



# Object Orientated Programming

Programming Skills Lecture

If you have a laptop with you, please go to:

<http://tinyurl.com/pvabdpm>

BEFORE the lecture starts.

(leave that window open for later...)

**Thanks!**

- What we will cover today
- An illustrating example
- Object Orientated
  - Objects, Classes, Instances
  - Inheritance, Composition, Polymorphism
- When do you use it



- Not everyone agrees on the precise definition of OOP
- It allows representation of “real” things
- Quick Example: we can talk about books in an abstract way
  - this becomes a class, when we talk about a specific book it becomes an instance(or object).
- Common Characteristics
  - Objects – Classes or Prototypes
  - Instances – Runtime creations of objects
  - Encapsulation
  - Inheritance
  - Composition
  - Polymorphism

- Object Orientated Programming is awkward for the first time
- Using an example helps
- Our Example – A Deck of Cards and Hands of Cards
- How could we represent this?



# A Simple Approach

- Array to represent the cards (Value and Suit)?

A	2	3	4	5	6	7	8	9	10	J	Q	K	A	2	3	4	5	6	7	8	9	10	J	Q
C	C	C	C	C	C	C	C	C	C	C	C	C	D	D	D	D	D	D	D	D	D	D	D	D

- How to track what is available? – another array?
- How do you interact reliably?
  - Does every piece of code do it the same way?
- Could get quite complicated
- What are the problems?

- How would you represent a hand?
- An array?
- How do you deal with increasing and decreasing?
- Do you sort it?

- ADTs were introduced previously which mainly deal with defining data types in terms of operations
- We are going to talk about OO in terms of classes and instances but this is not the only way
- OO takes this in a complimentary direction
  - Data and Operations (Methods) are brought together
  - Classes can have common methods which operate at a class level
  - Methods can be specific to an instance
  - Attributes (values) can be defined at a class level to be shared, local to an instance, or made a shared constant
  - Inheritance, Composition, Delegation, Polymorphism



- A Class can be thought of as a Blueprint which:
  - Defines the data held by a class
  - The methods to access and manipulate data
  - Methods which have significance to the class
  - A Class: Book could define attributes and methods
    - Attributes: title, author, pages
    - Methods: getPageCount, updateAuthorName
- An Object is a runtime creation with values
  - An Object is generally an instance of a Class
  - Can use anything in the definition of the class
  - An instance of a book could have the values:
    - title:Mort, author:Terry Pratchett, pages:320

- Attributes are how data is stored in an object, these are defined by the class
- Attributes can be primitive types or other classes
  - This is what is called composition – where one class is made up of other classes rather than replicate existing type behaviours
- In the prior example of books, the attributes would be:
  - Title – a string
  - Author – a string or possibly a class Author or Person
  - Pages – a number

- So lets go back to our exercise example of:
  - Cards
  - Deck
  - Hands
- What Classes do we have?
- What would be an object?
- For each class, come up with the set of attributes for that class

- Card
  - Suit – String (Taken from a Constant) or Enumeration
  - Value – String/Integer/Enumeration
- Hand
  - Cards – Array of Card
  - MaxSize – Maximum Number of Cards
  - CurrentCards – Current Number of Cards
- Deck
  - Cards – Array of Card
  - Deck Size – Integer
  - Cards Left – Integer
- What Improvements could be made?



- In ADTs, it was operations, in OO its often referred to as methods
- Types of Method
  - Constructor/Destructor
  - Instance Methods
  - Class Methods – Available from the Class itself not instances

- What Methods does each class need?
  - Card
  - Hand
  - Deck
- Remember Constructor, Access to data, alteration and other processes which need to be run on the data

- Card
  - Constructor - Suit and Value
  - GetSuit
  - GetValue
  - GetStringRepresentation
  - Destructor
- Deck
  - Constructor - Suits, Values, DeckSize
  - GetDeckSize
  - GetRemainingDeckSize
  - GetSuits
  - GetValues
  - AddCard – Card
  - DealCard
  - Shuffle
  - GetStringRepresentation
  - Destructor
- Hand
  - Constructor – Hand Size, Max Hand Size
  - AddCard – Card
  - RemoveCard – Card
  - RemoveRandom – Card
  - Sort
  - IsMaximumSize
  - GetCount
  - IsCardPresent – Card
  - DiscardAll
  - Destructor
  - GetStringRepresentation
  - ChangeMaxSize

- We now have classes with attributes and methods
  - Are they well organised?
  - Is there anything missing?
- Point out what you need to add in here
- An optional extra in many languages is to use enumerations for known set of values
  - These can be separately defined or embedded in classes



- ADTs did encapsulation and so does OO
- Often we don't want third-parties dealing directly with data
- Levels of Visibility of Classes, Attributes and Methods
- Commonly found levels
  - Private – only the class instances can see this attribute or method
  - Public – anything can access and change
  - Protected (called different things in different languages) – only descendants in the same class hierarchy or package structure
  - Different languages do things in slightly different ways

- We've likely just used composition – decks and hands
- Composition is making use of other classes to provide data and operations – Commonly defined as “x HAS A y”
- Inheritance is where a class extends explicitly another classes – it ‘inherits’ all the functions of the parent but can add to and override its methods and attributes – Commonly defined as “x IS A y”
- Composition allows use of other classes without needing to expose everything about them
- Inheritance allows you to expose directly behaviours from the parent

- The Card Class is important in our example
- Other classes use this.
- A Hand is made up of Cards as is the Deck
- So we use the Class Card in the definition of the classes Hand and Deck
- Is this the limit of composition in this example?
- How does Suit and Value possibly factor into this?

- Are there any classes which share functionality?
- Any which share common underlying data structures?
- Should one descend from the other or maybe should there be a parent class for both?
- Another way to do this is with Interfaces (sometimes known as Abstract Type with no data or implementation acting as a contract to define what needs to be provided)



- The Deck and Hand share many common features
  - Both hold N cards
  - Both have 'sort' methods
  - Both need to be able to add and remove cards
- There are some differences
  - Hands are more variable – from size and maximum size
  - Any others?
    - Sort methods?
- Maybe introduce a card holder class as a parent class and inherit from it

- Define a class CardHolder
- Attributes
  - HeldCards - List of Cards
  - Size – Number of Cards
- Methods
  - Add Cards
  - Remove Cards
  - Sort
- Deck and Hand can inherit these common functions and data structures from this class and override and add methods
- Create your version of this based on the example
- What are the different methods and data structures?

- Different Types of polymorphism
  - overloading (ad hoc operator polymorphism)
  - Subtyping (descendant class methods can be used if present)
  - Parametric (Generics)
- Overloading is the most commonly encountered
  - Simple operators like addition or subtraction are classic example
  - Multiple implementations defined for a single operator
  - Decision of implementation determined by types (runtime or compiler)

- OO is potentially powerful
- Suffers from a lack of common understanding
- Temptation to make things too complicated
- Use it when it makes sense to use it
  - Does your problem fit the OO approach?
  - Will it help with maintenance and extension?
  - Do your developers follow what is happening?
- Use it when there is loose “coupling” between major “things” in your design and those “things” are highly internally cohesive



- Object Orientated Programming
- Complementary to ADTs
- Found in a range of languages: C++, Java, Python, Smalltalk
- Don't view everything as a nail – its not always the best thing to use