Set
Set

• The Set component family allows you to manipulate finite sets of elements of any (arbitrary) type
Interfaces and Classes

- Standard
  - SetKernel
    - implements Set
  - extends
- Set
  - implements Set1L
  - implements Set2
  - implements Set3
Interfaces and Classes

Standard

extends

Kernel

extends

Set

Set1L

implements

Set2

implements

Set3

Standard has contracts for three methods:

- clear
- newInstance
- transferFrom
Interfaces and Classes

SetKernel

has contracts for five methods:
add
remove
removeAny
contains
size

extends

Standard

implements

Set1

Set2

Set3
Interfaces and Classes

Set

Set1L

Set2

Set3

SetKernel

Standard

Set

implements

extends

has contracts for two other methods:

add

remove
Mathematical Model

• The value of a Set variable is modeled as a (finite) set of elements of type $T$

• Formally:

  \[
  \text{type Set is modeled by finite set of } T
  \]
Constructors

• There is one *constructor* for each implementation class for *Set*

• As always:
  – The name of the constructor is the name of the implementation class
  – The constructor has its own contract (which is in the kernel interface *SetKernel*)
No-argument Constructor

• Ensures:

\[
\texttt{this} = \{ \}
\]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Set&lt;Integer&gt; si = new Set1L&lt;&gt;();</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Set&lt;Integer&gt; si = new Set1L&lt;&gt;();</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>si = {}</code></td>
</tr>
</tbody>
</table>
Methods for Set

- All the methods for Set are **instance methods**, i.e., you call them as follows:
  
  ```
  s.methodName(arguments)
  ```

  where `s` is an initialized non-null variable of type `Set<T>` for some `T`
add

```c
void add(T x)
```

- Adds \( x \) to \textit{this}.
- Aliases: reference \( x \)
- Updates: \textit{this}
- Requires: \( x \) is not in \textit{this}
- Ensures: \( \textit{this} = \#\textit{this} \text{ union } \{x\} \)
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si = { 49, 3 }</code></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td><code>si.add(k);</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>si.add(k);</td>
<td>( si = { 49, 3 } ) ( k = 70 )</td>
</tr>
<tr>
<td></td>
<td>( si = { 49, 3, 70 } ) ( k = 70 )</td>
</tr>
</tbody>
</table>
Example

Note the aliasing here between the “70s”, not shown in the tracing table but visible if you draw a diagram of this situation.

\[
\begin{array}{|c|}
\hline
\text{State} \\
\hline
\text{si} = \{ 49, 3 \} \\
\hline
\text{k} = 70 \\
\hline
\text{si} = \{ 49, 3, 70 \} \\
\text{k} = 70 \\
\hline
\end{array}
\]
remove

T remove(T x)

• Removes \( x \) from \texttt{this}, and returns it.
• Updates: \texttt{this}
• Requires:
  \( x \) is in \texttt{this}
• Ensures:
  \texttt{this} = \#\texttt{this} \setminus \{x\} \text{ and }
  \texttt{remove} = x
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| \( m = \text{si}.\text{remove}(k); \) | \( \text{si} = \{ 49, 3, 70 \} \)  
\( k = 3 \)  
\( m = -17 \) |
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( si = { 49, 3, 70 } )</td>
</tr>
<tr>
<td></td>
<td>( k = 3 )</td>
</tr>
<tr>
<td></td>
<td>( m = -17 )</td>
</tr>
<tr>
<td>( m = si.remove(k); )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( si = { 49, 70 } )</td>
</tr>
<tr>
<td></td>
<td>( k = 3 )</td>
</tr>
<tr>
<td></td>
<td>( m = 3 )</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
</table>
|          | $si = \{ 49, 3, 70 \}$  
|          | $k = 3$  
|          | $m = -17$  |
|          | $si = \{ 49, 70 \}$  
|          | $k = 3$  
|          | $m = 3$  |

The precondition for `remove(x is in this)` is satisfied whether or not there is aliasing involving the “3s” in this situation. Why?
removeAny

T removeAny()

- Removes and returns an arbitrary element from this.
- Updates: this
- Requires: \(| this | > 0\)
- Ensures:
  
  removeAny is in #this and
  
  \( this = #this \ \backslash \ \{ \text{removeAny} \)
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| si = { 49, 3, 70 }  
k = 134        |
| k = si.removeAny();|                     |
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = si.removeAny();$</td>
<td>$si = { 49, 3, 70 }$</td>
</tr>
<tr>
<td></td>
<td>$k = 134$</td>
</tr>
<tr>
<td></td>
<td>$si = { 3, 70 }$</td>
</tr>
<tr>
<td></td>
<td>$k = 49$</td>
</tr>
</tbody>
</table>
Example

Other possible outcomes are:

\[ si = \{ 49, 70 \} \]
\[ k = 3 \]

or:

\[ si = \{ 49, 3 \} \]
\[ k = 70 \]

\[ si = \{ 49, 3, 70 \} \]
\[ k = 134 \]

\[ si = \{ 3, 70 \} \]
\[ k = 49 \]
contains

boolean contains(T x)

• Reports whether x is in this.
• Ensures:

contains = (x is in this)
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si = { 49, 3, 70 }</code></td>
<td></td>
</tr>
<tr>
<td><code>k = -58</code></td>
<td></td>
</tr>
<tr>
<td><code>boolean b = si.contains(k);</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean b =</code></td>
<td><code>si = { 49, 3, 70 }</code></td>
</tr>
<tr>
<td><code>si.contains(k);</code></td>
<td><code>k = -58</code></td>
</tr>
<tr>
<td></td>
<td><code>si = { 49, 3, 70 }</code></td>
</tr>
<tr>
<td></td>
<td><code>k = -58</code></td>
</tr>
<tr>
<td></td>
<td><code>b = false</code></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$si = { 49, 3, 70 }$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td>\textbf{boolean} $b =$</td>
<td></td>
</tr>
<tr>
<td>\textit{si.contains}(k);</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$si = { 49, 3, 70 }$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td><code>boolean b =</code></td>
<td></td>
</tr>
<tr>
<td><code>si.contains(k);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$si = { 49, 3, 70 }$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td></td>
<td>$b = true$</td>
</tr>
</tbody>
</table>
The condition checked by `contains (x is in this)` is satisfied whether or not there is aliasing involving the “70s” in this situation. Why?

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>si = { 49, 3, 70 }</td>
</tr>
<tr>
<td>k = 70</td>
</tr>
<tr>
<td>b = true</td>
</tr>
</tbody>
</table>
size

int size()

- Reports the size (cardinality) of this.
- Ensures:
  
  \[ \text{size} = |\text{this}| \]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si = { 49, 3, 70 }</code></td>
<td><code>n = -45843</code></td>
</tr>
<tr>
<td><code>n = si.size();</code></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>si = { 49, 3, 70 }</td>
</tr>
<tr>
<td></td>
<td>n = -45843</td>
</tr>
<tr>
<td>n = si.size();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>si = { 49, 3, 70 }</td>
</tr>
<tr>
<td></td>
<td>n = 3</td>
</tr>
</tbody>
</table>
Overloading

• A method with the same name as another method, but with a different parameter profile (number, types, and order of formal parameters) is said to be overloaded

• A method may not be overloaded on the basis of its return type

• Java disambiguates between overloaded methods based on the number, types, and order of arguments at the point of a call
add

\texttt{void \hspace{1em} add(Set<T> \ s)}

- Adds to \texttt{this} all elements of \texttt{s} that are not already in \texttt{this}, also removing just those elements from \texttt{s}.
- Updates: \texttt{this, s}
- Ensures:
  \[
  \texttt{this} = \#\texttt{this union} \ #\texttt{s} \quad \text{and} \quad \texttt{s} = \#\texttt{this intersection} \ #\texttt{s}
  \]
The `add` method for receivers of type `Set<T>` is overloaded:

- one method takes an argument of type `T`, and
- one method takes an argument of type `Set<T>`. 

The `add` method modifies this set and the given set as follows:

- Adds to `this` all elements of `s` not already in `this`, also removing just those elements from `s`.
- Updates: `this`, `s`.
- Ensures:
  
  \[
  this = \#this \text{ union } \#s \quad \text{and} \quad s = \#this \text{ intersection } \#s
  \]
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( s_1 = { 1, 2, 3, 4 } )</td>
</tr>
<tr>
<td></td>
<td>( s_2 = { 3, 4, 5, 6 } )</td>
</tr>
<tr>
<td></td>
<td>( s_1.add(s2); )</td>
</tr>
</tbody>
</table>

\( s_1 \) and \( s_2 \) are initialized with the given elements.
**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s1 = { 1, 2, 3, 4 }</td>
</tr>
<tr>
<td></td>
<td>s2 = { 3, 4, 5, 6}</td>
</tr>
<tr>
<td>s1.add(s2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s1 = { 1, 2, 3, 4, 5, 6}</td>
</tr>
<tr>
<td></td>
<td>s2 = { 3, 4 }</td>
</tr>
</tbody>
</table>
In other words, this moves all elements of \( s_2 \setminus s_1 \) from \( s_2 \) into \( s_1 \); it “conserves” objects of type \( T \).

\[
\begin{align*}
  s_1 &= \{1, 2, 3, 4\} \\
  s_2 &= \{3, 4, 5, 6\} \\
  s_1.add(s2); \\
  s_1 &= \{1, 2, 3, 4, 5, 6\} \\
  s_2 &= \{3, 4\}
\end{align*}
\]
remove

Set<T> remove(Set<T> s)

- Removes from this all elements of s that are also in this, leaving s unchanged, and returns the elements actually removed.
- Updates: this
- Ensures:
  
  \[
  \text{this} = \#\text{this} \setminus s \quad \text{and}
  \]
  
  \[
  \text{remove} = \#\text{this intersection } s
  \]
remove

Set<T> remove (Set<T> s)

• Removes from this all elements of s that are also in this, leaving s unchanged, and returns the elements actually removed.

• Updates: this

• Ensures: 

  \( \text{this} = \# \text{this} \)

  remove = \# \text{this intersection } s

The `remove` method for receivers of type `Set<T>` is overloaded:

• one method takes an argument of type `T`, and

• one method takes an argument of type `Set<T>`. 
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s1 = { 1, 2, 3, 4 }$</td>
</tr>
<tr>
<td></td>
<td>$s2 = { 3, 4, 5, 6 }$</td>
</tr>
<tr>
<td></td>
<td>$s3 = { 10 }$</td>
</tr>
<tr>
<td>$s3 = s1.remove(s2);$</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s3 = s1.remove(s2);</code></td>
<td>( s1 = { 1, 2, 3, 4 } )</td>
</tr>
<tr>
<td></td>
<td>( s2 = { 3, 4, 5, 6 } )</td>
</tr>
<tr>
<td></td>
<td>( s3 = { 10 } )</td>
</tr>
<tr>
<td></td>
<td>( s3 = s1.remove(s2); )</td>
</tr>
<tr>
<td></td>
<td>( s1 = { 1, 2 } )</td>
</tr>
<tr>
<td></td>
<td>( s2 = { 3, 4, 5, 6 } )</td>
</tr>
<tr>
<td></td>
<td>( s3 = { 3, 4 } )</td>
</tr>
</tbody>
</table>
In other words, this “conserves” all elements of \( s_1 \) and \( s_2 \); they all wind up in some \( \text{Set}\langle T\rangle \) rather than being “lost”.

<table>
<thead>
<tr>
<th>State</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( s_1 = {1, 2, 3, 4} )</td>
</tr>
<tr>
<td></td>
<td>( s_2 = {3, 4, 5, 6} )</td>
</tr>
<tr>
<td></td>
<td>( s_3 = {10} )</td>
</tr>
</tbody>
</table>

\[
s_3 = s_1.remove(s_2), \\
s_1 = \{1, 2\}, \\
s_2 = \{3, 4, 5, 6\}, \\
s_3 = \{3, 4\}
\]
Iterating Over a $\text{Set}$

- Suppose you want to do something with each of the elements of a $\text{Set}\langle T \rangle \ s$
- How might you do that?
Iterating With `removeAny`

```java
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size() > 0) {
    T x = temp.removeAny();
    // do something with x
    s.add(x);
}
```
Iterating With `removeAny`

```java
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size() > 0) {
    T x = temp.removeAny(); // do something
    s.add(x);
}
```

Recall that `newInstance` returns a new object of the same **object type** as the receiver, as if it were a no-argument constructor; but we don’t need to *know* the object type of `s` to get this new object.
Iterating With `removeAny`

```java
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size() > 0) {
    T x = temp.removeAny(); // do something with x
    s.add(x);
}
```

Why `transferFrom` rather than `copyFrom`?

- **Performance**: there is no need for a copy, and `transferFrom` is far more efficient.
- **We really want** `s` **to be empty to start the iteration, and this does it.**
Iterating With \texttt{removeAny}

- This code has the following properties:
  - It introduces no dangerous aliases, so it is relatively easy to reason about; just think about values, not references
  - If what you want to do with each element is to change it, then the approach works because you may change the value of $x$ each time through the loop body
  - It is reasonably efficient (making no copies of elements of type $T$, though it does use \texttt{removeAny} and \texttt{add}, and these could be slow)
Iterating With \texttt{removeAny}

\begin{itemize}
\item This code has the following properties:
  \begin{itemize}
  \item It introduces no dangerous aliases, so it is relatively easy to reason about; just think about values, not references.
  \item If what you want to do with each element is to change it, then the approach works because you may change the value of $x$ each time through the loop body.
  \item It is reasonably efficient (making no copies of elements of type $T$, though \texttt{removeAny} and \texttt{add} may and these could be slow).
  \end{itemize}
\end{itemize}

\textbf{It does introduce an alias (where?) but it is of no consequence (why?).}
Iterators

• Conventional Java style for iterating over a “collection” like a Set is to use an iterator so you can do this without taking the collection apart and reconstituting it
One More Interface

```
Set1L implements SetKernel
Set2 implements SetKernel
Set3 implements SetKernel
```

```
SetKernel extends Standard
SetKernel extends Iterable
Set extends SetKernel
```
One More Interface

```
SetKernel implements Standard extends Iterable
Set1L implements Set
Set2 implements Set
Set3 implements Set
```

**Iterable** has a contract for one method: `iterator`
iterator

Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

\[
\text{entries}(\neg \text{this.seen} \times \neg \text{this.unseen}) = \text{this} \\
\text{and} \\
|\neg \text{this.seen} \times \neg \text{this.unseen}| = |\text{this}|
\]
Iterator<T> iterator()

- Returns an iterator over a set of elements of type T.
- Ensures:

\[
\text{entries}(\sim \text{this.seen} \ast \sim \text{this.unseen}) = \text{this}
\]

\[
|\sim \text{this.seen} \ast \sim \text{this.unseen}| = |\text{this}|
\]

Iterator is yet another interface in the Java libraries (in the package java.util).
Iterator<T> iterator()

- Returns an iterator over a set of elements of type T.
- Ensures:

\[
\text{entries}(\sim \text{this}.\text{seen} \ast \sim \text{this}.\text{unseen}) = \text{this}
\]

and

\[
|\sim \text{this}.\text{seen} \ast \sim \text{this}.\text{unseen}| = |\text{this}|
\]

We will return to decipher the contract after seeing the easiest way for this method to be used...
For-Each Loops

- Since `Set<T>` extends the interface `Iterable` (so it inherits the `iterator` method), you may write a `for-each loop` to “see” all elements of `Set<T> s`:

```java
for (T x : s) {
    // do something with x, but do not call methods on s, or change the value of x or s
}
```
For-Each Loops

Since `Set<T>` extends the interface `Iterable` (so it has the `iterator` method), you may write a `for-each loop` to “see” all elements of a `Set<T>`:

```java
for (T x : s) {
    // do something with x, but do not call methods on s, or change the value of x or s
}
```

This declares `x` as a local variable of type `T` in the loop; on each iteration, `x` is **aliased** to a different element of `s`. 

9 January 2015 OSU CSE
For-Each Loop Example

- Count the number of strings of length 5 in a `Set<String>`:

```java
Set<String> dictionary = ...
...
int count = 0;
for (String word : dictionary) {
    if (word.length() == 5) {
        count++;
    }
}
```
In Which Order?

• The kernel interface (\texttt{SetKernel} in this case) contains the contract for the \texttt{iterator} method, as specialized for the type \texttt{Set<T>}

• This contract specifies the \textit{order} in which the elements are seen
iterator Contract

• Two new *mathematical variables* are involved in the contract:
  – The *string of* $T$ called ~*this.seen* contains, in order, those values *already “seen”* in the for-each loop iterations up to any point
  – The *string of* $T$ called ~*this.unseen* contains, in order, those values *not yet “seen”* in the for-each loop iterations up to that point
iterator

Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

\[
\text{entries}(\sim this.\text{seen} \times \sim this.\text{unseen}) = this
\]

and

\[
|\sim this.\text{seen} \times \sim this.\text{unseen}| = |this|
\]
Iterator<T> iterator()

• Returns an iterator over a set of elements of type $T$.

• Ensures:

$$\text{entries}(\sim \text{this}.\text{seen} * \sim \text{this}.\text{unseen}) = \text{this}$$

and

$$|\sim \text{this}.\text{seen} * \sim \text{this}.\text{unseen}| = |\text{this}|$$
iterator

Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

$$\text{entries}(\sim\text{this}.\text{seen} \times \sim\text{this}.\text{unseen}) = \text{this}$$

and

$$|\sim\text{this}.\text{seen} \times \sim\text{this}.\text{unseen}| = |\text{this}|$$

The finite set of T of values already seen and not yet seen...
Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

\[
\text{entries}(\sim\text{this}.\text{seen} \times \sim\text{this}.\text{unseen}) = \text{this}
\]

and

\[
|\sim\text{this}.\text{seen} \times \sim\text{this}.\text{unseen}| = |\text{this}|
\]

The *finite set of* T of values already seen and not yet seen... is equal to the entire set \textit{this}.
Iterator<T>

- Returns an iterator over a set of elements of type T.
- Ensures:

\[
\text{entries}(\neg \text{seen} \times \neg \text{unseen}) = \text{this} \\
\text{and} \\
|\neg \text{seen} \times \neg \text{unseen}| = |\text{this}|
\]
Iterating With \texttt{iterator}

- The \texttt{for-each} code has the following properties:
  - It introduces aliases, so you must be careful to “follow the rules”; specifically, the loop body should not call any methods on \texttt{s}
  - If what you want to do to each element is to change it (when \texttt{T} is a mutable type), then the approach does not work because the loop body should not change \texttt{x}
  - It may be more efficient than using \texttt{removeAny} (i.e., it also makes no copies of elements of type \texttt{T}, though it does use \texttt{iterator} methods to carry out the for-each loop, and these could be slow)
Resources

• OSU CSE Components API: Set

• Java Libraries API: Iterable and Iterator
  – [http://docs.oracle.com/javase/7/docs/api/](http://docs.oracle.com/javase/7/docs/api/)