Outline

From Data Type to ADT

Values

A **value** is a unit of information used in a program. It can be associated with a constant or variable (a name) by an assignment statement:

```
person = 'George'
n = 4
```

All primitive type values (integer, float, string) are represented internally in the computer program's memory as a series of zeros and ones.

More complex types have values which are combinations of more primitive types.

From Data Type to ADT

Data Types

A **Data Type** is the set of possible values which all represent the same type of information and share the same behavior. In Python, most data types are classes, and a value of some data type is an object in that class.

```
int
str
float
list
dict
file
```

Defining an ADT = Specification

Data Abstraction

The data is represented using **abstract attributes**.

Example: a `card` has attributes `suit` and `rank`.
The behavior is given by specifying functions, with the signature , preconditions, and postconditions of procedural abstraction.

Data Abstraction is the hiding of the primitive components comprising the values of some type, and hiding the implementation of the operations using that type.

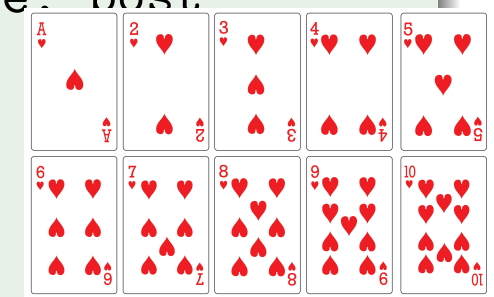
Abstract Data Type (**ADT**) is described by providing a specification for the data type, independent of any actual implementation, i.e. describes what operations are supported by the ADTs.

Defining an ADT = Specification

Example: playing card pages 41-42

ADT Card:
A simple playing card, characterized by:
rank: integer, in the range 1-13
suit: a character in ‘‘cdhs’’ for clubs, diamonds, hearts, and spades.

Operations:
create(rank, suit): create a new card; pre, post
suit(): card suit; pre, post ...
rank(): card rank; pre, post ...
suitName(): card suit name; pre, post ...
rankName(): card rank name; pre, post ...
toString(): string representation of card



Implementation of an ADT

From description to implementation

Given a description of ADT we can implement it.

see code in cardADT.py and test in test-cardADT.py

Concrete Representation

The abstract attributes are represented using types from the programming language or previously defined classes.

Example: a `suit` is now a member of the Python string class str, and is allowed to have the values `'c'`,`'d'`, `'h'` or `'s'`.

Class Specification

Python Classes

ADTs become **Python classes**, and their behaviors become **methods** for those classes. See Card_spec.py

Data Abstraction

In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter `'c'` corresponds to the suit `'clubs'`).

Functional Abstraction

Each method specification includes a comment listing all preconditions and postconditions.

Class Specification

Python Classes

ADTs become **Python classes**, and their behaviors become **methods** for those classes. See Card_spec.py

Data Abstraction

In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter `'c'` corresponds to the suit `'clubs'`).

Functional Abstraction

Each method specification includes a comment listing all preconditions and postconditions.

Class Specification

Python Classes

ADTs become **Python classes**, and their behaviors become **methods** for those classes. See Card_spec.py

Data Abstraction

In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter `'c'` corresponds to the suit `'clubs'`).

Functional Abstraction

Each method specification includes a comment listing all preconditions and postconditions.

Class Implementation

Data Representation

In the class definition, the constructor `__init__` will take parameters to set the attributes of new objects (or use default values).

Method Implementation

The actual Python code to implement the behavior of each method follows the comments listing preconditions and postconditions. Typical methods are **mutators**, which change attribute values, and **accessors** which return attribute values without changing them.

Class Implementation

Data Representation

In the class definition, the constructor `__init__` will take parameters to set the attributes of new objects (or use default values).

Method Implementation

The actual Python code to implement the behavior of each method follows the comments listing preconditions and postconditions. Typical methods are **mutators**, which change attribute values, and **accessors** which return attribute values without changing them.

Changing the Representation

The Abstraction Barrier

By keeping the specification (see Card-spec.py) and implementation (see Card.py) separate, it is possible to change the implementation—say, to use a more efficient algorithm—without having to change any program that uses the ADT respectful of its specification.

Changing the Representation

The Abstraction Barrier

The program which uses the ADT only has access to it through its methods, which are written to obey the specification—what types of parameters are passed and what types of values are returned. The concrete representation can change, but as long as the methods have the same signatures, and honors the same contracts (preconditions and postconditions), the program which calls them will still work.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

OOD and ADTs

Data Abstraction is only one of several ideas that have helped to advance software engineering.

OO design and programming uses ADTs as well as other principles.

Most OO gurus talk about three features that together make development truly object-oriented: encapsulation, polymorphism, and inheritance.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Encapsulation

Objects *know stuff* (data) and *do stuff* (operations). The process of packaging some data along with the set of operations that can be performed on the data is called encapsulation.

- also known as *information hiding*

- separates the issues of "what to do" from issues of "how to do" it.

- gives us implementation independence

Encapsulation alone makes the system *object-based* only.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Polymorphism

The word *polymorphism* means "many forms".

This is the principle that sending the same message (that is, calling the same method) to objects in different classes or different types should make the objects behave the same.

Example: recall cs1graphics library, where we can draw different shapes. Rectangle, circle, polygon, ... can be all drawn into a window. The behavior(result) of the draw operation is similar for all the drawable objects, but how it is performed is different for each of them.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Inheritance

Classes which share behaviors should not have to re-implement these behaviors if they can be **inherited** from a base class which implements that same behavior.

This principle promotes reuse of code, which in turn makes software more reliable, since bugs are more localized.

Terminology:
parent class - child class
superclass - subclass

Specifying the Deck Class

Sequential Collections

- A sequential collection is a container which allows one to traverse its objects sequentially.
- If the collection is not empty, it will have a first item.
- Each item (except the last) will have a next item after it.
- Starting from the first item, the entire collection is traversed by going to the next item until the last item is reached.
- A deck of cards is a sequential collection: the top card is the first, when the current card is removed, the next card is now at the top of the deck. The last card is at the bottom of the deck.

Specifying the Deck Class

Sorted Lists

- A sorted list is a homogeneous list where the items are increasing (each item is less than the next item) or decreasing (each item is greater than the next item).
- The items must be comparable: there is a binary operator (<) returning a boolean value.
- A deck of cards can be sorted, but for games, they are unsorted by shuffling.

Specifying the Deck Class

Problem: To Simulate a Deck of Cards

Provide a class whose objects will behave like a deck of cards: they can be shuffled and dealt to help simulate card games like poker or bridge.

Specifying the Deck Class

Objects

A `Deck` object will be a container class for `Card` objects, which have rank and suit attributes.

Specifying the Deck Class

Methods

- `shuffle` will ensure that dealing cards will produce a random sequence.
- `deal` will return a card from the deck, removing it from the deck in the process.

Implementing the Deck Class

Concrete Representation

Attributes:

- A Python list, `cards`, of `Card` objects, as defined in Chapter 2.

  Remark: An Abstract Data Type, when implemented, should only have attributes which are private, that is, with an underscore (_) as first character. The book does not do that here, which is unsafe. If a function outside the class has access to the concrete representation, then it will become broken if that representation changes, which is exactly what we want to avoid.

Implementing the Deck Class

Concrete Representation

Methods:

- __init__(self) creates a 52 Card deck.
- shuffle(self) prepares for random dealing by putting the list of Cards in random order.
- deal(self), returns a `Card` object, while removing it from the list `cards`.
- size(self) returns the number of cards remaining in the list. (See Deck.py in Chapter 3)

Implementing the Deck Class

Concrete Representation

```
def __init__(self):
    cards = []
    for suit in Card.SUITS:
        for rank in Card.RANKS:
            cards.append(Card(rank,suit))
    self.cards = cards
def shuffle(self):
    n = self.size()
    cards = self.cards
    for i,card in enumerate(cards):
        pos = randrange(i,n)
        cards[i] = cards[pos]
        cards[pos] = card
```