CSE 331
Software Design & Implementation

Spring 2022
Section 3 – ADTs, AFs, and HW3
Administrivia

• HW2 due yesterday (4/13) at 11PM!

• HW3 due next Wednesday (4/20) at 11PM!

• Any questions?
Abstract Data Types (ADTs)

• Abstraction representing some set of data
  – Meant to express the meaning/concept behind some Java class

• Different from implementation/Java fields!
  – Same ADT can have many different implementations
Abstract data types by example

Review ADT concepts through two examples:
* A Line ADT
* A Rectangle ADT

On the course website, see “Resources” → “Class and Method Specifications” for a handy guide with full details.
Abstraction Functions (AFs)

• Let’s say we have an ADT
  – And we choose some way to implement it

• How does the concrete implementation relate to our ADT?

• This is an abstraction function
  – Maps object implementation (our Java fields) to the abstract state
  – Ex: “How does a Triangle object from Triangle.java represent a Triangle ADT?”
  – Note: specific to implementation
Diagram

ADT specification

Abstract States

Abstraction Barrier

Fields in our Java class

Abstraction function (AF): Relationship between ADT specification and implementation
Line ADT

Concept: A line segment in the Cartesian co-ordinate plane
Line ADT: Class specification

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */

public class Line {
    // rep invariant, fields, methods, etc.
}

// Diagram of a line segment on an x-y axis
/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */

public class Line {
    // Abstract state is _____
    private Point p1, p2;
}

What is our abstraction function?
/**
 * A Line is a mutable 2D line segment with endpoints p1 and p2.
 */
public class Line {
    // Abstract state is line with endpoints p1 and p2
    private Point p1, p2;
}
Line ADT: Representation #2

```java
/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
   // Abstract state is ____
   private int x1, x2;
   private int y1, y2;
}
```

What is our abstraction function?
/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
    // Abstract state is line with endpoints (x1, y1) and (x2, y2)
    private int x1, x2;
    private int y1, y2;
}
/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */

public class Line {
    // Abstract state is _____
    private int x1, y1;
    private double angle;
    private double len;
}

What is our abstraction function?
Line ADT: Representation #3

/**
 * A Line is a mutable 2D line segment with endpoints p1 and p2.
 */

public class Line {
   // Abstract state is line with endpoints (x1, y1) and
   // (x1 + len * cos(angle), y1 + len * sin(angle))
   private int x1, y1;
   private double angle;
   private double len;
}

Does this representation have any advantages over #1?
Try it yourself!

Write your own specification of a Rectangle ADT on the handout.

Then give two different possible representations for your Rectangle ADT and write abstraction functions for them.
HW3

In HW3, you will be writing methods in the `Natural` class

Let’s look at the specification:

```java
/**
 * Represents an immutable, non-negative integer value
 * along with a base in which to print its digits, which we
 * can think of as a pair (base, value).
 * For example, (2, 5) represents the integer 5 (in decimal),
 * but it will show its digits as 101 (in binary) when
 * printed.
 *
 * We require that the base is at least 2 and at most 36 for
 * simplicity.
 */
public class Natural { ... }
```
Different Base Examples

Let’s take the value 10. We can use the constructor:

public Natural(int base, int value){…}

new Natural(10, 10) => “10”
new Natural(2, 10) => “1010”
new Natural(3, 10) => “101”
new Natural(4, 10) => “22”
Now, let’s look at the fields, RI, and AF:

// Shorthand: b = this.base, D = this.digits, and
//           n = this.digits.length
//
// RI: 2 <= b <= 36 and D != null and n >= 1 and
//     if n > 1, then D[n-1] != 0 (no leading zeros) and
//     for i = 0 .. n-1, we have 0 <= D[i] < b
//
// AF(this) = (b, D[0] + D[1] b + D[2] b^2 + ... +
//           D[n-1] b^{n-1})

private final int base;
private final int[] digits;

new Natural(2, 10) => [0, 1, 0, 1] => “1010”
leftShift()

Now let’s take a look at the left shift method:

/**
 * Produces a number whose digits, in this base, are the result of taking the
digits of this number and shifting them to the left m positions, writing
zeros in the now empty positions.
 * @return (this.base, this.value * this.base^m)
*/
public Natural leftShift(int m) { ... }

How do we multiply something by 10 in base-10? Add a zero

How do we multiply something by 2 in binary? Add a zero

How do we multiply something by 100 (10^2) in decimal? Add two zeroes

What’s the pattern? How can we do this in our code?
leftShift()

Now let’s take a look at the left shift code:

```java
public Natural leftShift(int m) {
    int[] digits = new int[this.digits.length + m];
    System.arraycopy(this.digits, 0, digits, m, this.digits.length);
    return new Natural(this.base, digits);
}
```

```plaintext
new Natural(10, 36) => [6,3] => leftShift(2) => ?
```
leftShift()

Now let’s take a look at the left shift code:

```java
public Natural leftShift(int m) {
    int[] digits = new int[this.digits.length + m];
    System.arraycopy(this.digits, 0, digits, m, this.digits.length);
    return new Natural(this.base, digits);
}
```

new Natural(10, 36) => [6,3] => leftShift(2)
=> [0,0,6,3] = (10,3600)
leftShift() 

Now let’s take a look at the left shift code:

```java
public Natural leftShift(int m) {
    int[] digits = new int[this.digits.length + m];
    System.arraycopy(this.digits, 0, digits, m, this.digits.length);
    return new Natural(this.base, digits);
}
```

```
new Natural(10, 36) => [6,3] => leftShift(2)
=> [0,0,6,3] = (10,3600)
```

```
new Natural(2, 10) => [0,1,0,1] => leftShift(3)
=> ?
```
leftShift()

Now let’s take a look at the left shift code:

```java
public Natural leftShift(int m) {
    int[] digits = new int[this.digits.length + m];
    System.arraycopy(this.digits, 0, digits, m, this.digits.length);
    return new Natural(this.base, digits);
}
```

new Natural(10, 36) => [6,3] => leftShift(2)
=> [0,0,6,3] = (10,3600)

new Natural(2, 10) => [0,1,0,1] => leftShift(3)
=> [0,0,0,0,1,0,1] = (2,80)

Does this make sense?
public int getValue() {
    int i = this.digits.length - 1;
    int j = 0;
    int val = this.digits[i];

    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        j = j + 1;
        i = i - 1;
        val = val * this.base + this.digits[i];
    }

    return val;
}
Proving `getValue()`

Let’s first prove that the invariant is established before the loop:

```java
public int getValue() {
    {{ RI, which includes n >= 1 }}
    int i = this.digits.length - 1;
    {{ ? }}
    int j = 0;

    int val = this.digits[i];

    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //     i + j = n - 1, where D = this.digits,
    //     n = this.digits.length, and b = this.base

    ... 
}
```
Proving `getValue()`

Let’s first prove that the invariant is established before the loop:

```java
public int getValue() {
    {{ RI, which includes n >= 1 }}
    int i = this.digits.length - 1;
    {{ n >= 1 and i = n - 1 }}
    int j = 0;
    {{ ? }}
    int val = this.digits[i];

    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    // i + j = n - 1, where D = this.digits,
    // n = this.digits.length, and b = this.base
    ...
}
```
Proving `getValue()`

Let’s first prove that the invariant is established before the loop:

```java
public int getValue() {
    {{ RI, which includes n >= 1 }}
    int i = this.digits.length - 1;
    {{ n >= 1 and i = n - 1 }}
    int j = 0;
    {{ n >= 1 and i = n - 1 and j = 0}}
    int val = this.digits[i];
    {{ ? }}

    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //     i + j = n - 1, where D = this.digits,
    //     n = this.digits.length, and b = this.base
    ...
}
```
Let’s first prove that the invariant is established before the loop:

```java
public int getValue() {
    {{ RI, which includes n >= 1 }}
    int i = this.digits.length - 1;
    {{ n >= 1 and i = n - 1 }}
    int j = 0;
    {{ n >= 1 and i = n - 1 and j = 0 }}
    int val = this.digits[i];
    {{ n >= 1 and i = n - 1 and j = 0 and val = D[i] }}

    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //     i + j = n - 1, where D = this.digits,
    //     n = this.digits.length, and b = this.base
    ...
}
```

Does this imply the invariant?
Let's prove the part after the loop:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        ...
    }
    return val;
}
```
Proving `getValue()`

Let’s prove the part after the loop:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        ...
    }
    {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
      and j = n-1 }}
    ⇔ {{ ? }}
    
    return val;
}
Proving `getValue()`

Let's prove the part after the loop:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        ...
    }
    {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
        and j = n-1 }}
    ⇔ {{ val = D[0] b^0 + D[1] b^1 + ... + D[n-1] b^{n-1} and i=0
        and j = n-1 }}

    return val;
}
```
Now let's prove the loop body:

```java
public int getValue() {
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //     i + j = n - 1, where D = this.digits,
    //     n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        j = j + 1;
        i = i - 1;
        val = val * this.base + this.digits[i];
    }
}
```
Proving `getValue()`

Now let's prove the loop body:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        {{ val = D[i]b^0 + D[i+1]b^1 + ... + D[n-1]b^j and i+j = n-1
            and j != n–1 } }
        j = j + 1;
        {{ ? } }
    }
    {{ ? } }
    i = i - 1;
    val = val * this.base + this.digits[i];
    }
    ...
```
Proving `getValue()`

Now let's prove the loop body:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
          and j != n–1 }}
        j = j + 1;
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^{j-1} and i+j-1 = n-1
          and j != n }}
        i = i - 1;
        {{ ? }}
        val = val * this.base + this.digits[i];
    }
    ...
}
```
Proving `getValue()`

Now let’s prove the loop body:

```java
public int getValue() {

    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n – 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
          and j != n–1 }}

        j = j + 1;
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^{j-1} and i+j-1 = n-1
          and j != n }}

        i = i - 1;
        {{ val = D[i+1] b^0 + D[i+2] b^1 + ... + D[n-1] b^{j-1} and i+j = n-1
          and j != n }}

        val = val * this.base + this.digits[i];
        {{ ? }}
    }

    ...
}
```
Proving `getValue()`

Now let’s prove the loop body:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //     i + j = n - 1, where D = this.digits,
    //     n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
            and j != n–1 }}
        j = j + 1;
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^{j-1} and i+j-1 = n-1
            and j != n }}
        i = i - 1;
        {{ val = D[i+1] b^0 + D[i+2] b^1 + ... + D[n-1] b^{j-1} and i+j = n-1
            and j != n }}
        val = val * this.base + this.digits[i];
        {{ (val – D[i])/b = D[i+1] b^0 + D[i+2] b^1 + ... + D[n-1] b^{j-1}
            and i+j = n-1 and j != n }}
        ⇔ {{ ? }}
    }
    ...
}
```
Proving `getValue()`

Now let's prove the loop body:

```java
public int getValue() {
    ...
    // Inv: val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and
    //      i + j = n - 1, where D = this.digits,
    //      n = this.digits.length, and b = this.base
    while (j != this.digits.length - 1) {
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1
          and j != n-1 }}
        j = j + 1;
        {{ val = D[i] b^0 + D[i+1] b^1 + ... + D[n-1] b^{j-1} and i+j-1 = n-1
          and j != n }}
        i = i - 1;
        {{ val = D[i+1] b^0 + D[i+2] b^1 + ... + D[n-1] b^{j-1} and i+j = n-1
          and j != n }}
        val = val * this.base + this.digits[i];
        {{ (val - D[i])/b = D[i+1] b^0 + D[i+2] b^1 + ... + D[n-1] b^{j-1}
          and i+j = n-1 and j != n }}
        ⇔ {{ val = D[i] + D[i+1] b^1 + ... + D[n-1] b^j and i+j = n-1 and j != n }}
    }
    ... It's correct!
}
Starter Code

Let’s end with skimming through the starter code and run the tests…