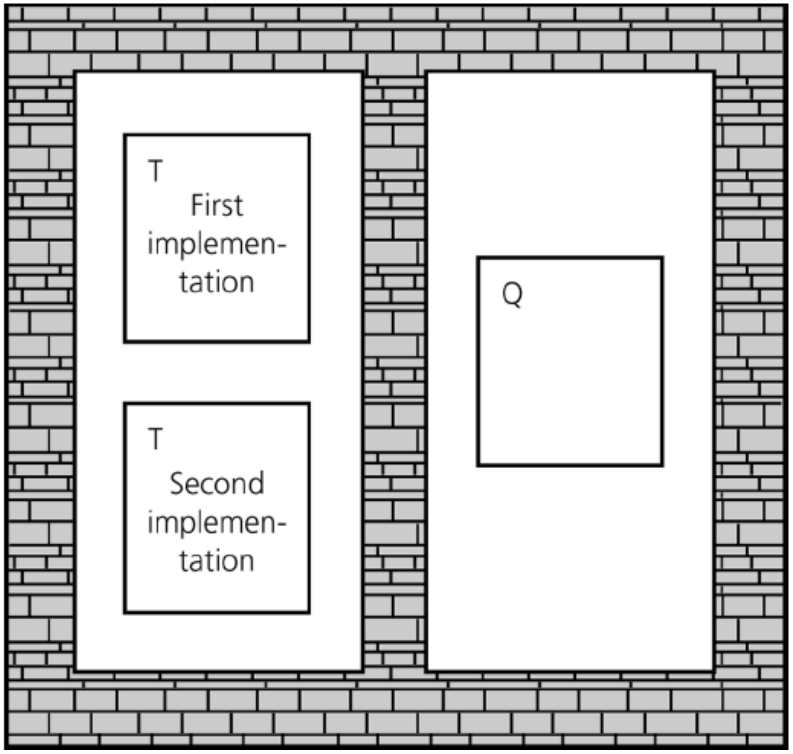
Abstract Data Types (ADTs): From Theory to Practice



A Very Quick History

- OO is both revolutionary and evolutionary
- OO evolved from Abstract Data Types (ADT)
- Grew out of the simulation and AI communities
  - Smalltalk
  - ModSIM
  - LISP with Flavors / CLOS
- Really took off when mainstream languages provided support
  - ADA95
    - The first standardized (ANSI) OO language
  - C++
    - Best of both worlds, structured programs in an OO language
  - Java
    - The latest bandwagon
- Complementing (or even replacing) “structured” and the high-tech version of “good”

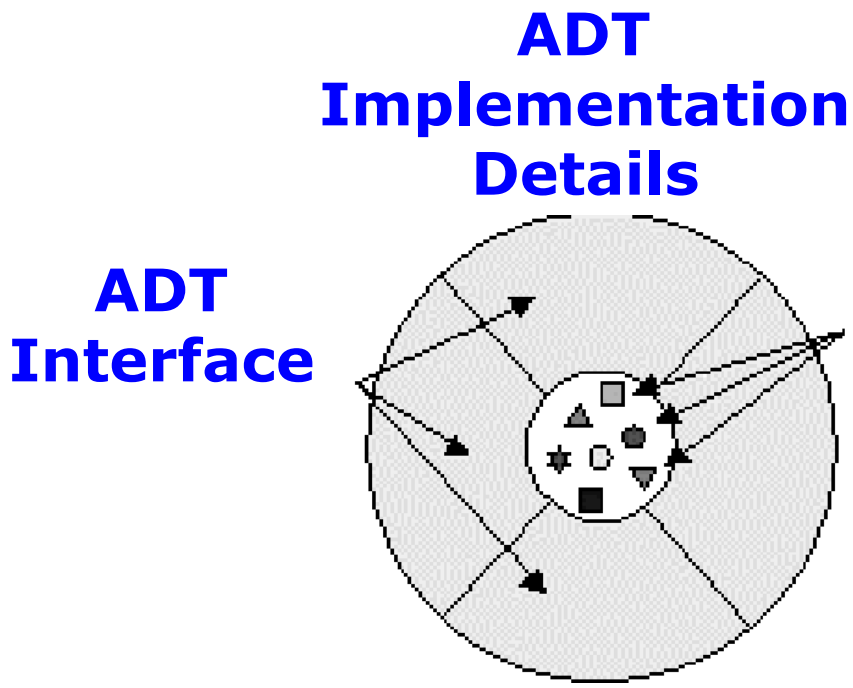
Modularity and Abstract Data Types (ADTs)

- An abstract data type (ADT) is characterized by:
  - a set of values
  - a set of operations

  It is not characterized by its data representation

- The data representation is private, so application code cannot access it:
  Only the operations can

- The data representation is changeable, with no effect on application code:
  Only the operations must be recoded



ADTs and Java

- There are various alternative conventions for expressing ADT contracts and implementations in Java. The main three of them are:
  1. Express the ADT's contract as an outline class declaration (showing only public members, and only heading of constructors and methods); express its implementation as a completed class declaration
  2. Express the ADT's contract as interface; express its implementation as a class that implements this interface
     - This convention makes the relationship between the contract and its implementation explicit in the source code
     - The disadvantage of this convention is that a Java interface does not allow us to specify the constructors and the attributes (state) of the ADT (although the implementing class can provide constructors and attributes)

ADTs and Java

❸ Express the ADT's contract as an abstract class; express its implementation as a class that extends this abstract class

- This convention fulfills the definition requirement of the ADT's state and behavior by specifying all the attributes and methods that characterize the ADT
- Abstract classes give a better ADT contract than Java interfaces since they provide state definition (through attributes) and specify all the methods (including constructors) of an ADT

Example: The Date ADT

- Class declaration:

```
public class Date {
 // Each Date value is a past, present, or future date
 // This date is represented by a year number y,a month
 // number m (1…12), and a day-in-month number d (1…31)
     public int y, m, d;
     public Date (int y, int m, int d) {
     // Construct a date with year y, month m, and day-
     // in-month d
        if (m < 1 || … )  throw …;
        this.y = y;  this.m = m;  this.d = d;
     }
```

Example: The Date ADT

```
public void advance (int n) {
 // Advance this date by n days (where n ≥ 0)
  int y = this.y, m = this.m, d = this.d + n;
  for (;;) {
        int last = …;  // no. of days in m, y
        if (d <= last)  break;
        d -= last;
        if (m < 12)  m++;
        else  { m = 1;  y++; }
      }
  this.y = y;  this.m = m;  this.d = d;
   }
}
```

Example: The Date ADT

- Possible application code:

```
Date today = new Date(2003, 2, 14);
today.advance(16);
System.out.println(today.y + '-' + today.m
                   + '-' + today.d);
```

This should print “2003-3-2”

Example: The `Date` ADT

- Problem: This data representation admits **improper** values (e.g., `m` = 0; or `m` = 2 and `d` = 30).
- Constructors and methods can (should) be coded to avoid improper values. E.g.:

```
Date today = new Date(2003, 2, 30);
```

will throw an exception

- But what if the data representation is accessed directly?

```
Date today = new Date(2003, 2, 14);
today.d += 16;
```

Example: The Date ADT Again

- A different data representation is possible:

```
public class Date {
// Each Date value is a past, present, or future date
// This date is represented by a day-in-epoch number
// d (where 0 represents 1 January 2000):
    public int d;
    public Date (int y, int m, int d) { … }
    public void advance (int n) { … }
}
```

- This makes advance faster, but Date() slower

Example: The `Date` ADT Again

- Recall existing application code:

```
Date today = new Date(2003, 2, 14);
today.advance(16);
System.out.println(today.y + '-' + today.m
     + '-' + today.d);
```

yields wrong value

fails to compile

fails to compile

Public vs. Private Data Representation

- If the data representation is public:
  - Application code might make improper values
  - Existing application code might be invalidated by change of representation
  - Loss of encapsulation

- If the data representation is private:
  - \+ Application code cannot make improper values
  - \+ Existing application code cannot be invalidated by change of representation
  - \+ Boosting encapsulation

The ADT `Date`

- Assumed application requirements:
  1. The values must be all past, present, and future dates
  2. It must be possible to construct a date from year number *y*, month number *m*, and day-in-month number *d*
  3. It must be possible to compare dates
  4. It must be possible to render a date in ISO format "*y*-*m*-*d*"
  5. It must be possible to advance a date by *n* days

ADT `Date` Operations

- A possible contract of the ADT `Date` defines the following allowable operations:
  - `Date(y, m, d): Int x Int x Int -> Date`
    - Note that after the arrow we write the type of the instance that this operation creates not the return type of the operation (Constructors do not return anything!)
  - `compareTo(D'): Date x Date -> Int`
    - In the operations except constructors the first type we write after the double dot is the type of the instance that calls the operation

    `toString(): Date -> String`

    `advance(n): Date x Int -> Date`
    - Note that there are two return types that operation `advance(n)` can provide: void and Date. If we want this operation to be a mutative transformer (i.e. to change the `Date` instance that calls the operation) then `advance(n)` must return void, else if we want this operation to be an applicative transformer (i.e. to create a new `Date` instance which is n days after the `Date` instance that calls the operation) then `advance(n)` must return Date

The Contract of the ADT Date

- The contract of ADT Date, expressed as an outline class definition:

```
public class Date {
// Each Date value is a past, present, or future date
  private …;
  public Date (int y, int m, int d);
  // Construct a date with year y, month m, and day-
  // in-month d
  public int compareTo (Date that);
  // Return –1 if this date is earlier than that,
  // or 0 if this date is equal to that,
  // or +1 if this date is later than that
  public String toString ();
  // Return this date rendered in ISO format.
  public void advance (int n);
  // Advance this date by n days (where n ≥ 0)
  }
```

Application Codes on ADT Date

- Possible application code:

```
Date today = …;
Date easter = new Date(2003, 4, 27);
today.advance(16);
if (today.compareTo(easter) < 0)
   System.out.println(today.toString());
```

- Impossible application code:

```
today.d += 16;
System.out.println(today.y + '-' + today.m
      + '-' + today.d);
```

fails to compile

fails to compile

fails to compile

Implementing the Contract of the ADT Date

```
public class Date {
  // Each Date value is a past, present, or future date
  // This date is represented by a year number y, a
  // month number m, and a day-in-month number d:
  private int y, m, d;
  public Date (int y, int m, int d) {
  // Construct a date with year y, month m, and day-in-
  //month d
     this.y = y; this.m = m; this.d = d;
  }
```

Implementing the Contract of the ADT `Date`

```
public int compareTo (Date that) {
// Return -1 if this date is earlier than that,
// or 0 if this date is equal to that,
// or +1 if this date is later than that
    return (this.y < that.y ? -1 :
            this.y > that.y ? +1 :
            this.m < that.m ? -1 :
            this.m > that.m ? +1 :
            this.d < that.d ? -1 :
            this.d > that.d ? +1 : 0);
    }
public String toString () {
    // Return this date rendered in ISO format.
       return (this.y + '-' + this.m + '-'
            + this.d);
    }
    public void advance (int n) {
    // Advance this date by n days (where n ≥ 0).
      …
    }
  }
```

complicated

Implementing the Contract of the ADT `Date` Again

```
public class Date {
 // Each Date value is a past, present, or future date
 // This date is represented by a day-in-epoch number
 // d(where 0 represents 1 January 2000):
 private int d;
 public Date (int y, int m, int d) {
 // Construct a date with year y, month m, and day-in-
 // month d
     …;
     this.d = …;
}
```

complicated

Implementing the Contract of the ADT `Date` Again

```
public int compareTo (Date that) {
// Return –1 if this date is earlier than that,
// or 0 if this date is equal to that,
// or +1 if this date is later than that
  return (this.d < that.d ? -1 :
                this.d > that.d ? +1 : 0);
}
public String toString () {
// Return this date rendered in ISO format
   int y, m, d;
   …;
   return (y + '-' + m + '-' + d);
}
public void advance (int n) {
// Advance this date by n days (where n ≥ 0).
   this.d += n;
}
}
```

complicated

ADT Design

- Operations are sufficient if together they meet all the ADT's requirements
  - Can the application be written entirely in terms of calls to these operations?
- An operation is necessary if it is not surplus to the ADT's requirements
  - Could the operation be safely omitted?
- A well-designed ADT provides operations that are necessary and sufficient for its requirements

ADT Design

- A constructor is an operation that creates a value of the ADT
- An accessor (selector) is an operation that uses a value of the ADT to compute a value of some other type
  - An observer uses a value of the ADT to compute a Boolean value
- A transformer (mutator) is an operation that computes a new value of the same ADT. A transformer is:
  - mutative if it overwrites the old value with the new value
  - applicative if it returns the new value, without overwriting the old value
- The values of an ADT are:
  - mutable if the ADT provides at least one mutative transformer
  - immutable if the ADT provides no mutative transformer
- A well-designed ADT provides at least one constructor, at least one accessor, and at least one transformer
  - The constructors and transformers together can generate all values of the ADT

Design of the ADT `Date`

- Recall the `Date` contract:

```
public class Date {
   private …;
   public Date (int y, int m, int d);
   public int compareTo (Date that);
   public String toString ();
   public void advance (int n);
}
```

- These operations are sufficient
- All these operations are necessary

Design of the ADT `Date`

- Consider another possible `Date` contract:

```
public class Date {
   private …;
   public Date (int y, int m, int d);
   public int getYear ();
   public int getMonth ();
   public int getDay ();
   public void advance (int n);
}
```

- These operations are sufficient
  - Date comparison and rendering are clumsier, but still possible
- All these operations are necessary

Design of the ADT `Date`

- Consider yet another possible `Date` contract:

```
public class Date {
   private …;
   public Date (int y, int m, int d);
   public int compareTo (Date that);
   public String toString ();
   public void advance (int n);
   public void advance1Day ();
}
```

- Operation `advance1Day` is unnecessary

Design of the ADT `Date`

- Recall our first `Date` contract:

```
public class Date {
   private …;
   public Date (int y, int m, int d);
   public int compareTo (Date that);
   public String toString ();
   public void advance (int n);
}
```

constructor

accessor

accessor

mutative transformer

Design of the ADT `Date`

- Consider another possible `Date` contract:

```
public class Date {
   private …;
   public Date (int y, int m, int d);
   public int compareTo (Date that);
   public String toString ();
   public Date plus (int n);
}
```

constructor

accessor

accessor

applicative transformer

Method Signature

- As we know, a method can be defined to accept zero or more formal parameters
- The name of the method together with the list of its formal parameters is called the signature of the method
- We can declare exclusively a method using its signature

```
public int length();
public String substring (int i, int j);

int_length // identification of method length
String_substring_int_int // identification of method
                         // substring
```

Preconditions & Postconditions

- A pair of statements about a method
  - The precondition statement indicates what must be true before the method is called
  - The postcondition statement indicates what will be true when the method finishes its work
- Preconditions and postconditions specify what a method accomplishes
- The programmer who calls the method is responsible for ensuring that the precondition is valid when the method is called
- The programmer who writes the method counts on the precondition being valid, and ensures that the postcondition becomes true at the method's end
- We can define an ADT using the preconditions, postconditions and signatures of its methods in an outline class definition

The contract of the ADT Date again using preconditions and postconditions

```
public class Date {
  // Each Date value is a past, present, or future
  // date. This date is represented by a year number
  // y, a month number m, and a day-in-month number d
  private …;
  /////////// Constructors //////////
  // Constructor: Date
  // Precondition:
  // a) y ≥ 1
  // b) 1 ≤ m ≤ 12
  // c) if m in (1, 3, 5, 7, 8, 10, 12) then 1 ≤ d ≤ 31
  // d) if m in (4, 6, 9, 11) then 1 ≤ d ≤ 30
  // e) if m == 2 and y a leap year then 1 ≤ d ≤ 29, else 1≤d≤28
  // Postcondition:
  // Constructs a valid Date instance with year y, month m and
  // day-in-month d
  public Date (int y, int m, int d);
```

The contract of the ADT Date again using preconditions and postconditions

```
////////////// Accesors //////////
// Method: compareTo
// Precondition:
// Date that is a valid instance of Date
// Postcondition:
// Returns -1 if this date is earlier than that,
// or 0 if this date is equal to that,
// or +1 if this date is later than that
public int compareTo (Date that);
// Method: toString
// Precondition:
// Postcondition:
// Returns this date rendered in ISO format (d-m-y)
public String toString ();
```

The contract of the ADT `Date` again using preconditions and postconditions

```
////////////// Transformers //////////
// Method: advance
// Precondition: n ≥ 0
// Postcondition:
// Advances this date by n days
public void advance (int n);
}
```

The ADT String

- A string is a sequence of characters
- The characters have consecutive indices
- A substring of a string is a subsequence of its characters
- The length of a (sub)string is its number of characters
- The empty string has length zero
- Assumed application requirements:
  1. The values are to be strings of any length
  2. It must be possible to determine the length of a string
  3. It must be possible to obtain the character at a given index
  4. It must be possible to obtain the substring at a given range of indices
  5. It must be possible to compare strings lexicographically
  6. It must be possible to concatenate strings

ADT (Immutable) `String` Operations

- The operations defined in a possible contract of ADT `String` are:

```
{ String(cs): Char[] -> String
} length(): String -> Int
  charAt(i): String x Int -> Char
  equals(S'): String x String -> Boolean
  compareTo(S'): String x String -> Int
  substring (i, j): String x Int x Int -> String
  concat (S'): String x String -> String
```

The contract of the ADT (Immutable) `String` again using preconditions and postconditions

```
public class String {
 // Each String value is an immutable string of
 // characters, of any length, with indices
 // starting at 0
 private …;
 /////////// Constructors //////////
 // Constructor: String
 // Precondition: cs[] != null
 // Postcondition:
 // Constructs a valid String instance consisting of
 // all the chars in cs
```

The contract of the ADT (Immutable) `String` again using preconditions and postconditions

```
public String String(cs);
////////// Accessors //////////
// Method: length
// Precondition:
// Postcondition: Returns the length of this string
public int length ();
// Method: charAt
// Precondition: 0 ≤ i ≤ length() – 1
// Postcondition:
// Returns the character at index i in this string
public char charAt (int i);
```

The contract of the ADT (Immutable) `String` again using preconditions and postconditions

```
// Method: equals
// Precondition: String that is a valid String instance
// Postcondition:
// Returns true if this string is equal to that, else
// it returns false
public boolean equals (String that);
// Method: compareTo
// Precondition: String that is a valid String instance
// Postcondition:
// Returns -1 if this string is lexicographically
// less than that, or 0 if this string is equal to
// that, or +1 if this string is lexicographically
// greater than that
public int compareTo (String that);
```

The contract of the ADT (Immutable) `String` again using preconditions and postconditions

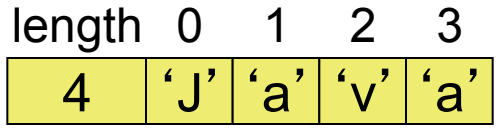
```
////////// Transformers //////////
// Method: substring
// Precondition: 0 ≤ i < j ≤ length()
// Postcondition:
// Returns the substring of this string consisting of
// the characters whose indices are i, …, j-1
public String substring (int i, int j);
// Method: concat
// Precondition: String that is a valid String instance
// Postcondition:
// Returns the string obtained by concatenating this
// string and that
public String concat (String that);    }
```

applicative transformer

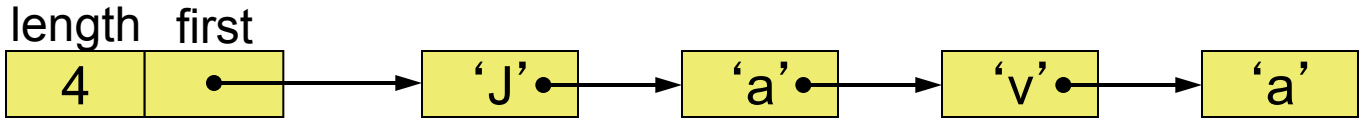
applicative transformer

Immutable Strings: Implementations

- Represent a string by its length *n* together with an array of exactly *n* characters, e.g.:



- Or represent a string by its length *n* together with an SLL of characters, e.g.:



- The array representation is much better, since these strings are immutable, we never insert or delete characters