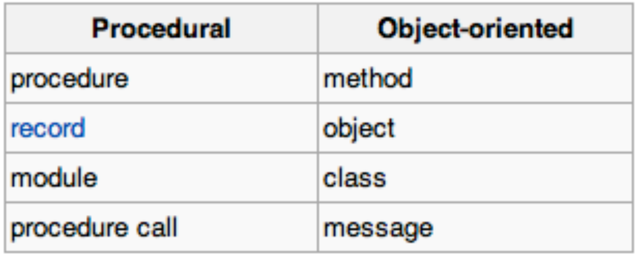
Object-Oriented Programming Design and Analysis

Elena Punskaya, op205@cam.ac.uk

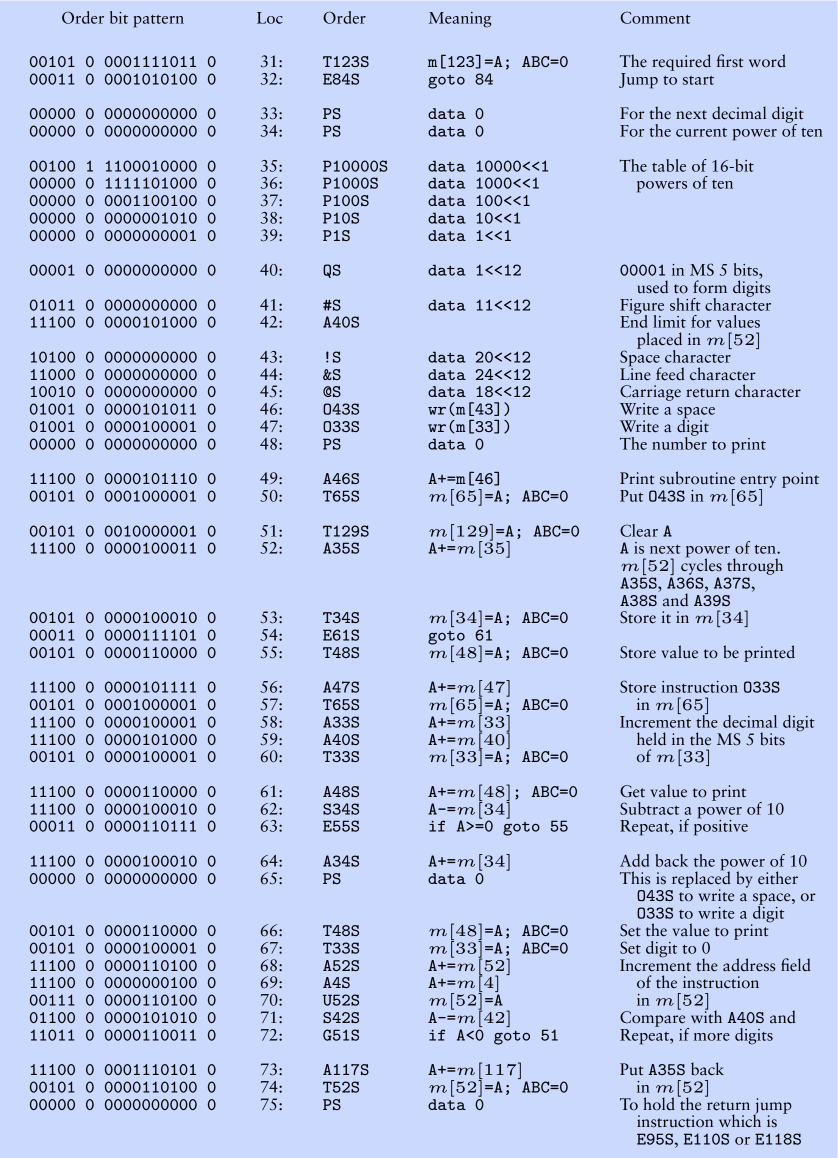
Object-Oriented Programming

- **First programs: “anything goes”**
- **1960s-1970s: structured programming**
  - any “computable function” can be achieved by
    - “sequencing” (ordered statements)
    - “selection” (conditions, e.g. if/else) and
    - “repetition” (iteration, e.g. for i=0, i<10, i++)
  - "Go To Statement Considered Harmful", Dijkstra
  - Procedural programming: modularity
- **1967 – “objects” first formal appearance in Simula’67**
- **1980 – Smalltalk-80: “everything is an object” (including primitive data types such as integers)**



Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

http://en.wikipedia.org/wiki/Procedural_programming_languages



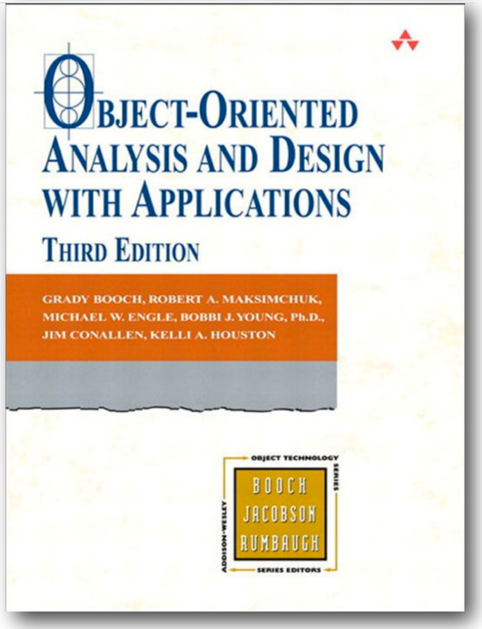
```
   Order bit pattern      Loc    Order      Meaning              Comment

00101 0 0001111011 0      31:    T123S      m[123]=A; ABC=0      The required first word
00011 0 0001010100 0      32:    E84S       goto 84              Jump to start

00000 0 0000000000 0      33:    PS         data 0               For the next decimal digit
00000 0 0000000000 0      34:    PS         data 0               For the current power of ten

00100 1 1100010000 0      35:    P10000S    data 10000<<1        The table of 16-bit
00000 0 1111101000 0      36:    P1000S     data 1000<<1           powers of ten
00000 0 0001100100 0      37:    P100S      data 100<<1
00000 0 0000001010 0      38:    P10S       data 10<<1
00000 0 0000000001 0      39:    P1S        data 1<<1

00001 0 0000000000 0      40:    QS         data 1<<12           00001 in MS 5 bits,
                                                                   used to form digits
01011 0 0000000000 0      41:    #S         data 11<<12          Figure shift character
11100 0 0000101000 0      42:    A40S                            End limit for values
                                                                   placed in m[52]
10100 0 0000000000 0      43:    !S         data 20<<12          Space character
11000 0 0000000000 0      44:    &S         data 24<<12          Line feed character
10010 0 0000000000 0      45:    @S         data 18<<12          Carriage return character
01001 0 0000101011 0      46:    O43S       wr(m[43])            Write a space
01001 0 0000100001 0      47:    O33S       wr(m[33])            Write a digit
00000 0 0000000000 0      48:    PS         data 0               The number to print

11100 0 0000101110 0      49:    A46S       A+=m[46]             Print subroutine entry point
00101 0 0001000001 0      50:    T65S       m[65]=A; ABC=0       Put O43S in m[65]

00101 0 0010000001 0      51:    T129S      m[129]=A; ABC=0      Clear A
11100 0 0000100011 0      52:    A35S       A+=m[35]             A is next power of ten.
                                                                 m[52] cycles through
                                                                 A35S, A36S, A37S,
                                                                 A38S and A39S
00101 0 0000100010 0      53:    T34S       m[34]=A; ABC=0       Store it in m[34]
00011 0 0000111101 0      54:    E61S       goto 61
00101 0 0000110000 0      55:    T48S       m[48]=A; ABC=0       Store value to be printed

11100 0 0000101111 0      56:    A47S       A+=m[47]             Store instruction O33S
00101 0 0001000001 0      57:    T65S       m[65]=A; ABC=0         in m[65]
11100 0 0000100001 0      58:    A33S       A+=m[33]             Increment the decimal digit
11100 0 0000101000 0      59:    A40S       A+=m[40]               held in the MS 5 bits
00101 0 0000100001 0      60:    T33S       m[33]=A; ABC=0         of m[33]

11100 0 0000110000 0      61:    A48S       A+=m[48]; ABC=0      Get value to print
11100 0 0000100010 0      62:    S34S       A-=m[34]             Subtract a power of 10
00011 0 0000110111 0      63:    E55S       if A>=0 goto 55      Repeat, if positive

11100 0 0000100010 0      64:    A34S       A+=m[34]             Add back the power of 10
00000 0 0000000000 0      65:    PS         data 0               This is replaced by either
                                                                   O43S to write a space, or
                                                                   O33S to write a digit
00101 0 0000110000 0      66:    T48S       m[48]=A; ABC=0       Set the value to print
00101 0 0000100001 0      67:    T33S       m[33]=A; ABC=0       Set digit to 0
11100 0 0000110100 0      68:    A52S       A+=m[52]             Increment the address field
11100 0 0000000100 0      69:    A4S        A+=m[4]                of the instruction
00111 0 0000110100 0      70:    U52S       m[52]=A                in m[52]
01100 0 0000101010 0      71:    S42S       A-=m[42]             Compare with A40S and
11011 0 0000110011 0      72:    G51S       if A<0 goto 51       Repeat, if more digits

11100 0 0001110101 0      73:    A117S      A+=m[117]            Put A35S back
00101 0 0000110100 0      74:    T52S       m[52]=A; ABC=0         in m[52]
00000 0 0000000000 0      75:    PS         data 0               To hold the return jump
                                                                   instruction which is
                                                                   E95S, E110S or E118S
```

The Squares Program (excerpt) written for EDSAC, 1949

http://www.cl.cam.ac.uk/~mr10/edsacposter.pdf

Object-Oriented Analysis and Design

> *In the early days of object technology, many people were initially* ***introduced to "OO" through programming*** *languages. They discovered what these new languages could do for them and tried to practically apply the languages to solve real-world problems. As time passed, languages improved, development techniques evolved, best practices emerged, and formal object-oriented methodologies were created..*
>
> *Object-Oriented Analysis and Design with Applications, Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, Kelli A. Houston*



OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS

THIRD EDITION

GRADY BOOCH, ROBERT A. MAKSIMCHUK, MICHAEL W. ENGLE, BOBBI J. YOUNG, Ph.D., JIM CONALLEN, KELLI A. HOUSTON

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

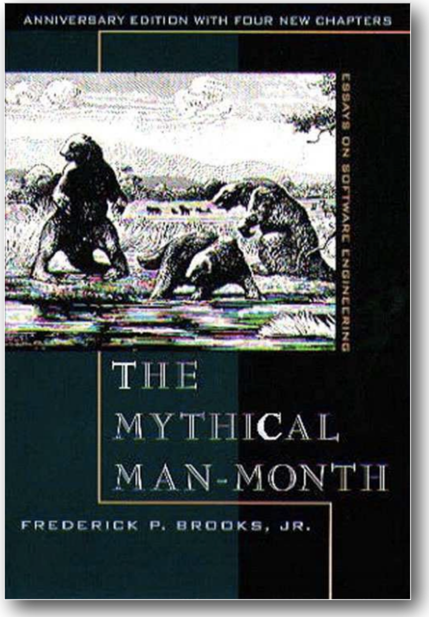
SERIES EDITORS

ADDISON-WESLEY

- **Object-Oriented Programming**
  - "*Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects ...*"
- **Object-Oriented Design**
  - "*Object-oriented design is a method of design encompassing the process of object- oriented decomposition ...*" – using object and classes to describe the system
- **Object-Oriented Analysis**
  - *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*

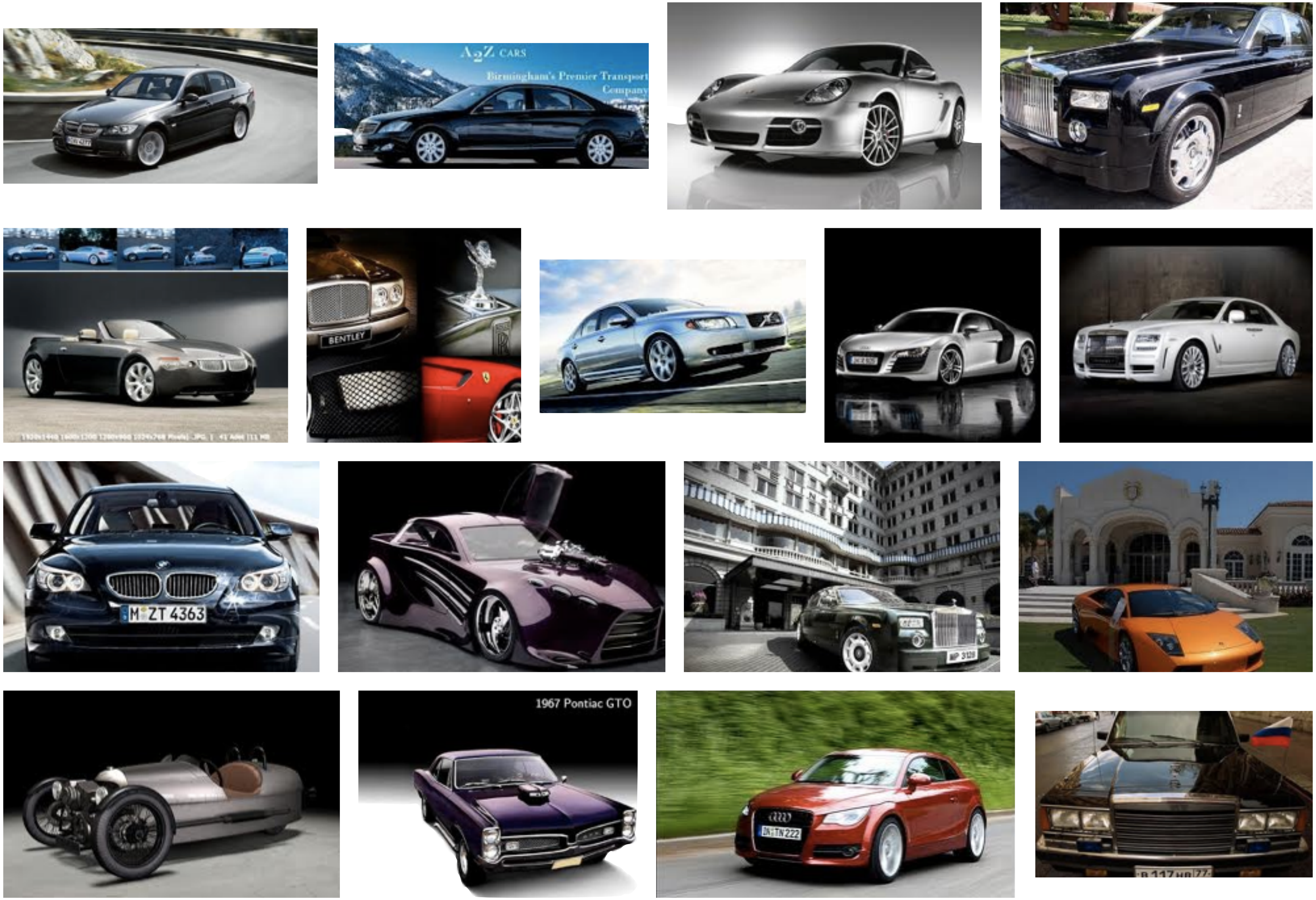
Abstractions in Software

> *Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction.*
> *The reality of software is not inherently embedded in space. Hence, it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics.*
>
> *Fred Brooks, "No Silver Bullet – Essence and Accident in Software Engineering"*



- **Object Oriented approach allows to use a meaningful abstraction in software design**
  - Looking at a design diagram anyone can have a good guess what the Circle class represents in the Drawing Editor or the Customer class in the Online Banking system
- **Essentially, Object Orientation is the way of describing a system in terms of meaningful abstractions (classes), relationship and interactions between them**
- **Can be used on both Conceptual and Implementation levels**

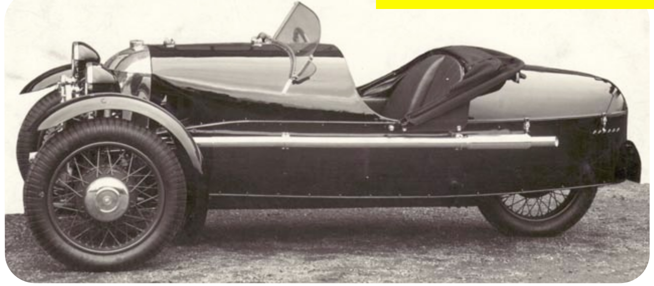
What are these?



Abstractions

- **Cars, of course!**
- **However, a Fiat Punto and a Buggatti Veron are very different objects, yet we can call both a car and be correct**
- **Abstractions are generalisations that define certain key characteristics and behaviour**
- **All cars have**
  - 4 wheels, at least a driver seat, an engine
- **All cars can**
  - move, stop, steer

there are always exceptions ...



www.morgan3wheeler.co.uk

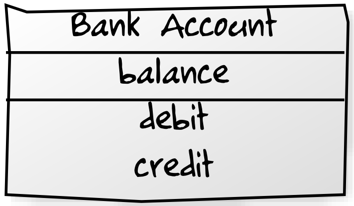
- **All birds have two wings and can fly, all cameras have a lens and can take pictures ...**

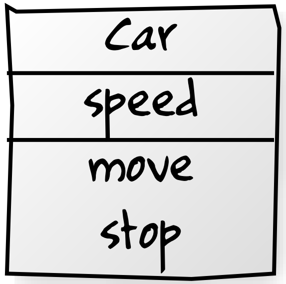
> *Classification is the means whereby we order knowledge. In object- oriented design, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms and eventually leads us to smaller applications and simpler architectures.*
>
> *Object-Oriented Analysis and Design with Applications, Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, Kelli A. Houston*

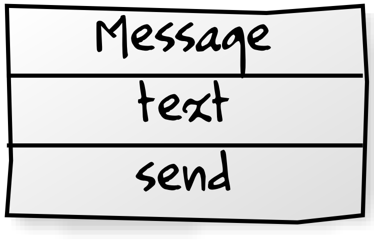
Class

- **A class represents a key concept within the system**
- **It encapsulates data and behaviour**
- **Object-oriented approach requires thinking about:**
    - What classes of objects are present in the problem?
    - What behaviour does each class have to provide?
    - What should happen when an action is requested of an object?

Concept	Data	Behaviour
bank account	balance	debit/credit
car	speed	move/stop
message	text	send

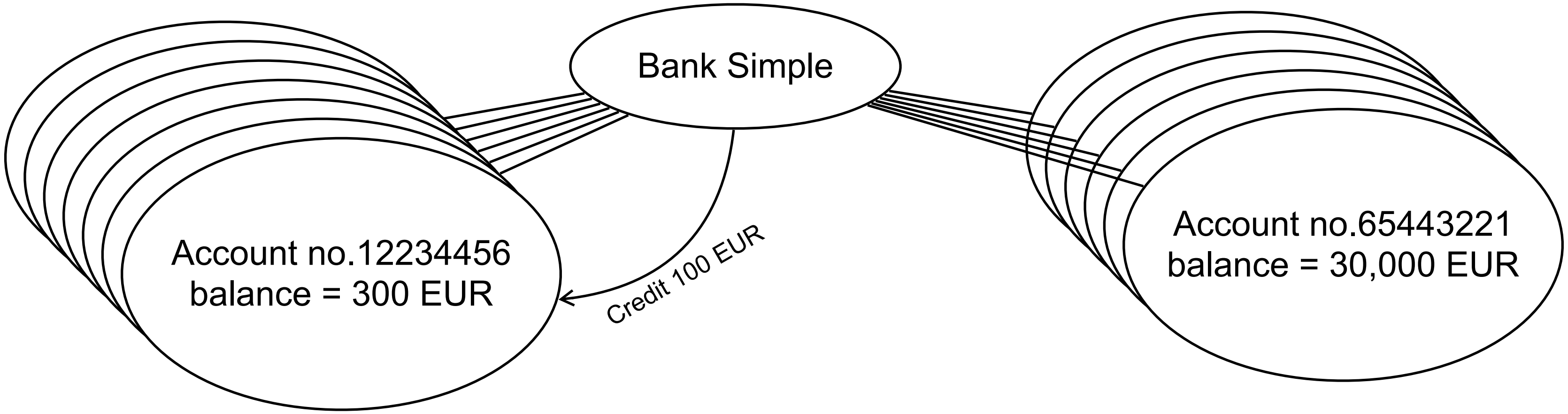






Object

- Each class can be used to create multiple *instances*, i.e. *objects* that can contain data and behave according to the class definition
- Bank account no.*12234456* vs Bank account no.*65443221*
- *Credit* 100 EUR, *Debit* 200 EUR
- Given the same data (state) two independent instances of the same class will behave exactly the same: e.g. *crediting* 100 EUR will increase the *current balance* by 100 EUR
- In production (when the software is used) a large number of objects are created, interact with each other, destroyed if no longer needed



Encapsulation

- Classes provide *abstractions*. An object can be used without any knowledge of how it works. This allows to describe the system in manageable concepts

- All drivers know that a steering wheel makes the car go left if we turn it left and right if we turn it right
- Most drivers have no idea why/how it works
- By exposing only WHAT it can do and not HOW, a designer can later improve the steering wheel without changing how the driver interacts with it,
  - e.g. adding power steering or adding play music controls
- In Object Orientation this approach is called Encapsulation or Information/Data Hiding and is complimentary to Abstraction

Data Hiding

- **Suppose class Person has a Name attribute, which defines a full name**
- **One way to access the Name would be to declare a direct access to the attribute (variable)**

```
class Person
{
    public String FIRST_NAME = “John”
    public String LAST_NAME = “Smith”
}

print Person.FIRST_NAME + Person.LAST_NAME
```

- **Consider a Change request: we sold the software in France and now need to display the Last Name first**

```
if (in France)
    print Person.LAST_NAME +
Person.FIRST_NAME
else
    print Person.FIRST_NAME +
Person.LAST_NAME
```

- **We can extend our Person class to have two variable and let the display code decide the order**
- **OR we can keep Name variables hidden from the outside and allow to access via a method**

```
class Person
{
    private String FIRST_NAME = “John”
    private String LAST_NAME = “Smith”

    public String Name
    {
        if (in France)
            return Person.LAST_NAME +
Person.FIRST_NAME
        else
            return Person.FIRST_NAME +
Person.LAST_NAME
    }
}

print Person.NAME
```

- **Compare the change request:**
  - in the original case we would need to modify code in ALL cases where the Name is shown
  - in the latter case we only need to make a change to the implementation of the method

Encapsulation and Data Hiding

- **Getters/Setters (i.e. Properties) are used to provide controlled access to internal data fields**

```
class Person
{
    private String email;
    public String getEmail { return email;
}
    public setEmail(String newEmail)
    {
      if ((newEmail != null) &&
          (newEmail.contains(‘@’))
      {
         email = newEmail;
      }
}
```

- **They allow to implement constraints checking, e.g. should not be *null* or should contain be formatted as an email address**
- **Control concurrent access**
- **Hide actual data sources (e.g. database)**

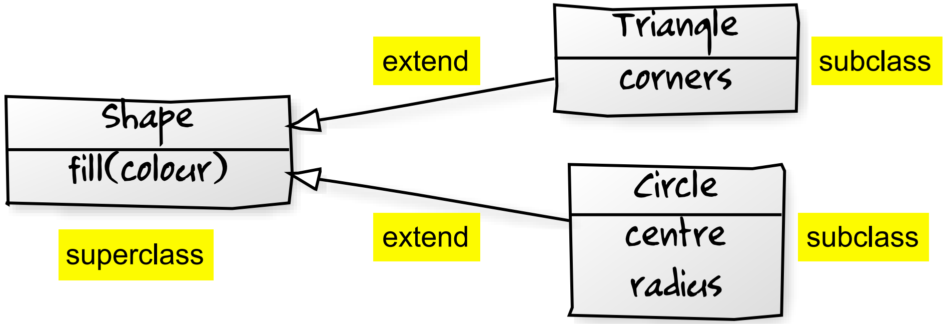
> *Startup style development? "We could have getters and setters, but... Life's too short"*
>
> *a Cambridge software startup, 2012*

> *"You only need to floss the teeth you want to keep"*
>
> *an old saying*

> *“the point of encapsulation isn't really about hiding the data, but in **hiding design decisions**, particularly in areas where those decisions may have to change. The internal data representation is one example of this, but not the only one and not always the best one. The protocol used to communicate with an external data store is a good example of encapsulation - one that's more about the messages to that store than it is about any data representation.”*
>
> *Martin Fowler, http://martinfowler.com/bliki/GetterEradicator.html*

Inheritance

- **Classes can be related**
  - *Superclass* AKA *base class* (parent) can be *extended/inherited from*
  - *Subclass* (child) can be *extending/deriving/inheriting from*
- **Each Subclass can hold all the data and perform all the actions of the Superclass**
- **Subclass, however, can also**
  - hold additional data
  - perform new actions
  - and/or perform original actions differently
- **Example:**
  - to draw a circle requires to know a point of origin and the radius
  - to draw a triangle requires us know know coordinates of its vertexes
  - **both** can have a fill colour



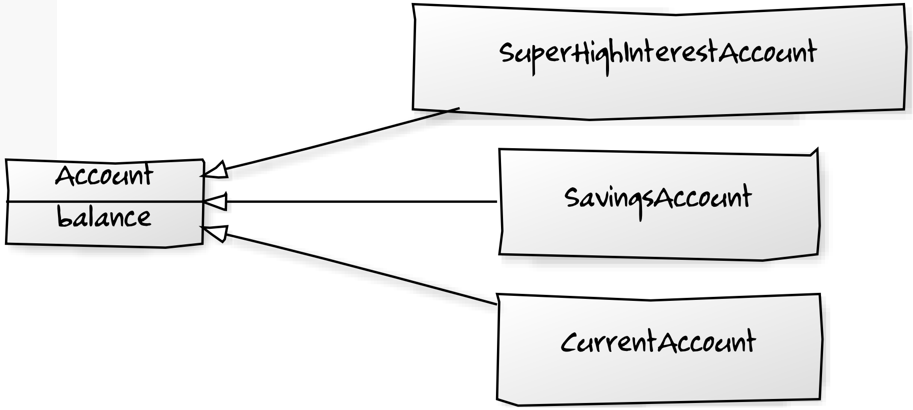
Polymorphism

- **Class inheritance hierarchy allows us to choose the level of abstraction at which we interact with an object:**
  - to calculate how much money is in the bank in total, the system needs only one piece of data - the *current balance*, every *account* will have the balance and for these purposed there is no difference if it’s a *savings* account or a *current* account
  - however, the way the balance calculated could be very different (including the rules about interest)
  - Polymorphism allows us to request the same action from objects yet allow for it to be executed in different ways
- **Why is it useful? Extensibility! Compare**

```
foreach CurrentAccount in Bank
{
    TotalMoney = TotalMoney + CurrentAccount.balance()
}
foreach SavingsAccount in Bank
{
    TotalMoney = TotalMoney + SavingsAccount.balance()
}
foreach SuperHighInterestAccount in Bank
{
    TotalMoney = TotalMoney +
SuperHighInterestAccount.balance()
}
```

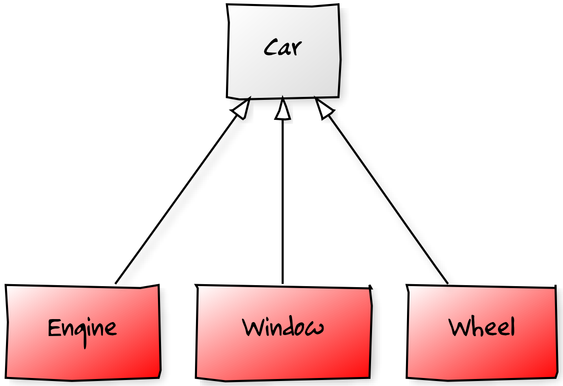
- **to**

```
foreach Account in Bank
{
    TotalMoney = TotalMoney + Account.balance()
}
```

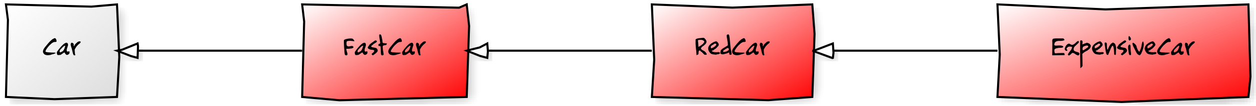


Polymorphism and Inheritance

- **Polymorphism separates the declaration of the functionality from specifics of its implementation**
- **Polymorphism is one of the key concept of Object Orientation**
- **Requires a principally different view on the system**
- **Identifying good key Classes and Inheritance Hierarchy is not simple**



Engine, Window, Wheel are NOT Cars

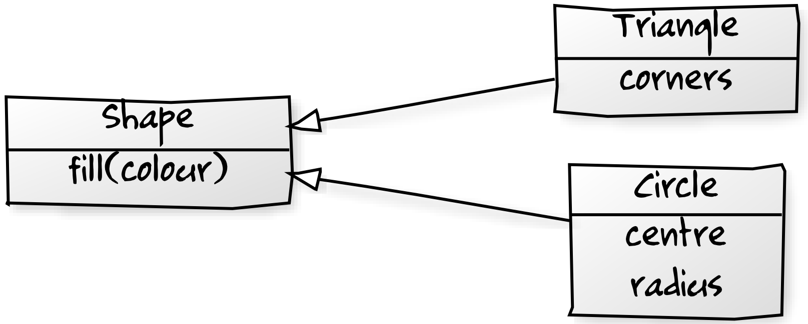


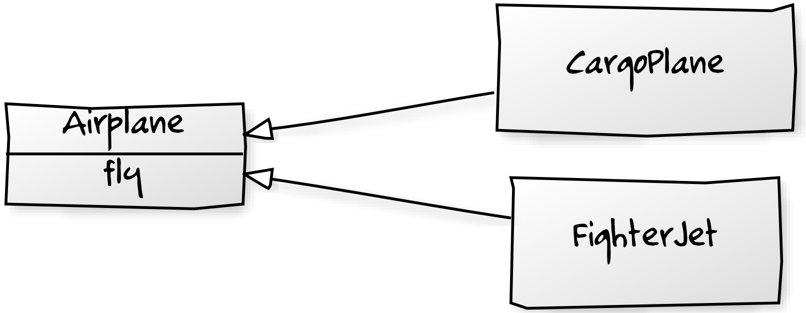
A new quality/feature does NOT equal a new class
Furthermore, Fast, Red, Expensive are Values not Attributes, where the Attributes could actually be Speed, Colour, Price

- **Inheritance “IS A” relationship – Subclass “IS A” Superclass**
  - Coffee IS a Drink, Car IS a Vehicle

Abstract Classes

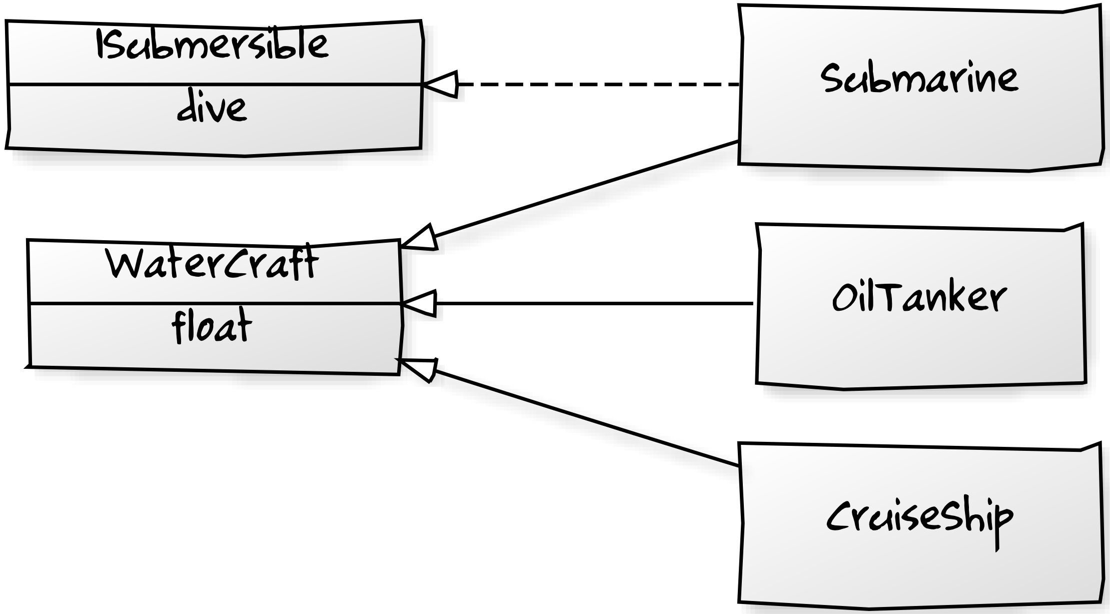
- In our previous example, we used Triangle and Circle classes that extend the Shape class
- In our system, Shape is a conceptual class, i.e. we will never have an actual object which is just a Shape, we will have either a Circle or Triangle
- This makes Shape class Abstract – a class that can not be instantiated
- Abstract classes are a high level “blueprints” for Objects in the system, but to actually make Objects we would need some “concrete” classes
- Abstract classes capture the higher level view of the system





Interfaces

- A purely abstract class that defines only behaviour (not data) is called an Interface
- All WaterCrafts can float on water, but only a Submarine can go under water
- Interfaces help to add specific behaviour to classes
- Typically, a class can only extend one superclass but it can and often will "implement" multiple interfaces



Object Orientation Summary

- **Object Oriented approach allows us to understand the requirements and design a solution on the conceptual level**
- **It allows us to design and build extensible solutions, addressing the key challenge of software engineering – building for change**
- **To achieve this it offers us encapsulation, inheritance, polymorphism**
- **Terminology**
  - Object is an Instance of a Class
  - Class, Subclass, Superclass
  - Inheritance, Polymorphism
  - Abstract Classes, Interfaces
- **It allows us to communicate ideas and concepts in a clear consistent way to all team members**
- **It works across all stages of the software development process from Analysis to Maintenance via Design and Implementation**