an object.
- Much more to commercial software development
- Simply writing lines of code
- There is investigation of the business requirements, analysis of the problem, design of the solution, and so on.
- Objects should be used at every stage of the development because they reduce the amount of information that has to be understood and improve the communication between members of the development team.

HISTORY OF PROGRAMMING

Commercial programming has had a number of generations, of which 'object-oriented' is just the latest:

- Machine code: Programming using binary numbers.
- Assembly language: Programming using alphanumeric symbols, or **mnemonics**, as **shorthand** for machine code. Assembly language is translated into machine code by a program called an **assembler**

HISTORY OF PROGRAMMING

- High-level languages: Programming using languages (such as Fortran and COBOL) that have high-level constructs such as types, functions, loops and branches. High-level languages (and later generations of programming languages) are translated into machine code using a program called a **compiler**.
- Structured programming: Programming using cleaner high-level languages (such as Pascal, Modula and Ada) that are characterized by fewer pitfalls for the programmer and more discipline in the way a program is broken down into sub-tasks and sub-systems.

HISTORY OF PROGRAMMING

- Object-oriented programming: Programming using **independent modules** of data and functions that **correspond to concepts** in the problem domain, such as Customer or ScrollBar. This modularity leads to even fewer pitfalls for the programmer and encourages the **reuse of code** across separate programs. Good object-oriented programming languages include Java and Eiffel, because they're well designed, pure and portable (available on many platforms).

Structured Programming

- Structured programming takes on the top-to-bottom approach.
- Structured programming is based around data structures and subroutines.

Structured Programming

- For example, invoice printers use structured programming. This type has clear, correct, precise descriptions.
- A structured program is decomposed into a hierarchy of processes. A process in this context is a body of
- code, typically a function or subroutine, that takes some input and manipulates it to produce an output. A process may be composed of other, more specialized processes, i.e., it may be a function that calls other functions.

OOP

- This type of programming uses sections in a program to perform certain tasks.
- It splits the program into objects that can be reused into other programs.
- They are small programs that can be used in other software.
- It splits the tasks into modular forms. This makes the program simpler and easier to read with less lines and codes.

OOP

- Each object or module has the data and the instruction of what to do with the data in it. This can be reused in other software .
- An object-oriented program is decomposed into a network of collaborating objects. An object represents a thing or concept and has a known set of behaviors that may be invoked by other objects. For any activity of the program, an object responsible for that activity may interact with other objects by invoking their behaviors, or "methods", until the activity is complete.

Advantage OOP

- Objects are easier for people to understand: This is because the objects are derived from the business that we're trying to automate, rather than being influenced too early by computer-based procedures or data storage requirements. For example, in a bank system, we program in terms of bank accounts, bank tellers and customers, instead of diving straight into account records, deposit and withdrawal procedures, and loan qualification algorithms.

Advantage OOP

- Specialists can communicate better: Over time, the software industry has constructed career ladders that newcomers are expected to climb gradually as their knowledge and experience increases. Typically, the first rung is **programmer**: fixing faults (bugs) in the code written by others. The second rung is **senior programmer**: writing the code itself. The third is **designer**: deciding what code needs to be written. Finally comes the role of **analyst**: talking to customers to discover what they need and then writing down a specification of what the finished system must be able to do.

Advantage OOP

- Data and processes are not artificially separated: In traditional methods, the data that needs to be stored is separated early on from the algorithms that operate on that data and they are then developed independently. This can result in the data being in inconvenient formats or inconvenient locations, with respect to the processes that need access. With object oriented development, data and processes are kept together in small, easy-to-manage packages; data is never separated from the algorithms. We also end up with less complex code that is less sensitive to changes in customer requirements.

Advantage OOP

- Code can be reused more easily: With the traditional approach, we start with the problem that needs to be solved and allow that problem to drive the entire development. We end up with a monolithic solution to today's problem. But tomorrow always brings a different problem to solve; no matter how close the new problem is to the last one we dealt with, we're unlikely to be able to break open our monolithic system and make it fit – we hamper ourselves by allowing a single problem to influence every part of the system.

Advantage OOP

- Object orientation is mature and well proven: This is not a new fad. The programming concepts emerged in the late 1960s while the methodologies have been around for at least a decade. Applying objects in such areas as software, databases and networks is now well understood.

UML

The notation used for illustrations, wherever possible, is the **Unified Modeling Language (UML)** . This has become the accepted **standard for software diagrams**. UML presentation conventions mean that some lines are thicker than others and that some characters are in bold or italics.

OBJECT CONCEPTS

- An **object** is a thing, an entity, a noun, something you can pick up or kick, anything **you** can imagine that has its own identity. Some objects are living, some aren't. Examples from the real world include a car, a person, a house, a table, a dog, a pot plant, a check book or a raincoat.

All objects have **attributes**: for example, a car has a manufacturer, a model number, a color and a price; a dog has a breed, an age, a color and a favorite toy. Objects also have **behavior**: a car can move from one place to another and a dog can bark.

- In object-oriented software, real world objects migrate into the code. In programming terms, our objects become stand-alone modules with their own knowledge and behavior (or, if you prefer, their own data and processes). It's common to think of a software object as a robot, an animal, or a little person: each object has certain knowledge, in the form of attributes, and it knows how to perform certain **operations for the benefit of the rest of the** program.

- For example, a person object might know its title, first name, last name, date of birth and address; it would be able to change its name, move to a new address, tell us how old it is, and so on.

Objects in the real world



aPerson



aNumber



aBankAccount



aDate



aCat

Here's how one looks in Java:

```
new Person("Sue Smith")
```

The effect of this expression is to create space for a new Person object and pass it the string “Sue Smith”, so that it can initialize itself (presumably, in this case, the object would record its name).

Once we've created an object, we can put it somewhere where we can find it later, by **assigning a name to it, as in:**

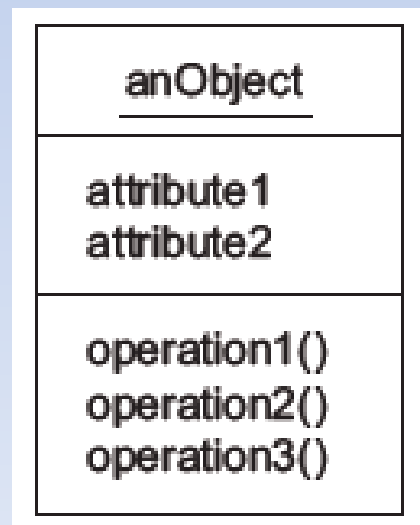
```
aPerson = new Person("Sue Smith");
```

Now, whenever we write down aPerson, we will be referring to the object that we just created.

DEPICTING OBJECTS

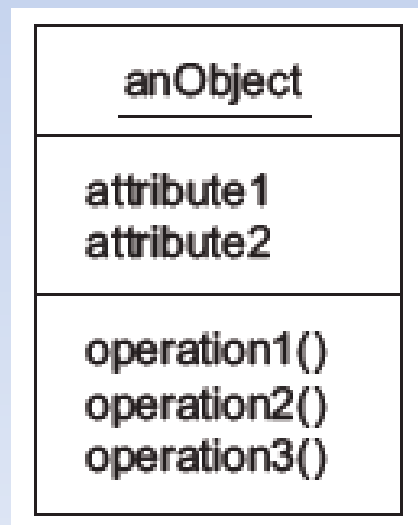
we need some way of showing them on a diagram so that we can describe them and think about them

The notation used here is a UML **object diagram**



The three parts of the box show

- the object's name (which is underlined)
- its attributes (its knowledge)
- its operations (its behavior)



aCoffeeMachine

drinkPrices
availableDrinks
drinkRecipes

displayDrinks()
selectDrink()
dispenseDrink()
acceptMoney()

- `public int add(int x, int y) {
 return x + y;
}` ---→ Operation
- `public String getName() {
 return this.name;
}` -→ Attribute

ENCAPSULATION

Encapsulation refers to an object hiding its attributes behind its operations (it seals the attributes in a capsule, with operations on the edge). Hidden attributes are said to be private.

As an example of why encapsulation is a good idea, consider an object representing a circle. A circle would be likely to have operations allowing us to discover its radius, diameter, area and perimeter. What attributes would we need to store in order to support this behavior?

- we could store the radius and calculate the other attributes on demand.
- we could store the diameter and calculate the other attributes from that.

Let's say we choose to store the diameter. Any programmer who was allowed to access the diameter might do so, rather than going via the 'get diameter' operation. If, for a later version of our software, we decided that we wanted to store the radius instead, we would have to find all the pieces of code in the system that used direct access to the diameter, so that we could correct them (and we might introduce faults along the way). With encapsulation, there's no problem.

```
/* File name : EncapTest.java */  
public class EncapTest{  
  
    private String name;  
    private String idNum;  
    private int age;  
  
    public int getAge(){  
        return age;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public String getIdNum(){  
        return idNum;  
    }  
}
```

```
public void setAge( int newAge){  
    age = newAge;  
}
```

```
public void setName(String newName){  
    name = newName;  
}
```

```
public void setIdNum( String newId){  
    idNum = newId;  
}  
}
```

The variables of the EncapTest class can be access as below:

```
/* File name : RunEncap.java */  
public class RunEncap{  
    public static void main(String args[])  
    {  
        EncapTest encap = new EncapTest();  
        encap.setName("James");  
        encap.setAge(20);  
        encap.setIdNum("12343ms");  
        System.out.print("Name : " + encap.getName()+  
        " Age : "+ encap.getAge());    }}
```

Name : James Age : 20

Thank You
Terima Kasih