Advanced Object-oriented Programming

Lecture 8

Queues
public class CardHand {
    public boolean isEmpty() {
    }
    public void add(BandCard b) {
    }
    public BandCard getTop() {
    }
    public void removeTop() {
    }
}
We need a **data representation** for a class CardHand: some way of storing the list of cards, in such a way that we can deliver the required functionality.

Arrays would seem a reasonable starting point:

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

\[\downarrow\]

```
 b1  b2  b3  b4  b5
```
We need a **data representation** for a class CardHand: some way of storing the *list* of cards, in such a way that we can deliver the required functionality.

**Arrays** would seem a reasonable starting point:

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

\[\downarrow\]

```
b1 | b2 | b3 | b4 | b5
```
We need a data representation for a class CardHand: some way of storing the list of cards, in such a way that we can deliver the required functionality.

Arrays would seem a reasonable starting point:

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

\[
\downarrow
\]

| b1 | b2 | b3 | b4 | b5 |
We need a data representation for a class CardHand: some way of storing the list of cards, in such a way that we can deliver the required functionality.

Arrays would seem a reasonable starting point:

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

\[\downarrow\]

```
b1  b2  b3  b4  b5
```
Data Representation

add(add(add(add(empty, b1), b2), b3), b4), b5)

...This allows us to store a list of values, and makes `getTop()` easy:

```
return theArray[0];
```
Bearing in mind the Stack example, we may add a pointer to show the last position where a card was added (the bottom of the hand):

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

\[ \downarrow \text{pointer} \]

\[
\begin{array}{ccccc}
\text{b1} & \text{b2} & \text{b3} & \text{b4} & \text{b5} \\
\end{array}
\]

This makes add straightforward:

```
add(BandCard b) body
\ntheArray[++pointer] = b;
```
Data Representation

Bearing in mind the Stack example, we may add a pointer to show the last position where a card was added (the bottom of the hand):

\[
\text{add}(\text{add}(\text{add}(\text{add}(\text{add}(\text{empty}, b_1), b_2), b_3), b_4), b_5)
\]

\[
\begin{array}{ccccc}
   b_1 & b_2 & b_3 & b_4 & b_5 \\
\end{array}
\]

This makes \text{add} straightforward:

\[
\text{add(BandCard b) body} \\
\text{theArray[++}\text{pointer}] = b;
\]
void add(BandCard b) {
    theArray[++pointer] = b;
}

add(add(add(add(add(empty, b1), b2), b3), b4), b5)
void add(BandCard b) {
    theArray[++pointer] = b;
}

add(add(add(add(add(empty, b1), b2), b3), b4), b5)
Adding b5

```
void add(BandCard b) {
    theArray[++pointer] = b;
}
```

```
add(add(add(add(add(empty, b1), b2), b3), b4), b5)
```

```
b1 | b2 | b3 | b4 | b5
↓   ↓
```
void add(BandCard b) {
    theArray[++pointer] = b;
}

add(add(add(add(add(empty, b1), b2), b3), b4), b5)
Implementing `removeTop` is trickier, so let’s look at a few cases, and use the Maude equations to work out what to do.
Maude spec

cq  removeTop(add(C,B)) = empty  if  isEmpty(C) .

removeTop(add(empty, b1))

↓p

| b1 | b2 | b3 | b4 | b5 |

= empty

↓p

| b1 | b2 | b3 | b4 | b5 |
\textbf{removeTop}

\texttt{cq \ removeTop(add(C,B)) = empty \ if \ isEmpty(C).}

\begin{align*}
\text{removeTop}(\text{add}(\text{empty}, \ b1)) & \\
\downarrow p & \\
\begin{array}{ccccc}
\hline
b1 & b2 & b3 & b4 & b5 \\
\hline
\end{array} & = & \text{empty} & \\
\downarrow p & \\
\begin{array}{ccccc}
\hline
b1 & b2 & b3 & b4 & b5 \\
\hline
\end{array}
\end{align*}
cq  removeTop(add(C,B)) = add(removeTop(C), B)  
if  not(isEmpty(C)) .  

removeTop(add(add(empty, b1), b2))  

\[
\begin{array}{c}
\downarrow p \\
b1 & b2 & b3 & b4 & b5 \\
\end{array}
\]

=  add(removeTop(add(empty, b1)), b2)  

\[
\begin{array}{c}
\downarrow p \\
b2 & b2 & b3 & b4 & b5 \\
\end{array}
\]
Maude spec

cq  \text{removeTop(add(C,B)) = add(removeTop(C), B)}
\textbf{if} \ not(\text{isEmpty}(C)) .
And if we look at

\[
\text{removeTop}(\text{add}(\text{add}(\text{add}(\text{empty}, b1), b2), b3))
\]

(exercise!) we see that we need to:
- make the 2nd element 1st; the 3rd element 2nd; the 4th 3rd, etc.
- and decrement the pointer
A loop would be sensible for the first item.
Back to Java

the data representation

```java
public class CardHand {

    private final BandCard[] theCards;
    private int endIndex;

    public CardHand() {
        theCards = new BandCard[BandPack.PACK_SIZE];
        endIndex = -1;
    }
}
```

Is this right?
This is right if we start off with an empty hand.
public class CardHand {

    private final BandCard[] theCards;
    private int endIndex;

    public CardHand() {
        theCards = new BandCard[BandPack.PACK_SIZE];
        endIndex = -1;
    }
}

Is this right?
This is right if we start off with an empty hand.
public class CardHand {

    private final BandCard[] theCards;
    private int endIndex;

    public CardHand() {
        theCards = new BandCard[BandPack.PACK_SIZE];
        endIndex = -1;
    }

    Is this right?
    This is right if we start off with an empty hand.
public class CardHand {

    private final BandCard[] theCards;
    private int endIndex;

    public CardHand() {
        theCards = new BandCard[BandPack.PACK_SIZE];
        endIndex = -1;
    }
}

Is this right?

This is right if we start off with an empty hand.
Implementing the Functionality 1

```java
public void add(BandCard b) {
    theCards[++endIndex] = b;
}

public boolean isEmpty() {
    return endIndex < 0;  // more efficient
    // than testing
    // endIndex == -1
}
```
public void add(BandCard b) {
    theCards[++endIndex] = b;
}

public boolean isEmpty() {
    return endIndex < 0;  // more efficient
    // than testing
    // endIndex == -1
}
Implementing the Functionality 2

```java
public BandCard getTop() {
    return theCards[0];
}

public void removeTop() {
    for (int i = 0; i < endIndex; i++) {
        theCards[i] = theCards[i+1];
    }
    endIndex--;
}
```

functionality: getTop, removeTop
Implementing the Functionality 2

functionality: getTop, removeTop

public BandCard getTop() {
    return theCards[0];
}

public void removeTop() {
    for (int i = 0 ; i < endIndex ; i++) {
        theCards[i] = theCards[i+1];
    }
    endIndex--;
}
That’s CardHand

And we’re done!

We’ve implemented (v1.0 of) CardHand, using the Maude spec to help with the trickiest part (removeTop)

Everyone Happy?
And we’re done!

We’ve implemented (v1.0 of) CardHand, using the Maude spec to help with the trickiest part (removeTop)

Everyone Happy?
That’s CardHand

And we’re done!

We’ve implemented (v1.0 of) CardHand, using the Maude spec to help with the trickiest part (removeTop)

Everyone Happy?
CardHand, Again

Well,

```java
public void removeTop() {
    for (int i = 0 ; i < endIndex ; i++) {
        theCards[i] = theCards[i+1];
    }
    endIndex--;
}
```

This is rather inefficient, as we loop through the array segment `theCards[0..endIndex]` every time we remove a card from the top of the hand (i.e., every round of the game.)
CardHand, Again

Well,

```java
public void removeTop() {
    for (int i = 0 ; i < endIndex ; i++) {
        theCards[i] = theCards[i+1];
    }
    endIndex--;
}
```

This is rather inefficient, as we loop through the array segment `theCards[0..endIndex]` every time we remove a card from the top of the hand (i.e., every round of the game.)
Less Loopy Solution

Just as `endIndex` is a ‘pointer’ keeping track of the last card added, why not introduce another pointer, keeping track of the top of the hand?

(Very like in the Stack example, but at the other end of the array.)

The idea is that instead of shuffling all the cards down to the start of the array, we simply increment our new pointer.
Less Loopy Solution

Just as `endIndex` is a ‘pointer’ keeping track of the last card added, why not introduce another pointer, keeping track of the top of the hand?

(Very like in the Stack example, but at the other end of the array.)

The idea is that instead of shuffling all the cards down to the start of the array, we simply increment our new pointer.
Changing the Data Representation

First, we need to fix how these pointers are to be used (whether the index of the first/last card is equal/+1/-1).

In Java, for all arrays a, the valid indices are integers i such that

\[ 0 \leq i < a.length \]

We’ll keep to that pattern: every card in the hand will have an index i such that

\[ \text{startIndex} \leq i < \text{endIndex} \]

And we also want to ensure that

\[ 0 \leq \text{startIndex} \leq \text{endIndex} \leq \text{BandPack.PACK_SIZE} \]
Changing the Data Representation

First, we need to *fix* how these pointers are to be used (whether the index of the first/last card is equal/+1/-1).

In Java, for all arrays \( a \), the valid indices are integers \( i \) such that

\[ 0 \leq i < a.length \]

We’ll keep to that pattern: every card in the hand will have an index \( i \) such that

\[ \text{startIndex} \leq i < \text{endIndex} \]

And we also want to ensure that

\[ 0 \leq \text{startIndex} \leq \text{endIndex} \leq \text{BandPack.PACK_SIZE} \]
Changing the Data Representation

First, we need to fix how these pointers are to be used (whether the index of the first/last card is equal/+1/-1).

In Java, for all arrays \( a \), the valid indices are integers \( i \) such that

\[
0 \leq i < a.length
\]

We’ll keep to that pattern: every card in the hand will have an index \( i \) such that

\[
startIndex \leq i < endIndex
\]

And we also want to ensure that

\[
0 \leq startIndex \leq endIndex \leq BandPack.PACK_SIZE
\]
Changing the Data Representation

First, we need to fix how these pointers are to be used (whether the index of the first/last card is equal/+1/-1).

In Java, for all arrays $a$, the valid indices are integers $i$ such that

$$0 \leq i < a.length$$

We’ll keep to that pattern: every card in the hand will have an index $i$ such that

$$startIndex \leq i < endIndex$$

And we also want to ensure that

$$0 \leq startIndex \leq endIndex \leq BandPack.PACK_SIZE$$
Changing the Data Representation

For example, in the following state

```
<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
</tr>
</thead>
</table>
```

↑    ↑    ↑

The cards in the hand are b2, b3, and b4.

NB — this is different to how `endIndex` was used in v1.0.
Changing the Data Representation

For example, in the following state

\[
\begin{array}{cccccc}
\text{b1} & \text{b2} & \text{b3} & \text{b4} & \text{b5} & \text{b6} \\
\uparrow & \uparrow & \uparrow \\
\end{array}
\]

The cards in the hand are b2, b3, and b4.

NB — this is different to how `endIndex` was used in v1.0.
Properties of the Data Representation

Four simple principles will guide us through the implementation:

1. `startIndex` is the first card;
2. `endIndex` is (+1) the last card;
3. `startIndex` ≤ `endIndex`;
4. `endIndex` ≤ `BandPack.PACK_SIZE`;
Properties of the Data Representation

Four simple principles will guide us through the implementation:

1. \( \text{startIndex} \) is the first card;
2. \( \text{ endIndex } \) is (+1) the last card;
3. \( \text{ startIndex } \leq \text{ endIndex } \);
4. \( \text{ endIndex } \leq \text{ BandPack.PACK_SIZE} \);
Properties of the Data Representation

Four simple principles will guide us through the implementation:

1. `startIndex` is the first card;
2. `endIndex` is (+1) the last card;
3. `startIndex ≤ endIndex`;
4. `endIndex ≤ BandPack.PACK_SIZE`;
Properties of the Data Representation

Four simple principles will guide us through the implementation:

1. `startIndex` is the first card;
2. `endIndex` is (+1) the last card;
3. `startIndex` \( \leq \) `endIndex`;
4. `endIndex` \( \leq \) `BandPack.PACK_SIZE`;
Initialising the Data Representation

We declare and initialise the array and pointers thus:

```java
class CardHand v1.1

private final BandCard[] theCards = 
    new BandCard[BandPack.PACK_SIZE];
private int startIndex = 0;
private int endIndex = 0;
```

This gives the empty array segment, as there are no $i$ such that $0 \leq i < 0$.

And, since we start off with an empty hand, we’ve kept to all four principles.
Initialising the Data Representation

We declare and initialise the array and pointers thus:

```java
class CardHand v1.1

private final BandCard[] theCards =
    new BandCard[BandPack.PACK_SIZE];
private int startIndex = 0;
private int endIndex = 0;

This gives the empty array segment,
as there are no $i$ such that $0 \leq i < 0$.

And, since we start off with an empty hand,
we’ve kept to all four principles.
Initialising the Data Representation

We declare and initialise the array and pointers thus:

```java
class CardHand v1.1

private final BandCard[] theCards =
    new BandCard[BandPack.PACK_SIZE];
private int startIndex = 0;
private int endIndex = 0;
```

This gives the empty array segment, as there are no $i$ such that $0 \leq i < 0$.

And, since we start off with an empty hand, we’ve kept to all four principles.
Working with the Data Representation

There are some minor changes to the implementations of `isEmpty` and `getTop`:

```java
public boolean isEmpty() {
    return endIndex <= startIndex;
}

public BandCard getTop() {
    return theCards[startIndex];
}
```

Principles 1–3
Principle 1
Working with the Data Representation

There are some minor changes to the implementations of `isEmpty` and `getTop`:

```java
public boolean isEmpty() {
    return endIndex <= startIndex;
}

public BandCard getTop() {
    return theCards[startIndex];
}
```

Principles 1–3
Principle 1
Working with the Data Representation

There are some minor changes to the implementations of `isEmpty` and `getTop`:

```java
public boolean isEmpty() {
    return endIndex <= startIndex;
}

public BandCard getTop() {
    return theCards[startIndex];
}
```

Principles 1–3

Principle 1
Working with the Data Representation

And introducing `startIndex` makes `removeTop` straightforward:

```java
public void removeTop() {
    startIndex++;
}
```

but the main change is in `add` — let’s see why…

Principle 1
Working with the Data Representation

And introducing `startIndex` makes `removeTop` straightforward:

```java
public void removeTop() {
    startIndex++;
}
```

but the main change is in `add` — let’s see why…

Principle 1
Working with the Data Representation

And introducing `startIndex` makes `removeTop` straightforward:

```java
public void removeTop() {
    startIndex++;
}
```

but the main change is in `add` — let’s see why…

Principle 1
Let’s Suppose...

```java
CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
```

![Diagram]

```plaintext
b1 | b2 | b3 | b4 | b5 | b6
↑   ↑   ↑   ↑   ↑   ↑
```
Let’s Suppose...

CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
Let’s Suppose...

```java
CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
```
Let's Suppose...

```java
CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
```

```
↑  ↑  ↑  ↑
b1 | b2 | b3 | b4 | b5 | b6
```
Let's Suppose...

```java
CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
```

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
</tbody>
</table>

CardHand: ch
Let's Suppose...

CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
Let's Suppose...

```java
CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
```

```
  b1  b2  b3  b4  b5  b6
    ↑  ↑  ↑  ↑  ↑  ↑
```
Let's Suppose...

CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
Let’s Suppose...

CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
Let's Suppose...

CardHand ch = ...;
ch.add(b5);
ch.removeTop();
ch.add(b6);
ch.removeTop();
ch.add(b7);
Adding b7

**add(Bandcard b)**

if there’s no room

shuffle down

move pointers

add the card
Adding b7

add(Bandcard b)

if there's no room

shuffle down

move pointers

add the card
Adding b7

add(Bandcard b)

if there’s no room

shuffle down

move pointers

add the card
Adding b7

```
add(Bandcard b)
if there's no room
shuffle down
move pointers
add the card
```

```
| b1 | b2 | b3 | b4 | b5 | b6 |
|----------------|
| b3 | b2 | b3 | b4 | b5 | b6 |
| b3 | b4 | b3 | b4 | b5 | b6 |
| b3 | b4 | b5 | b4 | b5 | b6 |
| b3 | b4 | b5 | b6 | b5 | b6 |
```

```
↑
↓
↑
```

```
| b3 | b4 | b5 | b6 | b5 | b6 |
|----------------|
| b3 | b4 | b5 | b6 | b5 | b6 |
| b3 | b4 | b5 | b6 | b5 | b6 |
| b3 | b4 | b5 | b6 | b5 | b6 |
| b3 | b4 | b5 | b6 | b5 | b6 |
```

```
↑
↑
↑
```
Adding b7

add(Bandcard b)

if there's no room

shuffle down

move pointers

add the card
Adding b7

add(Bandcard b)

if there’s no room

shuffle down

move pointers

add the card
Adding b7

\text{add(Bandcard b)}

if there's no room

shuffle down

move pointers

add the card
Adding b7

```plaintext
add(Bandcard b)
if there's no room
shuffle down
move pointers
add the card
```
Adding b7

add(Bandcard b)

if there’s no room

shuffle down

move pointers

add the card
Adding b7

add(Bandcard b)

if there’s no room

shuffle down

move pointers

add the card
```java
add(Bandcard b)

// if there's no room, make room
if (endIndex == BandPack.PACK_SIZE) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;
```
add(Bandcard b)

// if there's no room, make room
if (endIndex == BandPack.PACK_SIZE) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }

    // move the pointers
    startIndex = 0;
    endIndex = len;
}

// add the card
theCards[endIndex++] = b;
The Code

```java
add(Bandcard b)
```

```java
// if there's no room, make room
if (endIndex == BandPack.PACK_SIZE) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;
```
add(Bandcard b)

// if there's no room, make room
if (endIndex == BandPack.PACK_SIZE) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0; i < len; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;
add(Bandcard b)

// if there's no room, make room
if (endIndex == BandPack.PACK_SIZE) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0; i < len; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;
Work out how each of the four principles determined the code in the preceding slide.
That’s CardHand v1.1

That completes the implementation of CardHand v1.1.

Note that the implementation assumes

- that `getTop()` and `removeTop()` will only be called when the hand is not empty (which makes sense, as the trump game is over as soon as one player’s hand is empty)
- that `add()` is called only when the hand is not full (again, the game would be over as soon as one player’s hand is full)

The assumptions are reasonable, but should be documented.
That's CardHand v1.1

That completes the implementation of CardHand v1.1.

Note that the implementation *assumes*

- that `getTop()` and `removeTop()` will only be called when the hand is not empty (which makes sense, as the trump game is over as soon as one player’s hand is empty)
- that `add()` is called only when the hand is not full (again, the game would be over as soon as one player’s hand is full)

The assumptions are reasonable, but should be documented.
That’s CardHand v1.1

That completes the implementation of CardHand v1.1.

Note that the implementation assumes

- that `getTop()` and `removeTop()` will only be called when the hand is not empty (which makes sense, as the trump game is over as soon as one player’s hand is empty)

- that `add()` is called only when the hand is not full (again, the game would be over as soon as one player’s hand is full)

The assumptions are reasonable, but should be documented.
That's CardHand v1.1

That completes the implementation of CardHand v1.1.

Note that the implementation *assumes*

- that `getTop()` and `removeTop()` will only be called when the hand is not empty (which makes sense, as the trump game is over as soon as one player’s hand is empty)
- that `add()` is called only when the hand is not full (again, the game would be over as soon as one player’s hand is full)

The assumptions are reasonable, but should be *documented*. 
Class Invariants

Our use of an array and two pointers relied upon the following property:

\[ 0 \leq \text{startIndex} \leq \text{endIndex} \leq \text{BandPack.PACK\_SIZE} \]

For example, if \( \text{startIndex} < 0 \), then the code

```java
public BandCard getTop() {
    return theCards[startIndex];
}
```

would throw an `ArrayIndexOutOfBoundsException`. 
Class Invariants

Our use of an array and two pointers relied upon the following property:

$$0 \leq \text{startIndex} \leq \text{endIndex} \leq \text{BandPack.PACK_SIZE}$$

For example, if $\text{startIndex} < 0$, then the code

```java
public BandCard getTop() {
    return theCards[startIndex];
}
```

would throw an `ArrayIndexOutOfBoundsException`. 
Class Invariants

Our implementation guarantees that this will never happen. It is simply impossible to get any CardHand instance into a state where startIndex < 0. (This would not be true if startIndex wasn’t private!)

The reason for this is that startIndex is initialised (when a new instance is created) to 0, and none of the (public) methods can set startIndex to less than 0.

Properties like this, which are true on creation, and ‘preserved’ by all public methods, are called class invariants.
Our implementation guarantees that this will never happen. It is simply impossible to get any CardHand instance into a state where `startIndex < 0`. (This would not be true if `startIndex` wasn’t private!)

The reason for this is that `startIndex` is initialised (when a new instance is created) to 0, and none of the (public) methods can set `startIndex` to less than 0.

Properties like this, which are true on creation, and ‘preserved’ by all public methods, are called class invariants.
Class Invariants

Our implementation guarantees that this will never happen. It is simply impossible to get any CardHand instance into a state where `startIndex < 0`. (This would not be true if `startIndex` wasn’t private!)

The reason for this is that `startIndex` is initialised (when a new instance is created) to 0, and none of the (public) methods can set `startIndex` to less than 0.

Properties like this, which are true on creation, and ‘preserved’ by all public methods, are called class invariants.
Our implementation guarantees that this will never happen. It is simply impossible to get any CardHand instance into a state where \( \text{startIndex} < 0 \).

(This would not be true if \( \text{startIndex} \) wasn’t private!)

The reason for this is that \( \text{startIndex} \) is initialised (when a new instance is created) to 0, and none of the (public) methods can set \( \text{startIndex} \) to less than 0.

Properties like this, which are true on creation, and ‘preserved’ by all public methods, are called **class invariants**.
Class Invariants

We would like

\[
\text{startIndex} \leq \text{ endIndex } \leq \text{ BandPack.PACK\_SIZE}
\]

also to be a class invariant.

It is true on initialisation, but it is possible to call methods in such a way that the following holds:

\[
\text{ endIndex } < \text{ startIndex}
\]

(e.g., call removeTop on an empty hand).

Similarly, it is possible to call methods so that

\[
\text{ BandPack.PACK\_SIZE } < \text{ endIndex}
\]

Exercise: how?
Class Invariants

We would like

\[ \text{startIndex} \leq \text{ endIndex } \leq \text{BandPack.PACK\_SIZE} \]

also to be a class invariant.

It is true on intialisation, but it is possible to call methods in such a way that the following holds:

\[ \text{ endIndex } < \text{ startIndex} \]

(e.g., call removeTop on an empty hand).

Similarly, it is possible to call methods so that

\[ \text{BandPack.PACK\_SIZE} < \text{ endIndex} \]

**Exercise:** how?
Class Invariants

We would like

\[ \text{startIndex} \leq \text{endIndex} \leq \text{BandPack.PACK\_SIZE} \]

also to be a class invariant.

It is true on intialisation, but it is possible to call methods in such a way that the following holds:

\[ \text{endIndex} < \text{startIndex} \]

(e.g., call removeTop on an empty hand).

Similarly, it is possible to call methods so that

\[ \text{BandPack.PACK\_SIZE} < \text{endIndex} \]

Exercise: how?
That’s All, Folks!

Summary

- Implementing ADTs
- Class invariants

Next:

List ADTs