Data Abstractions
→ **Abstract Data Types**

→ **Data Abstractions**
  - How to define them
  - Implementation issues
  - Abstraction functions and invariants
  - Adequacy (and some requirements analysis)

→ **Towards Object Orientation**
  - Differences between object oriented programming and data abstraction
Imagine we want to hold information about dates

- E.g. year, month, day, hours, minutes, seconds, day of week, etc.
  - Could use an integer arrays: int date[3];
  - and write some support functions for computing day of week, comparing dates,
- But suppose we then decide years need 4 digits rather than 2 (i.e. 2001 instead of 01)
  - we have to change every part of the program that uses dates

Encapsulation

- we really want to distinguish the abstract notion of a 'date' from it's concrete representation
- we want to hide all the details about how dates are represented
- Benefits:
  - modifiability, testability, readability, reduced complexity, [Y2K compliance(!?)]

Example:

```c
int today[3];
int lecture1_time[3];
...
today[0] = 01;
today[1] = 10;
today[2] = 01;
lecture1_time[0] = 9;
lecture1_time[1] = 0;
lecture1_time[2] = 0;
```
→ Programming languages provide:
  ✦ Some concrete data types
    ➢ integers, characters, arrays, ...
  ✦ Some abstract data types
    ➢ floating point, lists, tables, two dimensional arrays, records, ...
  ✦ Abstract data types are implemented using concrete datatypes
    ➢ (but you don’t need to know this to use them)

→ Operations are provided for each datatype
  ✦ e.g. creation, assignment, etc.
  ✦ ... but you cannot muck around with the internal representations
    ➢ e.g. float is represented in two parts, but you cannot access these directly
  ✦ But: some languages do allow you access to the internal representations
    ➢ e.g. in C, you can use pointers to access the internals of arrays
    ➢ this removes the distinction between the abstraction and the implementation
    ➢ it destroys most of the benefits of abstraction
    ➢ it causes confusion and error
→ Encapsulation is improved if you create your own data abstractions

.choice of what abstractions to create depends on the application

<table>
<thead>
<tr>
<th>Application</th>
<th>Useful data abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler writing</td>
<td>tables, stacks, ...</td>
</tr>
<tr>
<td>Banking</td>
<td>accounts, customers, ...</td>
</tr>
<tr>
<td>Mathematical computing</td>
<td>matrices, sets, polynomials, ...</td>
</tr>
<tr>
<td>Graph Editing</td>
<td>graphs, nodes, edges, positions</td>
</tr>
</tbody>
</table>

.choice of operations depends on how you want to manipulate the data

.e.g. bank accounts: open, close, make a deposit, make a withdrawal, check the balance, ...

.e.g. graphs: initialize, add nodes, remove nodes, check whether there is an edge between two nodes, get the label for a node,...

→ Most languages support creation of new datatypes

... but they might not force you to specify the data abstraction

... and they might not enforce information hiding
→ **Four groups of operators:**

- **Creators**
  - create new objects of the datatype

- **Producers**
  - take existing objects of the datatype and build new ones

- **Mutators**
  - modify existing objects of the datatype

- **Observers**
  - tell you information about existing objects of the datatype (without changing them)

→ **Immutable datatypes...**

- ...don’t have mutators
  - they can be created and destroyed, but not modified
  - once you’ve created an object you cannot change it

---

**Example: sets**

**Creators:**
- create a new empty set, ...

**Producers:**
- set union,
- set intersection, ...

**Mutators:**
- add an element,
- remove an element, ...

**Observers:**
- set size,
- set membership,
- set equality,
- test for empty set, ...
The abstraction should:

- name the data type
- list its operations
- describe the data abstraction in English
  - say whether it's mutable or not
- give a procedural abstraction for each operation
  - the abstraction only lists the "public" operations
  - there may be other "private" procedures hidden inside...

```/*
datatype set has operators create, insert, delete, member, size, union, intersection.
overview:
sets are unbounded mathematical sets of integers. They are mutable: insert and delete are the
mutation operations.
operations:
procedure create () returns set
effects: x is a new empty set

procedure insert (set s, int x) returns null
effects: adds x to the set s such that s' = s \( \{ x \} \)

procedure delete (set s, int x) returns null
requires: x ∈ s
effects: s' = s - \( \{ x \} \)

... (etc) ... */```
public class IntSet {
    //Overview: IntSets are mutable, unbounded sets of integers. A typical IntSet is \{x_1, \ldots, x_n\}

    //Creators
    public IntSet ()
        //effects: Initializes this to be the empty set

    //Mutators
    public void insert (int x)
        //effects: adds x to the set this such that this' = this \{x\}

    public void delete (int x)
        //requires: x \in this
        //effects: this' = this - \{x\}

    //Observers
    public boolean member (int x)
        //effects: returns true if x \in this, false otherwise

    //Producers
    public IntSet intersection (IntSet a)
        //effects: returns a new set representing a \cap this
Implementing Data Abstractions

→ Choose a representation that:
  ➜ permits all operations to be implemented easily (and reasonably efficiently)
  ➜ permits frequent operations to run faster

→ Example: sets
  ➜ an unsorted array with repeated elements
    ➜ insert is very fast, union is fast, intersection and member are slow, delete is very slow
  ➜ a sorted array
    ➜ insert is very slow, member is very fast, intersection is fast, union is slow
  ➜ a linked list
    ➜ insert is fast, delete is fast, union is slow, takes more memory

→ Choose a programming mechanism
  ➜ Package
    ➜ hides 'private' code, package has to be 'imported' (e.g. Ada, C, Modula)
  ➜ Object
    ➜ provides inheritance, operations called by message passing (e.g. C++, Java)
  ➜ Abstract datatype
    ➜ provides strong type checking, object becomes part of the language (e.g. C, ML)
Abstraction vs Implementation

- There is a mapping between abstract objects and their representations
  - several rep objects might map to the same abstraction object
  - some rep objects might be invalid
  - every abstract object must have a rep object
A data abstraction is adequate if...
- it provides all the operations the 'users' (e.g. other programmers!) will need

Choices, choices, choices...
- e.g. for sets, member(s,x) isn't strictly necessary:
  - could do intersection(s,create_set(x)) and test if the result is empty
  - could do delete(s,x) and see if we get an error message
  - but member(s,x) is much more convenient.
- Such choices affect functionality, convenience & efficiency
  - functionality: make sure all required operations are possible
  - convenience: make sure that typical/frequent operations are simple to use
  - efficiency: make frequent operations cheaper (usually by choosing an appropriate rep type - this should not affect the choice of abstraction)

Some requirements analysis is needed
- What data objects will be needed?
- What operations will need to be performed on them?
- What usage patterns are typical?
  - “use cases” / “scenarios” are helpful here
Object Orientation

→ Object Orientation extends data abstraction
  ⇨ Data abstraction becomes the main structuring mechanism for programs
    ➢ No fixed control structure
  ⇨ Object Oriented programming languages have:
    ➢ Abstraction
    ➢ Encapsulation - methods and objects are bundled together
    ➢ Polymorphism - same name can be used for different objects’ methods
    ➢ Dynamic binding - don’t know which method/object is referred to until runtime
    ➢ Inheritance - can extend existing data abstractions to create new ones

→ Use **OO design principles** in any programming language
  ⇨ Write data abstractions for all complex data structures
    ➢ Hide the implementations using ADTs or packages
    ➢ Only access the data abstractions through their defined operations (‘methods’)
  ⇨ Some OOP mechanisms are less important
    ➢ Polymorphism & dynamic binding are not relevant at the design level (these are programming tricks that make programs more complex)
    ➢ Inheritance can be done manually
Data Abstractions lead to good program design
- They help with encapsulation (information hiding)
- They help reduce the complexity of software interfaces
- They make programs more modifiable

Need some analysis to choose good data abstractions
- Adequacy: have you included all the operations that users need
- Can switch between implementations to improve efficiency

Data abstraction abstract data types
- ADTs are one way to implement data abstraction
- Can also use packages, objects, ...

Data abstraction object-oriented programming
- Data abstraction is really a design technique (the basis of OOD)
- Can use it in any programming language
- Some programming languages provide more support than others