COMP1002
DATA STRUCTURES AND ALGORITHMS

LECTURE 3: STACKS, QUEUES AND OBJECTS

Curtin University
Curtin Computing

Last updated: 9 March 2020

Copyright Warning

Objectives

- Revise Object orientation
- Introduce Abstract Data Types
- Provide first examples of ADTs – Stacks and Queues
- Discuss applications of Stacks/Queues incl. Equation Solving
  - Postfix evaluation
  - Infix to postfix conversion

OBJECT ORIENTATION

Revision slides from OOPD/FOP

Object-Orientation

- In object-oriented programming, we bundle the behaviour (methods) and data (attributes) together
- Benefits:
  - OO protects data from being used incorrectly
  - Increases code reuse (fewer errors)
  - Makes code easier to read and maintain
  - Objects "know" how to respond to requests
  - Relates to how objects function in the real world

Reference: Module 4 of Object-oriented Program Design, Curtin University, 2017

Classes – Specifying Objects

- Before we can use an object, we need to describe it as a **class** (of objects).
  - Similar to how we define a function once and use it multiple times
- The class specifies the state and behaviour an object can have:
  - State: what the object is
    - attributes or member fields
  - Behaviour: what the object does
    - methods or functions

Encapsulation

- A (an object of a) class makes use of the "information hiding" principle
  - Communication with the rest of the software system is clearly defined
    - methods are the means for communication
  - Its obligations to the software system are clearly defined
    - what services the class offers (via data and methods)
  - Implementation details should be hidden from the user
    - don't need to know how it does things to use it

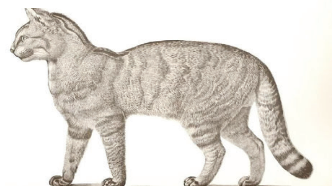
Class Specification

- Must include:
  - Details of the communication with the rest of the software system (method names)
  - The exact data representation required
  - Exactly how the required functionality is to be achieved (method implementation)

Classes and Objects

- An object is an **instance of** a class
- The class definition provides a template for an object
- An object gives details for a particular instance

Generic cat = class "cat"



http://s460.photobucket.com/user/stefer24/media/scan0024.jpg.html

Specific cat = instance "Oogie" of class "cat"

Class roles

- Every class is designed with a specific **role** in mind.
- The total set of functional requirements for a software system is broken down into a set of tasks
- Collections of tasks are grouped together and mapped to roles
- Roles are mapped to specific classes

Class Responsibility

- Take the requirements for a software application:
  - Identify the classes required
  - Assign specific Responsibilities to each class
  - Determine relationships between classes (see later)
  - Repeat the above steps until the design is correct
  - Each responsibility should be handled by that class and no other
    - Example: If a responsibility for keeping track of a person's name is assigned to a class called PersonClass then:
    - No other class should have this information
    - Other classes which need this information should refer to this class when the information is required

Comparison to non-OO design

- In a top-down procedural approach, we design an algorithm by starting with a main module and using step-wise refinement to determine the processing steps
- Some of these steps get refined into sub modules and the process repeats until the design is refined enough to code
- Under Object Orientation this all changes…

OO design

- Before the algorithm is designed:
  - The classes are identified
  - Each class is assigned role(s) or responsibilities
  - The required sub modules are designed (i.e. Constructors, accessors, etc)
  - Each Class is thoroughly tested via a test harness

- Finally, the main algorithm and any required sub modules is designed (making use of the developed classes in the process)

Nouns and Verbs

- Like algorithm design, the determination of classes is still a bit of an art form
- One simple technique is the nouns and verb approach:
  - Nouns are mapped to classes
  - Verbs are mapped to sub modules within classes
  - The definition of noun and verb gets stretched to cover collections of words
  - Result is that:
    - Sub module names should always describe an action (i.e. getName)
    - Class names should always describe a thing (e.g. PersonClass)
- It is important to note that the set of classes proposed will change over the design phase

Object Communication

- Sometimes referred to as **message passing**:
  - When an object of one class calls an object of another class it is passing a message (i.e. A request to the object to perform some task)
- The [public] methods must provide the functionality required for the class to fulfill its role.
- There are five categories of methods in a class:
  - The Constructors
  - The Accessor Methods (aka Interrogative Methods)
  - The Mutator Methods (aka Informative Methods)
  - Doing Methods (aka Imperative Methods)
  - [Private] methods

Classes in Python

- Order your code consistently

- Declare the components of each class in the following order:
  - Declarations for class constants and variables (global to the class)
  - Declarations for the Constructors (__init__)
  - Declarations of **instance variables** (local to each instance, usually in _init_)
    - e.g. self.myVar = value
  - *Accessor methods*
  - *Mutator methods*

    Python instance and class variables are public, so basic set/gets are not req'd
  - Doing methods ("public")
  - Internal methods ("private")

Classes in Python

- Note that everything in Python is "public" (unlike Java, C++) so we can only **treat** methods and data as private
- Use _methodName to indicate "private methods"

- Put the class files in a separate python file, e.g. DSAStack.py
- Your programs will then import from DSAStack as needed
- Unit tests (testing you classes/methods)
  - Option 1: Separate UnitTestDSAStack.py
  - Option 2: Include tests in DSAStack.py using

    if __name__ == "__main__":

    <tests in here>

Example: song

```
class Song():

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_a_song(self):
        for line in self.lyrics:
            print(line)

lumberjack = Song(["I'm a lumberjack and I'm OK",
                   "I sleep all night",
                   "And I work all day"])

spam = Song(["SPAM, SPAM, SPAM, SPAM",
             "spam, spam, spam, spam"])

lumberjack.sing_me_a_song()
spam.sing_me_a_song()
```

Instance variable

Object of class Song

Song: lumberjack
lyrics: ["I'm a lumberjack and I'm OK", "I sleep all night", "And I work all day"]

https://learnpythonthehardway.org/book/ex40.html

Self

- Why do I need self when I make __init__ or other functions for classes?
- If you don't have self, then code like cheese = 'Gorgonzola' is ambiguous.
- That code isn't clear about whether you mean the *instance's* cheese attribute/variable, *or* a local variable named cheese.
- With self.cheese = 'Gorgonzola' it's very clear you mean the instance attribute self.cheese.
- You can use any variable name, but self is the convention.

https://learnpythonthehardway.org/book/ex40.html

OO Design…Where to begin?

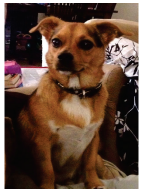
- Find your objects
- If we wanted to keep track of our household animals: cats, dogs and birds
- We could make classes for cats, dogs and birds
- For each animal, we might track:
  - name
  - date of birth
  - colour
  - breed

Cat
Name: DOB: Colour: Breed:

Test our objects out…

CAT
Name: Oogie
DOB: 1/1/2006
Colour: Grey
Breed: Fluffy

DOG
Name: Dude
DOB: 1/1/2011
Colour: Brown
Breed: Jack Russell



BIRD
Name: Big Bird
DOB: 10/11/1969
Colour: Yellow
Breed: Canary

CLASS RELATIONSHIPS

Goals of Object-Orientation

- Reuse / Extensibility
  - Reuse: each class provides its functionality to other classes
  - Can inherit from a class to reuse/extend its functionality
- Modularization - low coupling, high cohesion
  - Objects should be responsible for their own data state
  - Objects should represent a single concept and all methods should relate to that concept (high cohesion)
  - Only the object's interface should matter to a user of that object, not the details of its implementation (low coupling)

- ***Note***: *many of these slides are from Object-Oriented Program Design*

Class Relationships

- The classes of objects which communicate with each other via message passing share some form of relationship (association):
  - Aggregation
  - Composition
  - Inheritance
  - Other

Class Relationships

- Aggregation:
  - One class is declared as a class field within the other class
  - Communication is one way (most of the time?), from class to class field
- Composition:
  - One class is included as part of the other class
  - The included class does not exist without the host class

Class Relationships

- Inheritance:
  - One class is a descendant of another class
  - Uses polymorphism, method overloading or direct references to the superclass to communicate.
  - Communication is one way, from child to parent (sound familiar!!)
- Other:
  - Where objects of one class are related to another in a manner which is NOT aggregation or inheritance.
  - These other relationships will be discussed in future units.

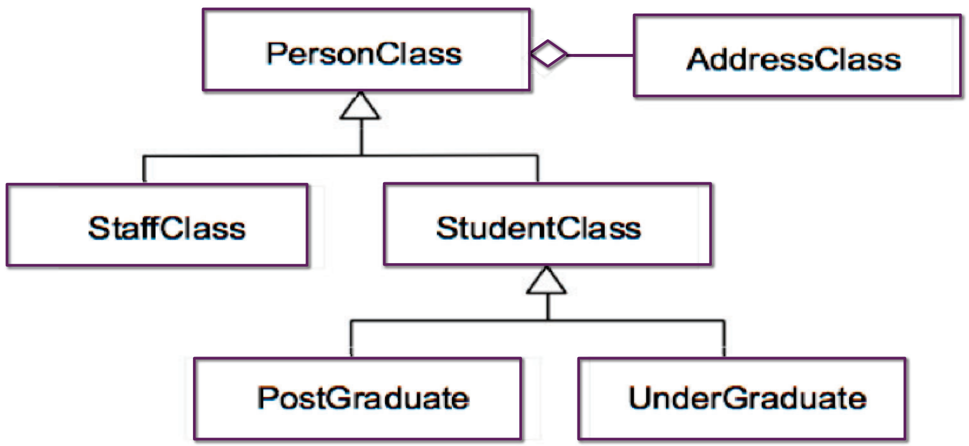
Object Communication

- Also referred to as message passing:
- When an object of one class calls a method in an object of another class it is passing a message
- A request to the object to perform some task

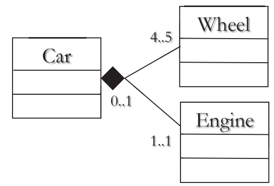
Modelling Languages

- Used to show the relationships between different classes and different instances of classes (i.e. objects) in a particular software
- Usually graphical
- Most commonly adopted methodology is known as **UML**:
  - **Unified:** a union of the approaches put forward by Grady Booch, James Rumbaugh and Ivar Jacobson
  - **Modelling:** a graphical representation (or model) of an OO software design
  - **Language:** provides a standard way of expressing object relationships (i.e. contains rules for syntax & semantics)
- Software Engineering units teach UML and OO software design.
- For now we will simply look at the UML notation for class diagrams - describing inheritance and aggregation/composition.

Uni People Example

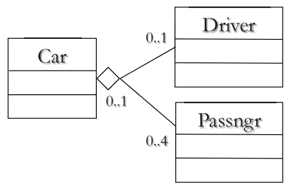


Class Relationships (1)



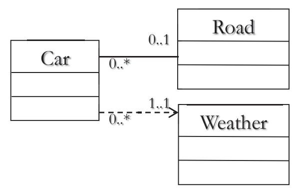
- Composition
  - **“has-a”** or “whole-part” relationship
    - UML: Shown with solid diamond beside container class
    - *e.g.*, Car “has-a” Wheel
  - Strong lifecycle dependency between classes
    - Car is not a car without four Wheels and an Engine
    - When Car is destroyed, so are the Wheels and Engine
  - In code:
    - Car would have Wheel and Engine as class fields

Class Relationships (2)



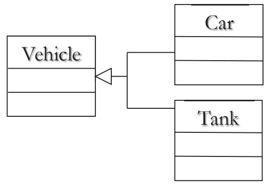
- Aggregation
  - Weaker form of composition, but is still **“has-a”**
    - UML: Shown with open/unfilled diamond beside container
  - Lifecycle dependency usually not strong
    - Car does not always have a driver
    - When Car is destroyed driver and passengers are not
    - Drivers can drive different cars
  - In code:
    - Car would have Driver and Passenger as class fields
    - …exactly like composition!

Class Relationships (3)



- Association and Dependency
  - Indicates interaction between classes
    - Association = solid line, Dependency = dashed line
    - Difference is murky: UML is a *guide*, not a *law*
  - Used to show that one class invokes methods on another
    - … but that there is no other relationship beyond this
    - With arrow, implies *unidirectional* (Car calls Weather, not vice-versa)
    - No arrow implies *bidirectional* (Car and Road call each other)
  - In code: Any way that a method call can be set up and made
    - *e.g.*, Weather object is passed as a parameter to a Car method
      - *e.g.*, Car.setAggressiveness(Weather currentConditions)
    - *e.g.*, Road has a class field of all Cars on that Road (aggregation?)

Class Relationships (4)



- Inheritance
  - **“is-a”** relationship
    - Indicates one class is a sub-type of another class
    - Shown with an open triangle arrowhead beside super-type
  - Implies the specialisation of the super-type
    - Super-type synonyms: ‘parent’, ‘base’
    - Sub-type synonyms: ‘child’, ‘derived’
  - In code: During class declaration; syntax is language-specific
    - Python: class Car(Vehicle):
    - Java: public class Car **extends** Vehicle
    - C++/C#: public class Car **:** Vehicle

Inheritance

- Inheritance is the ability of a new class of object to take on all of the properties of an existing class
  - i.e. the state and the functionality
- **Super Class**: The original class
- **Sub Class**: The new class which inherits all of the functionality of the super class
- The sub class can then:
  - Introduce additional state (class fields)
  - Modify the inherited functionality.
  - Introduce new functionality
    - i.e. more specialised
- The super class generally has less functionality than the sub class
    - i.e. more generalised

Aggregation v's Inheritance

- An aggregation relationship is implied by the class field declarations
- An inheritance relationship is explicitly stated (given in brackets on the class definition)

- Note that BOTH relationships encapsulate the functionality of one class within another:
  - Any inheritance relationship can be re-expressed as an aggregation relationship and vice versa.
  - The choice is based upon which relationship is most appropriate.

Class Responsibility

- Each class has a designated role or responsibility in the software system
- It may be that some classes have duplicated functionality
- This duplicated functionality can be removed and placed into a super class which the original classes inherit from
- It is important to ensure that a sub class never assumes the role of its super class
- If the sub class requires some super class functionality then it should call the appropriate super class method

Super Class - Sub Class Communication

- Communication is one way:
  - Sub class calls super class methods but not the other way around
- The word *super is used to refer to the super class*
- super() by itself is a call to the super class' __init__ method

- super().methodName() is a call to a public method in the super class
- Example:
  - In a super class there is a toString() method
    - outStr = super().toString()
  - The sub class toString method wishes to generate a string containing its own state plus the super class state:
    - outStr = super().toString() + self.state

The Base Class

- All classes except one inherit from another class
- A special class, known as the *base class, is the only class that does not*
- In Python this base class is called *object*
- If no inheritance relationship is specified then it automatically inherits from the base class
  - Note: In Python 2, a class definition needed to state it inherited from object – def class person(object)

Super Class / Sub Class Object Construction

- In order to construct a sub class object, a super class object must also be created
- The order of object construction is from the base class through to the sub class

animals.py - Dog Class (Lecture 9)

```
class Dog():

    myclass = "Dog"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Cat Class (Lecture 9)

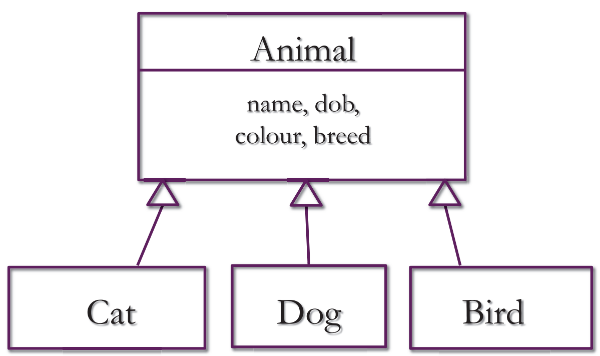
```
class Cat():

    myclass = "Cat"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Bird Class (Lecture 9)

```
class Bird():

    myclass = "Bird"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

Example: Inheritance

- Repetition should be avoided if possible
- Cat, Dog and Bird are nearly identical
- Factor out the duplicated fields and methods…



Example: animals.py

```
class Animal():

    myclass = "Animal"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def __str__(self):
        return(self.name + '|' + self.dob + '|' + self.colour+'|'+self.breed)

    def printit(self):
        spacing = 5 - len(self.myclass)
        print(self.myclass.upper(), spacing*' ' + ': ', self.name,'\tDOB: ',
                    self.dob,'\tColour: ', self.colour,'\tBreed: ', self.breed)
```

Example: animals.py – magic!

```
class Dog(Animal):

    myclass = "Dog"

class Cat(Animal):

    myclass = "Cat"

class Bird(Animal):

    myclass = "Bird"
```

Just the differences between the **Animal** superclass and the subclasses

These changes would have no impact on Shelter.py or pets.py

Polymorphism and Method Overriding

- An important aspect of inheritance for polymorphism is the ability to override methods of the base class
  - Consider passing a Tank to a method void drive(Vehicle veh)
  - A call to veh.accelerate() will actually call Tank's accelerate()
    - Which will behave differently to Car's accelerate()
- What is happening here?
  - Tank somehow becomes Vehicle. How?
  - What if you wanted to get back to Tank from Vehicle?
    - Since it really is a Tank, surely you can do it

Overloading vs Overriding

- Overloading is when many methods share the same name but differ in their parameters
  - Constructors are a good example: default, alternate and copy constructor all have the same name, different parameters
    - Uniqueness is defined by name + parameter *types*
      - This is called the method's *signature*, or *prototype*
    - *e.g.*, Car(String model) and Car(int numSeats) are different
    - But: Car(String model) and Car(String ownerName) cannot be disambiguated - will cause compiler error
    - Note that return type is *not* part of the method signature
  - Most modern languages support overloading
    - C and Fortran are a couple that don't support overloading

Overloading vs Overriding

- Overriding is where a method has exactly the same signature as a method in a *super/parent/base* class
  - *i.e.*, the child class is overriding the behaviour of the parent
  - Only applies to object-oriented languages, and all O-O languages support it
    - Overriding = specialisation, one of the cornerstones of O-O
- A method can be an overload *and* an override
  - Overloads the name of another method *in the* ***current*** *class*
  - Overrides the signature of a method *in the* ***parent*** *class*

this, super keywords

- Keyword ‘this’ is a reference to the *current* *object*

e.g.,
```
public Tank clone() {
    return new Tank(this); // Use copy constructor to make copy of ourselves
}
```

- Keyword ‘super’ is a ‘reference’ to the current object’s *parent* *class*
  - Use it to force a call to the parent class’s code

e.g.,
```
public Tank(Tank otherTank) {   // Copy constructor
    super(otherTank);  // Call parent’s copy constructor code first
    // Now do our own copy constructor code... }
```

e.g.,
```
public void doSomething() {   // A method
    super.doSomething();         // Call parent’s doSomething() code
    // Now do our own code... }
```

- super and this are relative to the current object/class
  - this = current object
  - super = current class’s direct parent class

Casting Between Types

- Changing from one type to another is called casting
  - You can also cast between numeric primitive types
    - *e.g.*, ints to floats and vice-versa, but not int to String.
    - C/C++ let’s you cast *anything* - it’s your problem if its wrong!

```
float fNum = 1.01;
int iNum = (int)fNum;              ← Cast by placing target data type in brackets
```

- Java (and pretty much every language) will implicitly do casts for you when it knows that the cast is ‘safe’
  - Since Tank is-a Vehicle, casting Tank to Vehicle is safe

```
Tank t = new Tank();
Vehicle v1 = t;                ← Implicit cast
Vehicle v2 = (Vehicle)t;       ← Explicit cast, same result as implicit cast
```

- There’s no need to explicitly do the casting here

Casting Between Types (2)

- So when do you have to cast? And why?
  - When you are casting between numeric types
    - because loss of information can occur, *e.g.,* float 1.01 → int 1
  - When you are attempting to downcast to a derived class
    - *e.g.*, casting Vehicle to Tank is not safe since the compiler cannot be sure that the object (of known type Vehicle) is a Tank or not
      - Tank is-a Vehicle does not mean Vehicle is-a Tank!
- If you know the cast is OK you can do it explicitly
  - *e.g.*, You know that the Vehicle really is a Tank
  - Compiler then leaves it to run-time to try the cast
    - Fails at run-time with a ClassCastException if it’s not a Tank

Casting Between Types (3)

```
Vehicle v = new Tank();      ← Implicit cast is happening here
Tank t1 = v;                 ← Compiler error
Tank t2 = (Tank)v;           ← OK, and will work at run-time too
Car c1 = (Car)v;             ← Will compile, but fails at run-time
```

- Some notes on casting
  - Primitives:
    - Casting from floats to ints will truncate the decimal places
    - Casting from ints to floats may lose some numerical precision
  - Classes
    - Object is a handy class to use for making general-purpose containers - simply contain an Object and you can contain *anything*
      - You have to explicitly cast back to the right class later though

Checking Class Type

- Downcasting sounds a bit risky
  - What if you aren't totally sure of the object's true class?
    - Downcasting could cause a ClassCastException
    - Could catch this exception and try again, but that's ugly
  - Java provides you with a solution: instanceof keyword
    - Let's you check if object A is really an instance of class X

```
Vehicle v = new Tank();
if (v instanceof Tank) {        ← Check if it really is a Tank
   Tank t1 = (Tank)v;
}
```

- **Warning:** try to limit your use of instanceof since it can be an indication of bad design and makes polymorphism redundant
  - Plus, if you are certain that the cast is OK, instanceof is a waste

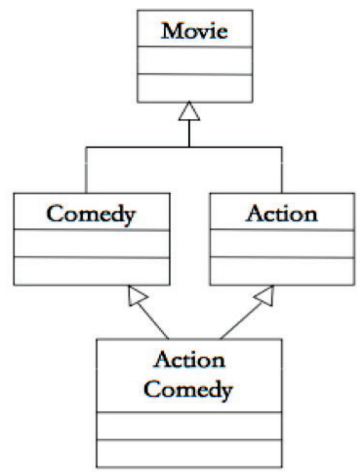
Multiple Inheritance

- So what do we do if a class is required to inherit the state and functionality of more than one super class?
- So Tank "is-a" Vehicle
  - But Tank "is-an" Artillery as well, not just a Vehicle and Artillery is not always a Vehicle, so can't put Artillery in between Tank and Vehicle
  - ie: Tank really has more than one base class
- One solution: allow multiple inheritance (eg: Python, C++)
  - Tank inherits from *both* Vehicle and Artillery

Multiple Inheritance – Problems

- Theoretically, multiple inheritance is fine
- But in practice (in the code), things can get messy
- Say both Vehicle and Artillery define a method getSize()
  - If Tank does not override getSize(), which getSize() version should the compiler call? Vehicle's? Artillery's?
  - Worse, what if Artillery.getSize() refers to the size of the *shells* it fires, but Vehicle.getSize() refers to the *vehicle's* size?
- In more complicated inheritance hierarchies, you can even inherit from the same class more than once!
  - The next slide shows an example of this

Multiple Inheritance - Example



Interfaces (Java)

- Interfaces are used as a solution to resolve (some of) the problems with multiple inheritance
  - An interface is essentially an abstract class where:
    - All methods are abstract (ie: have no implementation)
    - All methods are public
    - No class fields exist
- In other words, an interface class only defines a set of public methods that its child classes must implement
  - Note that interfaces cannot have a constructor
    - There's nothing to construct, so what would be the point?
  - Interfaces can inherit from (extend) other interfaces, but do not have to (unlike classes, which extend at least Object)

Interfaces and Multiple Inheritance (Java)

- Many multiple inheritance issues can then be resolved
  - Allow inheritance from as many interfaces as required
    - Interface inheritance
  - BUT only allow inheritance from a single class, which includes abstract classes
    - Implementation inheritance
- Why does this help?
  - Because interfaces cannot have any code
  - Thus there is never any confusion as to which base class's method should be invoked - there is only ever one base class with an implementation (all others are interfaces)

Interfaces and Multiple Inheritance (Java)

- Interfaces are not a magic cure-all
  - *e.g.*, If Vehicle and Artillery are both made into interfaces, but getSize( ) has different meanings for both:
    - Tank still can't properly choose how to override getSize( )
    - C# has the ability to define different methods, one per interface
  - *e.g.*, Action Comedy
    - Action and Comedy aren't abstract, and so can't be interfaces
    - Could make *all* movie genres into interfaces, and have separate implementation classes inheriting from these. Messy!
  - Limits code reuse potential
    - Interfaces have no implementation (code) to reuse!

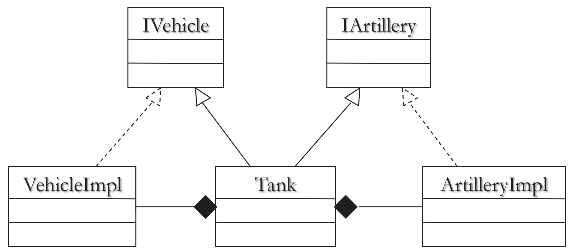
Interfaces and Multiple Inheritance (Java)

- Interfaces are not a magic cure-all
  - *e.g.*, If Vehicle and Artillery are both made into interfaces, but getSize( ) has different meanings for both:
    - Tank still can't properly choose how to override getSize( )
    - C# has the ability to define different methods, one per interface
  - *e.g.*, Action Comedy
    - Action and Comedy aren't abstract, and so can't be interfaces
    - Could make *all* movie genres into interfaces, and have separate implementation classes inheriting from these. Messy!
  - Limits code reuse potential
    - Interfaces have no implementation (code) to reuse!

Emulating Multiple Impl Inheritance

- Ideally Tank would inherit from Vehicle and Artillery
  - …and both would have code that Tank can reuse
    - *i.e.,* they are not interfaces, probably abstract classes instead
- The aforementioned issues with M.I. are in our way
  - But we can *emulate* M.I. with interfaces and composition
    - Have Tank inherit from interface IVehicle
    - Have Tank compose with (contain) a class VehicleImpl that implements all the would-be-non-abstract methods of IVehicle
      - VehicleImpl might also inherit from IVehicle, but will have to bomb out on any truly-abstract method - a bit messy
    - Have Tank 'delegate' calls to equivalent methods in VehicleImpl
      - VehicleImpl code can then be shared (re-used) with other classes
    - Then do the same with IArtillery

Emulating M.I. - Example



Class vs Abstr Class vs Interface Inheritance

- Inherit from classes…
  - …when you need to specialise behaviour of existing class
- Inherit from abstract classes…
  - …where a lot of the code in derived classes is common among most/all of the derived classes
  - The abstract class is then a 'repository' for shared code
- Use interfaces and composition+delegation…
  - …everywhere else
    - It avoids wasting your precious single base class
    - It also helps you get around integrating with or reusing existing classes - inherit from one, compose+delegate with others

Interfaces in Code

- Naming:
  - A prefixed capital 'I' is common for interfaces, eg: IVehicle
- In code:
  - Declaring: Almost identical to declaring a class
    - Java: public **interface** IVehicle { … methods here … }
    - C#: public **interface** IVehicle { … methods here … }
  - Inheriting from:
    - Java: public class Tank **implements** IVehicle, IArtillery
    - C#: public class Car **:** IVehicle, IArtillery
  - Can use extends and implements keywords together:
    - public class Tank **extends** MilitaryObject **implements** IVehicle, IArtillery

ABSTRACT DATA TYPES

SortedList – an Abstract Data Type

- Last week we saw the value of sorting, and some sorting algorithms
- Our searching will be faster if the data is sorted
- How do we maintain a sorted list if the data is changing?
- Over time we may need to insert and delete values
- Create a class SortedList holding the list and the current number of elements
- The main operations are:

find – insert - delete

SortedList - Class Diagram (UML)

SortedList
- theSortedList : array of integers - numElements : integer
+ find(int key): int + insert(int key): none + delete(int key): none

find

- assume theSortedList and numElements are classfields

```
Submodule: find (AKA linear search)
Import: key (item to find)
Export: location (index)
Assertion: returns the location of key if it exists in the array,
otherwise throws an exception

  location=0, found = false
  DO
     IF theSortedList[location].equals <-- key
        found = true
     ELSE
        increment location
  WHILE NOT found AND location < numElements

  IF NOT found
     throw appropriate exception
```

insert

```
Submodule: insert
Import: key (item to insert)
Export: None
Assertion: inserts key/value into
array at correct, sorted position,
otherwise throws an exception
```

- three scenarios
  1. **End** of list of values
     - position = element [numElements]
     - easy!

```
IF theSortedList is not full         ← throw exception if it is!
   theSortedList [numElements] = insertValue
   increment numElements
ENDIF
```

1. **Beginning** of list of values
   - element [0]

```
IF theSortedList is not full                  ← throw exception if it is!
   FOR ii= numElements, ii>0, decrement ii    ← Shuffle elements away to make room
      theSortedList[ii] = theSortedList[ii-1]
   ENDFOR
   theSortedList[0] = insertValue
   increment numElements
ENDIF
```

insert

3. Somewhere inside list of values
   - Need to search for position, then insert
   - Array needs to shuffle down to make space

```
position=0
IF theSortedList is not full                                          ← throw exception if it is!
   WHILE  insertValue < theSortedList[position] AND position < numElements
      increment position
   ENDWHILE

   FOR ii=numElements, ii>position, decrement ii                      ← Shuffle elements away
      theSortedList[ii] = theSortedList[ii-1]
   ENDFOR
   theSortedList[position]=insertValue
   increment numElements
ENDIF
```

delete (remove)

```
Submodule: delete
Import: key (item to insert)
Export: None
Assertion: deletes key/value from
array, otherwise throws an exception
```

- need to ensure array is not empty!
  - throw exception if it is
- three scenarios
  1. End of list of values
     - Element [n-1] is deleted
     - Decrement count.
  2. Beginning of list of values
     - Element [0] is deleted
     - Starting from element [1], shuffle the rest of the elements down by one, overwriting element [0].
     - Decrement count.
  3. Somewhere else in list
     - Element[x?]
     - Find the element to delete.
     - Starting from the next element, shuffle the rest of the elements down by one, overwriting the element to delete.
     - Decrement count.

Time Complexity – SortedList

Operation	Best Case	Average Case	Worst Case
find	O(1)	O(N)	O(N)
insert	O(N)	O(N)	O(N)
delete	O(N)	O(N)	O(N)

Each operation needs to do a "find", then a possible shuffle O(N)

Data Structures

- Arrays are a type of *data structure*
  - They define how to organise data in memory
  - In particular, arrays store a set of elements in a single contiguous block of memory, accessed via an index
- Data structures such as arrays can be useful as they are, but they aren't always a perfect fit
  - Many applications need to access data differently to the array's 'index-update' approach
    - *e.g.,* an order processing queue: take from front, add to rear
  - Problem: an array is really *how a computer operates*
    - RAM is just one long 1D array (same with disk storage)

Abstract Data Types

- So there can be a gap between the data structure (how it works) and the *usage* of that structure
- Abstract Data Types are there to define behaviour
  - ADT: a set of methods that provide access to data in a way that is natural for the application
  - How the methods manipulate the underlying data structure to achieve this is not the app's problem
    - Even the data structure used is hidden!
  - ADTs make developing applications much easier
    - Write the ugly details once and wrap it all in nice methods
    - Lets you later concentrate on the application logic rather than the details of manipulating the data structure

Abstract Data Types as Objects

- ADTs are defined in terms of operations
- Objects bundle state and operations together
- Our objects (classes) must include
  - Code to implement all ADT operations
  - Instance variables to support the required state (e.g. array of data, count)
  - Methods for initialising the objects
  - Support methods, e.g. display( )
  - Validation and exception handling throughout

- We may choose different internal implementations:
  - Data types and structures (e.g. arrays, lists, trees)
  - Algorithms (e.g. sorting, searching, traversing)

STACKS AND QUEUES

Stacks and Queues

- Two very common ADTs are stacks and queues
  - Queue: elements taken out in the order they were added
    - FIFO: first-in, first-out (although not all queues are FIFO queues)
  - Stack: data elements are taken out in *reverse* order
    - LIFO: last-in, first-out
  - Elements *must* be taken out in the appropriate order: you can't jump in and grab the 5th element
- Such processing occurs a lot in the real world
  - And we often need to model such processes in software
- But: arrays aren't necessarily best for implementing these ADTs

Queue vs Array

- Consider the behaviour of a queue vs an array:
  - Nothing stops you from accessing array element [5]
    - But a queue should only take the first element each time
  - If you take the first array element [0], element [1] doesn't automatically move to position [0]
    - So then you have to remember that the 'new-first' element is [1],
    - or shuffle all the elements up by one yourself
- Solution: use methods to make the array behave like a queue
  - Just because it's messy doesn't mean it's impossible
    - …but it means we only have to CODE AND TEST IT ONCE!
  - If we code it right, using it in the application will simplify (and clarify) the rest of the code enormously

Stacks

- Let's start with stacks, because they are easier!
- A stack is an ADT that implements a LIFO list
  - Think of a stack of plates – add to top, take from top
  - Some example applications for stacks:
    - Converting a character string into an int (*e.g.,* "10" → 10)
    - Storing information for method calls
    - Evaluating a mathematical expression (We'll see later on)
- Since it's an ADT, we'll first talk about *what* a stack's behaviour is
  - Then we will discuss *how* to implement a stack
    - In particular: with an array data structure (this time)

Stack Methods

- Being LIFO, a stack has a few obvious methods, with standard names that everyone recognises:
  - push() – add a new item to the top of the stack
  - pop() – take the top-most item from the stack
  - top() – look at the top-most item, but leave it on the stack
    - Synonym: peek()
  - isEmpty() – check if the stack is empty
- There are also extra methods that often appear
  - isFull() – checks if the stack is full
    - Arrays can get full, but some data structures don't have this issue
  - count() – number of elements in the stack
    - Synonyms: size(), numElements() (not as standardised!)

Stack Implemented with an Array

- Java and Python have built-in classes for stacks, but we'll develop our own DSAStack to illustrate the concept
  - DO NOT USE BUILT-IN DATA TYPES AND ALGORITHMS IN DSA
  - Let's create a stack of double values to hold numbers
- The only data structure we know (so far) for storing sets of data is the array … so we'll use arrays
- How are we going to do it?
  - Look for similarities that we can exploit
  - Consider: A stack grows and shrinks on *one side*
  - Similarly, array elements start at [0], and can be added to / removed from the end until the array capacity is reached

Stacks with Arrays

- So, if we make the *top* of the stack be the *back* of the array, we can grow/shrink without much hassle
  - Counter-intuitive, but simplifies the code a lot!
- The idea is to keep track of the count of elements in the array
  - The element at [count – 1] is then the top of the stack
    - – 1 because arrays are zero-based in Java/Python, remember!
  - New items then get stored in slot [count]
    - [count-1] is the top, so [count] is the next unused slot
  - When count == array.length, the stack isFull

Stack - Pseudocode

```
Class DSAStack
Class fields : stack (double array), count (integer)
Class constant : DEFAULT_CAPACITY ← 100

Default constructor
  alloc stack array with DEFAULT_CAPACITY elements
  count ← 0

Alternate constructor IMPORT maxCapacity (integer)
  alloc stack array with maxCapacity elements
  count ← 0


ACCESSOR getCount IMPORT none EXPORT count

ACCESSOR isEmpty IMPORT none EXPORT empty (boolean)
  empty ← (count = 0)

ACCESSOR isFull IMPORT none EXPORT full (boolean)
  full ← (count = stack length)

                                <continued next slide>
```

Stack - Pseudocode (cont.)

```
MUTATOR push IMPORT value EXPORT none
  IF isFull() THEN
    ABORT                  ← ie: throw an exception
  ELSE
    stack[count] ← value
    count ← count + 1
  ENDIF

MUTATOR pop IMPORT none EXPORT topVal
  topVal ← top()
  count ← count - 1

ACCESSOR top IMPORT none EXPORT topVal
  IF isEmpty() THEN
    ABORT
  ELSE
    topVal ← stack[count - 1]
  ENDIF
```

Application: Palindrome

- How can we check if a string (or number) is a palindrome?
- Need to check if it's the same forward and backward.
- We can achieve this with a stack…

```
IMPORT: inString
EXPORT: match
create a new palStack
FOR ch ← 0 TO inString.length -1 DO
   palStack.push ← ch
ENDFOR

pos = 0
match = TRUE

WHILE match AND NOT palStack.isEmpty
   match = inString[pos] == palStack.pop
   pos = pos + 1
ENDWHILE
```

Application: ReadInt

- In the lecture on recursion we saw that the system stack can be used to convert characters read from the keyboard to an integer.
- We can also achieve this with our own stack.

```
create a new intStack
ch = readChar
WHILE '0' <= ch <= '9'
   digit = ch - '0'
   intStack.push<-- digit
   ch = readChar
ENDWHILE

value = 0
powerOfTen = 1

WHILE NOT intStack.isEmpty
   digit = intStack.pop
   value = value + digit * powerOfTen
   powerOfTen *= 10
ENDWHILE
```

Application: Evaluation of Maths Equations

- Stacks *really* become useful for non-obvious tasks
  - Evaluation of maths expressions is one of those tasks
- The problem:
  - We normally see equations in the form:
    `(10.3 * (14 + 3.2)) / (5 – 2 * 3)`
  - There are many precedence rules that need to be followed
    - BIMDAS or BOMDAS
    - Makes it hard to write code to solve it in the right order

Infix to Postfix

- Solution: Re-order the equation so that higher precedence operations come before lower ones
  - Plus we get rid of brackets, even nested brackets
  - Then we just need to read it from left-to-right
- How?
  - Normal equations are in what is called 'infix' notation
    - Unfortunately it's not possible to rewrite equations in infix to get rid of precedence ordering and brackets. Consider:

```
Normal:        (10.3 * (14 + 3.2)) / (5 + 2 – 4 * 3)
Left-to-Right: 14 + 3.2 * 10.3 / -4 * 3 + 5 + 2  (ie: no BIMDAS)
```

    - Close, but the 10.3 / -4 is wrong – we needed to 'postpone' evaluating it until after the + 2. But with infix we can't postpone

Postfix

- Solution: use a different notation, postfix
  - Put the operator *after* the operands it applies to (the 'post')
  - Each operator then applies to the two operands that precede the operator
- How does this help?
  - You only evaluate operands once you see an operator
    - Before that, you just keep adding operands to a pile
    - Since the operator must be applied to the *last* two operands (LIFO), your 'pile' is in fact a stack

Infix vs Postfix Examples

- The original equation in Postfix:

```
Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)
Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /
```

- Some simpler examples:

Infix	Postfix
`3 * 4`	`3 4 *`
`2 - 4 + 3`	`2 4 - 3 +`
`4 + 2 * 3`	`4 2 3 * +`
`(4 + 2) * 3`	`4 2 + 3 *`
`((2 - 3) / 4 * (1 + 9)) * 2`	`2 3 - 4 / 1 9 + * 2 *`

Postfix Properties

- Points to note:
  - The order of the operands is left **unchanged**
  - Operators are listed in **precedence order**
    - … even the effect of brackets has been taken into account
  - Equal-precedence operators are kept in the infix order
    - left to right associativity
      - *e.g.*, 2 – 4 + 3 → 2 4 – 3 + NOT 2 4 3 + –
      - Reason: 2 – 4 is in fact 2 + (-4), so we *must* keep the –ve sign related to the 4: 2 – 4 ≠ 4 – 2

      2 4 3 + – is actually postfix for 2 – (4 + 3)
      - Same reasoning applies to \ : A \ B ≠ B \ A
      - + and * aren't so problematic, since A + B = B + A

Evaluating Postfix

- Evaluating postfix expressions will give some more insight into why it all works
  - We'll discuss infix → postfix conversion a little later
    - … because it's harder!
- Unsurprisingly, we use a stack in the evaluation
  - Push operands onto stack until an operator is encountered
  - Pop off last two operands and apply the operator to them
    - Apply the operator *in-order*, not LIFO order (important for –, /)
  - Push the result back on the stack ready for the next op
  - When no more operands/operators are left in the postfix, the answer is the (single) value remaining on the stack

Postfix Evaluation Example

```
Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)
Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /
```

PFix	Eval Stack Contents	What's Happening?
10.3	10.3	<push 10.3>
14	10.3 14	<push 14>
3.2	10.3 14 3.2	<push 3.2>
+	10.3 17.2	<2 pops> → 14 + 3.2, <push ans>
*	177.16	<2 pops> → 10.3 * 17.2, <push ans>
5	177.16 5	<push 5>
2	177.16 5 2	<push 2>
+	177.16 7	<2 pops> → 5 + 2, <push ans>
4	177.16 7 4	<push 4>
3	177.16 7 4 3	<push 3>
*	177.16 7 12	<2 pops> → 4 * 3, <push ans>
-	177.16 -5	<2 pops> → 7 - 12, <push ans>
/	-35.432	<2 pops> → 177.16 / -5, <push ans>
<end>	-35.432	<pop> → Final answer

Infix to Postfix Conversion

- Converting infix to postfix *also* uses a stack
  - Postfix needs to re-arrange operators into the right place
  - So we need to 'hold on' to operators until we reach the right point in the equation to insert them back in
    - Remember that operands don't change their order
  - The method behind this is to hold back an operator until we see an equal-or-lower-precedence operator
    - If the new operator is higher precedence, we have to put it 'on top' of the other operator (in a stack), since it takes precedence
  - Brackets are an extra wrinkle
    - Approach: treat sub-equations in brackets as if they were isolated from the rest of the equation (because they are!)

Infix to Postfix Conversion: Algorithm

```
                                        NOTE: Methods in red must also be implemented,
                                               but are fairly straightforward tasks
postfix ← empty
WHILE infix has more terms DO
   term ← ParseNextTerm()               ← Extract next term (operator, operand) from infix eqn

   IF (term = '(') THEN
      opStack.push('(')                 ← '(' gets put straight onto the stack

   ELSE IF (term = ')') THEN
      WHILE (opStack.top ≠ '(') DO      ← Find corresponding '('
         postfix ← postfix + opStack.pop   ← Pop remaining operators for the bracketed sub-equation
      ENDWHILE
      opStack.pop                       ← Pop the '(' and discard it

   ELSE IF (term = '+') OR (term = '-') OR (term = '*') OR (term = '\') THEN
      WHILE (NOT opStack.isEmpty) AND (opStack.top ≠ '(') AND
            (PrecedenceOf(opStack.top) >= PrecedenceOf(term)) DO
         postfix ← postfix + opStack.pop  ← Move higher/equal precedence ops to postfix eqn
      ENDWHILE
      opStack.push(term)                ← Always put the new operator onto the stack

   ELSE                                 ← Term must be an operand if it isn't an operator
      postfix ← postfix + term          ← Add operand to postfix equation
   ENDIF
ENDWHILE

WHILE (NOT opStack.isEmpty) DO          ← Pop any remaining operators from the stack
   postfix ← postfix + opStack.pop
ENDWHILE
```

Infix to Postfix Example

```
Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)
Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /
```

Infix	Postfix So Far	Operator Stack
(		(
10.3	10.3	(
*	10.3	(*
(	10.3	(* (
14	10.3 14	(* (
+	10.3 14	(* (+
3.2	10.3 14 3.2	(* (+
)	10.3 14 3.2 +	(*
)	10.3 14 3.2 + *	<empty>
/	10.3 14 3.2 + *	/
(	10.3 14 3.2 + *	/ (
5	10.3 14 3.2 + * 5	/ (
+	10.3 14 3.2 + * 5 2	/ (+
2	10.3 14 3.2 + * 5 2	/ (+
-	10.3 14 3.2 + * 5 2 +	/ (–
4	10.3 14 3.2 + * 5 2 + 4	/ (–
*	10.3 14 3.2 + * 5 2 + 4	/ (– *
3	10.3 14 3.2 + * 5 2 + 4 3	/ (– *
)	10.3 14 3.2 + * 5 2 + 4 3 * –	/
<end>	10.3 14 3.2 + * 5 2 + 4 3 * – /	<empty>

Postfix Conversion ‘Checklist’

- Things to keep in mind:
  - Don’t forget to write down the brackets in the infix!
  - New operators ALWAYS go onto the stack
    - They *never* get put directly onto the postfix expression
    - The only question is whether to first pop the operator that is *already on the stack* off to the postfix expression
  - Brackets NEVER appear in the postfix
    - And closing brackets never appear in the operator stack – they are only markers to indicate the end of the sub-equation
  - Remember to pop off any remaining operators at the end of each sub-equation or at the end of the full equation

FIFO Queues

- A FIFO queue is an ADT implementing a FIFO list
  - Other kinds of queues aren’t FIFO, eg: priority queue
- Examples of where FIFO queues are needed
  - Bank transactions: processed in the order they are made
  - Customer orders: first come, first served

Queue Methods

- Queues (FIFO or otherwise) have the following methods
    - Note: naming isn’t as standardised as it is with stacks
  - enqueue() – add item to the queue
    - FIFO queues add to the end, priority queues insert in priority order
    - Synonyms: add(), insert()
  - dequeue() – take item from the front of the queue
    - Synonyms: remove(), delete()
  - peek() – check the front item, but don’t take it off
    - Synonyms: front()
  - isEmpty() – check if the queue is empty
  - isFull() – check if the queue is full. Optional
  - count() - number of elements in the queue. Optional

FIFO Queue with an Array

- Unlike stacks, queues grow on one side (the end) and shrink on the other (the front)
  - No synergies with arrays to be taken advantage of here!
- Two options are available:
  - Shuffle queue elements forward when front is dequeued
    - Exactly like a real-world queue, like at the bank
  - Leave elements as-is and change which index is ‘front’
    - *i.e.,* dequeued indexes are no longer used
    - Circular queue: allow the queue to cycle around the array, so that previously-dequeued indexes can be re-used

‘Shuffling’ vs Circular Queues

- Time Efficiency:
  - Shuffling: every dequeue must move N elements up by 1
  - Circular: Only need to adjust front index – much faster
- Space Efficiency:
  - Both have same space usage: circular queues can just start at idx [5], go through [length-1] and wrap around to end at [4].
  - But both still have a maximum size (due to fixed-size array)
- Code Complexity:
  - Shuffling: easy to understand, code, and maintain
  - Circular: Dealing with the wrap-around can be tricky – simplify it by storing the count as well as start/end indexes

FIFO Queue – Pseudocode (Shuffling)

```
Class DSAQueue
Class field : queue (double array), count (integer)
Class constant : DEFAULT_CAPACITY ← 100

Default constructor
  // implement this yourself

Alternate constructor IMPORT maxCapacity (integer)
 // implement this yourself


ACCESSOR getCount IMPORT none EXPORT count

ACCESSOR isEmpty IMPORT none EXPORT empty (boolean)
 // implement this yourself

ACCESSOR isFull IMPORT none EXPORT full (boolean)
 // implement this yourself
```

<continued next slide>

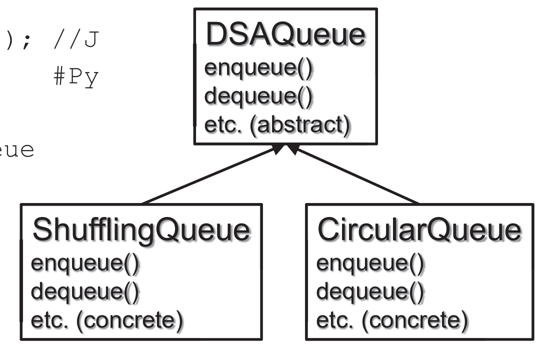
FIFO Queue – Pseudocode (cont.)

```
MUTATOR enqueue IMPORT value EXPORT none
 // implement this yourself

MUTATOR dequeue IMPORT none EXPORT frontVal
 // implement this yourself

ACCESSOR peek IMPORT none EXPORT frontVal
 // implement this yourself
```

FIFO Queues - Polymorphism

- We can implement queues as shuffling or circular queues
- Using polymorphism, we can minimise changes required
  - Switch between implementations by changing one line of code

```
myQ = new ShufflingQueue( ); //J
myQ = ShufflingQueue( )      #Py

// use methods from DSAQueue
myQ.enqueue(200)
myQ.peek( )
```



Next Week

- Linked lists
- Iterators