

CS2040 Data Structures and Algorithms

Lecture Note #2

Abstract Data Type

Outline

1. Abstract Data Type

1.1 Data Structure

1.2 Understanding ADT

2. Java Interface

2.1 Using Java interface to define ADT

2.2 Making an interface → FracADT Interface

2.3 Fraction class: Variable based Implementation

2.4 FractionArr class: Array based Implementation

2.5 Using interface as a type

1 Abstract Data Type

Collection of data + set of operations
on the data

1.1 Data Structure (1)

- ❑ **Data structure** is a construct that can be defined within a programming language to store a collection of data
 - **Arrays**, which are built into Java, are data structures
 - We can create other data structures. For example, our fraction class from lecture 1 is a data structure.

```
class FractionV1 {  
    public int num, denom;  
  
    public FractionV1(int iNum, int iDenom) {  
        num = iNum;  
        denom = iDenom;  
    }  
  
    public int getNum() { return num; }  
    public int getDenom() { return denom; }  
    ...  
}  
...  
FractionV1[] fList = new FractionV2[100];
```

1.1 Data Structure (2)

- We can also implement a fraction differently as follows:

```
class FractionV2 {  
    public int[] members; // index 0 is num, index 1 is denom  
    public static final int num = 0;  
    public static final int denom = 1;  
  
    public FractionV2(int iNum, int iDenom) {  
        members = new int[2];  
        members[num] = iNum;  
        members[denom] = iDenom;  
    }  
  
    public int getNum() { return members[num]; }  
    public int getDenom() { return members[denom]; }  
    ...  
}
```

1.1 Data Structure (3)

- ❑ Even though both FractionV1 and FractionV2 implement the same entity which is a Fraction we cannot do the following

```
FractionV1 f1 = new FractionV1(1,2);  
FractionV2 f2 = f1; // ← Compilation error here
```

- ❑ Since they are two different data types.

1.2 Abstract Data Type (ADT)

- ❑ An **Abstract Data Type (ADT)** is then looking at data types independent of the implementation details
- ❑ An **ADT** represent a **collection of data** together with a **specification of a set of operations (functional abstraction)** on the data
 - Functional abstraction → Indicate **what** ADT operations do, **not** **how** to implement them
 - Also does not specify how the data is to be stored



- **Data structures/algorithms** is then the **how to implement them** part

Real Life Analogy of ADT

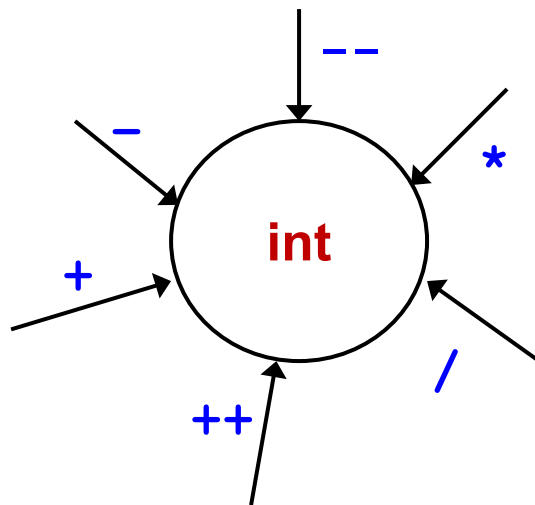


All cars have a common interface

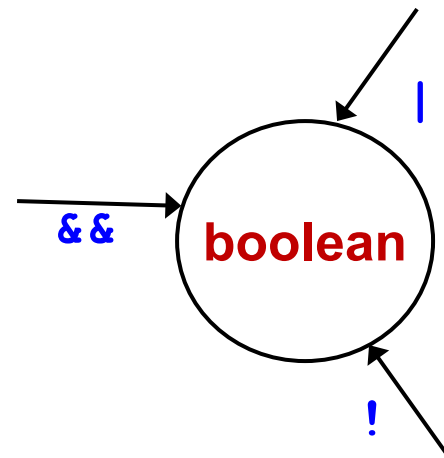
- Steering wheel
- Gear box (manual or auto)
- Pedals (acceleration, brake, clutch)
- ...

Primitive Types as ADTs

- Java's predefined data types can be viewed as ADTs
- Representation details are hidden which aids ease of usage and portability
- Examples: **int**, **boolean**, **double**



int type with the operations
(e.g.: `--`, `/`) defined on it.



boolean type with the operations
(e.g.: `&&`) defined on it.

2 Java Interface

Specifying related methods

2.1 Java Interface to define ADT

- Java interfaces provide a way to specify a **common set of operations** for possibly unrelated classes and can be used to specify an ADT
- Java **interface**
 - uses the keyword **interface**, rather than **class**
 - specifies methods to be implemented
 - A Java interface is a group of related methods with empty bodies (before Java 8 ...)
 - can have constant definitions (which are implicitly **public static final**)
- A class is said to implement the interface if it provides implementations for **ALL** the methods in the interface

2.2 Making an Interface → FracADT

- There can be many ways to implement class representing a positive fraction
- We can make it an ADT by specifying an interface
- Let's call the interface FracADT (the design here makes implementations of FracADT immutable classes)

FracADT.java

```
public interface FracADT {  
    public int getNum();    //returns numerator part  
    public int getDenom(); //returns denominator part  
    public void setNum(int iNum); //sets new numerator  
    public void setDenom(int iDenom); //sets new denominator  
  
    public FracADT add(FracADT f);    //returns this + f  
    public FracADT minus(FracADT f); //returns this - f  
    public FracADT times(FracADT f);  //returns this * f  
    public FracADT divide(FracADT f); //returns this / f  
    public FracADT simplify(); //returns simplified version  
}
```

2.3 Implementing the FracADT

- Two possible ways of implementing it
 - Using 2 variable to store the numerator/denominator (we have done this)
 - Using an array of size 2 to store the numerator/denominator
- This results in two possible classes

2.3 Fraction class – variable based (1)

- Skeleton program for Fraction.java

Fraction.java

```
class Fraction implements FracADT {
    public int num;
    public int denom;

    // Constructors
    public Fraction() {
        this(1,1); // calls the other constructor
    }
    public Fraction(int iNum, int iDenom) {
        setNum(iNum);
        setDenom(iDenom);
    }

    // Accessors
    public int getNum() { return num; }
    public int getDenom() { return denom; }

    // Mutators
    public void setNum(int iNum) { num = iNum; }
    public void setDenom(int iDenom) { denom = iDenom; }
```

2.3 Fraction class – variable based (2)

Fraction.java

```
// Fill in the code for all the methods below
public FracADT simplify() {
    int divisor = gcd(num,denom);
    Fraction result = new Fraction(num/divisor,denom/divisor);

    return result;
}
public FracADT add(FracADT f) { /* fill in the code */}
public FracADT minus(FracADT f) { /* fill in the code */}
public FracADT times(FracADT f) { /* fill in the code */}
public FracADT divide(FracADT f) { /* fill in the code */}

// Overriding methods toString() and equals()
public String toString() { /* fill in the code */}
public boolean equals(Object obj) { /* fill in the code */}

// Returns greatest common divisor of a and b
public static int gcd(int a, int b) { /* fill in the code */}
}
```

2.4 FractionArr class – Array based

■ Skeleton program for FractionArr.java

FractionArr.java

```
class FractionArr implements FracADT {
    public int[] members; // index 0 is num, index 1 is denom
    public static final int num = 0;
    public static final int denom = 1;

    //Constructor - note we don't have the default constructor here
    public FractionArr(int iNum, int iDenom) {
        members = new int[2];
        setNum(iNum);
        setDenom(iDenom);
    }

    // Accessors
    public int getNum() {return members[num];}
    public int getDenom() {return members[denom];}

    // Mutators
    public void setNum(int iNum) {members[num] = iNum;}
    public void setDenom(int iDenom) {members[denom] = iDenom;}

    // The rest are omitted here
}
```


2.5 Interface can be used as a type

- Each interface is compiled into a separate bytecode file, just like a regular class
 - We **cannot create an instance of an interface**, but we can use an interface as a data type for a variable, or as a result of casting

```
public boolean equals (Object obj) {  
    if (obj instanceof FracADT) {  
        FracADT temp1 = ((FracADT) obj).simplify(); // result of casting  
        FracADT temp2 = simplify();  
        return ((temp1.getNum() == temp2.getNum()) &&  
                (temp1.getDenom() == temp2.getDenom()));  
    }  
    return false;  
}
```

Summary

- We learn about the need for ADTs
- We learn about using **Java Interface** to define an ADT
- With this, we will learn and define various kinds of ADTs/data structures in subsequent lectures

End of file