Data types

A data type is characterized by:

- a set of *values*
- a *data representation,* which is common to all these values, and
- a set of *operations,* which can be applied uniformly to all these values

Abstract Data Types

- An Abstract Data Type (ADT) is:
  - a set of *values*
  - a set of *operations,* which can be applied uniformly to all these values
- To *abstract* is to leave out information, keeping (hopefully) the more important parts
  - What part of a Data Type does an ADT leave out?

Class defines a data type

- A class defines a *data type*
  - The possible *values* of a class are called objects
  - The *operations* on the objects are called methods
  - The *data representation* is all the fields that are contained within the object
- If there is no external access to the data representation, you have an *abstract data type*
- Sun's Java classes are (almost all) abstract data types

ADTs are better than DTs

- It is the responsibility of a class to protect its own data, so that objects are always in a valid state
  - Invalid objects cause program bugs
  - By keeping the responsibility in one place, it is easier to debug when invalid objects are present
- The less the user has to know about the implementation, the easier it is to use the class
- The less the user *does* know about the implementation, the less likely s\he is to write code that depends on it
- "Less is more"

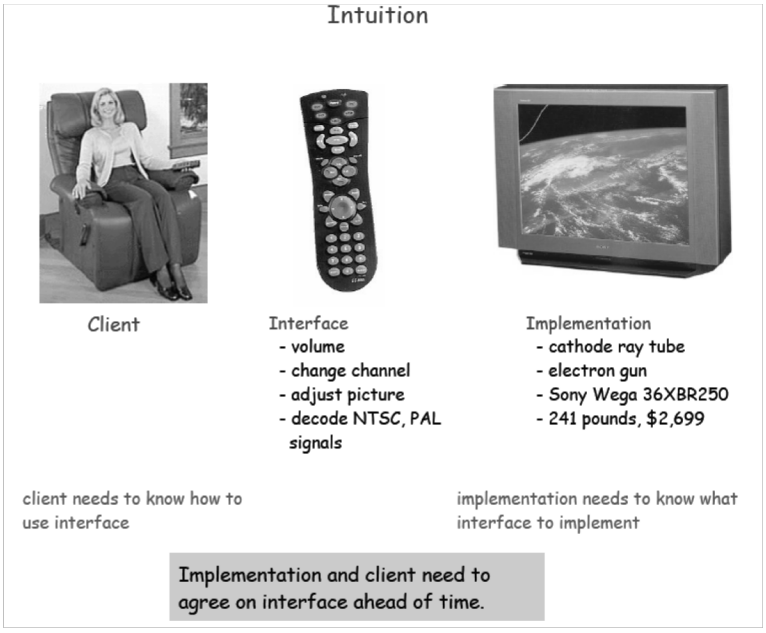
Abstract Data Types

- **Data type**: set of values and operations on those values.
- Ex: int, String, ComplexNumber, BigNumber, Card, Deck, Wave, Tour, . . .

- **Abstract data type** is a **data type** whose internal representation is hidden. They are language independent.

- Separate implementation from design specification.

- CLASS: provides data representation and code for operations.
- CLIENT: uses data type as black box.
- INTERFACE: contract between client and class.



Intuition

Client

Interface
- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation
- gas plasma monitor
- Pioneer PDP-502MX
- wall mountable
- 4 inches deep
- $19,995

client needs to know how to use interface

implementation needs to know what interface to implement

Can substitute better implementation without changing the client.

ADT Implementation in Java

Java ADTs.

- Keep data representation hidden with `private` access modifier.
- Define interface as operations having `public` access modifier.

```
public class Complex {
   private double re;
   private double im;

   public Complex(double re, double im) { . . . }
   public double abs()                  { . . . }
   public String toString()             { . . . }
   public Complex conjugate()           { . . . }
   public Complex plus(Complex b)       { . . . }
   public Complex times(Complex b)      { . . . }
}
```

Advantage: can switch to polar representation without changing client.

Note: all of the data types we have created are actually ADTs!

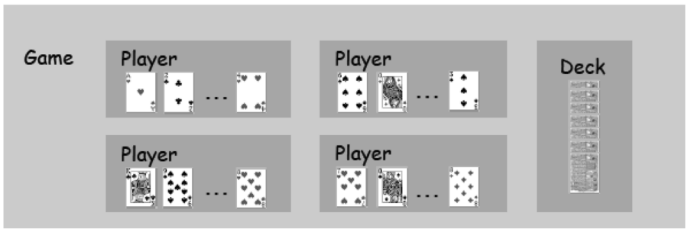
Modular Programming and Encapsulation

ADTs enable modular programming.

- Split program into smaller modules.
- Separate compilation.
- Different clients can share the same ADT.

ADTs enable encapsulation.

- Keep modules independent (include `main` in each class for testing).
- Can substitute different classes that implement same interface.
- No need to change client.



Abstract Data Types

1. **Constructors**: these operations are used whenever we want to create a new object of that particular type. Constructors are required as the creation of a new object belonging to an ADT requires a sequence of steps

2. **Destructors**: these are operations that are going to be executed when an object is removed from the system (because it is not used any longer)

3. **Inspectors**: these are operations that allows one to inspect the content of an object or test properties of the object (without modifying it)

4. **Modifiers**: these are operations that either modify an object or generate new objects.

Abstract Data Types

int is primitive integer type.

int is immutable, so it has no mutators.

- **creators**: the numeric literals 0, 1, 2, …
- **producers**: arithmetic operators +, -, ×, ÷
- **observers**: comparison operators ==, !=, <, >
- **mutators**: none (it's immutable)

Abstract Data Types

String is the string type.

String is immutable.

- **creators**: String constructors
- **producers**: concat, substring, toUpperCase
- **observers**: length, charAt
- **mutators**: none (it's immutable)

Contracts

- Every ADT should have a contract (or specification) that:
  - Specifies the set of valid values of the ADT
  - Specifies, for each operation of the ADT:
    - Its name
    - Its parameter types
    - Its result type, if any
    - Its observable behavior
  - Does *not* specify:
    - The data representation
    - The algorithms used to implement the operations

Importance of the contract

- A contract is an agreement between two parties; in this case
  - The implementer of the ADT, who is concerned with making the operations correct and efficient, *and also* with preserving the flexibility to make changes later
  - The applications programmer, who just wants to *use* the ADT to get a job done
- It does not matter if you are *both* of these parties; the contract is *still essential* for good code
- This separation of concerns is essential in any large project

Promise no more than necessary

- For a general API, the implementer should provide as much generality as feasible
  - This is the way c# classes are (mostly) written
- But for a specific program, the class author should provide only what is essential at the moment
  - In Extreme Programming terms, "You ain't gonna need it!"
  - In fact, XP practice is to *remove* functionality that isn't currently needed!
  - Your documentation should not expose anything that the application programmer does not need to know
- If you design for generality, it's easy to add functionality later—but removing it may have serious consequences

Implementing an ADT

- To implement an ADT, you need to choose:
  - a data representation that
    - must be able to represent all possible values of the ADT
    - should be private
  - a set of methods that support normal use of the ADT
    - The user must be able to create, possibly modify, and examine the values of the ADT
  - an algorithm for each of the possible operations that
    - must be consistent with the chosen representation
    - all auxiliary (helper) operations that are not in the contract should be private

Interfaces

- In many cases, an `interface` makes a better contract than a concrete implementation
- Why? Because the interface can only describe what the ADT does
  - It cannot contain fields, operators, instance constructors, or types.
  - It cannot provide an implementation.
  - Interfaces are not allowed to have constructors.

ADT Examples

Fraction

Let us make an example. We would like to create an ADT that represents fractions; the domain should be the set of all the possible fractions (since these are the values we are interested in manipulating), and the operations we would like to perform are:

- check if two fractions are equal
- read the numerator and denominator of the fraction
- simplify a fraction
- add and multiply fractions

The result should be a new data type (e.g., called **Fraction**), so that we can declare variables of this type and use them in our program.

Fraction

Domain of the ADT

We will try to follow a standard pattern in developing the specification of the ADT. The first component of the specification is the description of its domain. The domain is the set of values that we can store in objects of this type. For example:

Fraction ADT

DOMAIN: the set of all the possible fractions

Fraction

Constructor: we can assume one constructor that generates a brand new object of type fraction; the operation requires as input the numerator and denominator of the fraction we wish to create. The result should be the new fraction created. We will denote this as follows:

Fraction createFraction (int num, int den)

// creates a new fraction, having num as numerator and den as denominator

// the result is an object of type Fraction

Destructors: we will ignore this for the moment

Fraction

Inspectors: we will consider the following inspectors:

int getNumerator()

// the operation, applied to a fraction, returns the numerator of the fraction

int getDenominator()

// the operation, applied to a fraction, returns the denominator of the fraction

boolean isEqualTo (Fraction f)

Fraction

Modifiers: we will use the following modifiers:

```
// the operation, applied to a fraction, return true if the fraction has the same
// value as the fraction f, false otherwise
void simplifyFraction()
// this operation, applied to a fraction, simplifies the fraction to its normal form
void incrementFraction (Fraction f)
// this operation, applied to a fraction, change its value by adding to it the value
```

A Simple ADT: A Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The positions of the data items don't matter (unlike a list).
  - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
- The items do *not* need to be unique (unlike a set).
  - {7, 2, 10, 7, 5} isn't a set, but it is a bag

A Simple ADT: A Bag (cont.)

- The operations supported by our Bag ADT:
  - `add(item)`: add `item` to the `Bag`
  - `remove(item)`: remove one occurrence of `item` (if any) from the `Bag`
  - `contains(item)`: check if `item` is in the `Bag`
  - `numItems()`: get the number of items in the `Bag`
  - `grab()`: get an item at random, without removing it
    - reflects the fact that the items don't have a position (and thus we can't say "get the 5th item in the Bag")
  - `toArray()`: get an array containing the current contents of the bag
- Note that we *don't* specify *how* the bag will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}                    (see ~cscie119/examples/bag/Bag.java)
```

- An interface specifies a set of methods.
  - includes only the method headers
  - *cannot* include the actual method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class:

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        …
}
```

(see ~cscie119/examples/bag/ArrayBag.java)

- When a class header includes an `implements` clause, the class must define all of the methods in the interface.

Dictionary

1. A name or type, specifying a set of data (e.g. `Dictionary`).
2. Descriptions of all the operations (or methods) that do things with that type (e.g. `find, insert, remove`)

   The descriptions indicate *what* the operations do, not *how* they do it.

- A *data structure* is a systematic way of organizing and accessing data from a computer (e.g. array, linked list).

  Data structures are used to implement ADTs.

Dictionary

`find (key)`:	returns a record with the given key, or null if no record has the given key
`insert(key,data)`:	inserts a new record with given key and data `ERROR` if the dictionary already contains a record with the given key
`remove(key)`:	removes the record with the given key `ERROR` if there is no record with the given key

Dictionary

```
public interface Dictionary {

    public Object find(Object key);

    public void insert(Object key, Object data)
                    throws DuplicatedKeyException;

    public void remove(Object key)
                    throws NoKeyException;

}
```

Dictionary

```
public class LinkedListDictionary implements Dictionary {

    protected int size;
    protected DNode head, tail;

    public LinkedListDictionary() {

        size = 0;
        head = new DNode(null, null,null);
        tail = new DNode(null,null,null);
        head.setNext(tail);

    }
```

Dictionary

```
    public Object find(Object key) {
        if (size == 0) return null;
        else {
              ⋮
        }
    }
    public void insert (Object key, Object data) throws
            DuplicatedKeyException {
          ⋮
    }
```