Set
Set

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

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

- `Standard` extends `Kernel` and implements three methods:
  - `clear`
  - `newInstance`
  - `transferFrom`

- `Standard` also implements `Set`, which extends `SetKernel` and implements:
  - `impl1L`
  - `impl2`
  - `impl3`

22 March 2021
OSU CSE
Interfaces and Classes

SetKernel

Standard

has contracts for five methods:
add
remove
removeAny
contains
size

SetKernel

extends

implements

Set3
Set has contracts for three other methods:
- add
- remove
- isSubset
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:

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

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

- All the methods for \textit{Set} are \textit{instance methods}, i.e., you call them as follows:
  \begin{verbatim}
  s.methodName(arguments)
  \end{verbatim}
  where \texttt{s} is an initialized non-null variable of type \texttt{Set<T>} for some \texttt{T}
void add(T x)

• Adds x to this.
• Aliases: reference x
• Updates: this
• Requires: x is not in this
• Ensures:
  this = #this union {x}
### Example

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

```c
si.add(k);
```
### Example

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

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

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

T remove(T x)

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

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

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

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

The precondition for \texttt{remove}(x \textit{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 \ {removeAny}
Example

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

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

Other possible outcomes are:

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

or:

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

boolean contains(T x)

• Reports whether x is in this.
• Ensures:

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

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

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

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

int size()

• Reports the size (cardinality) of this.
• Ensures:

\[ size = |this| \]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| $n = \text{si.size();}$ | $\text{si = \{ 49, 3, 70 \}}$  
$\text{n = -45843}$         |
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| $n = \text{\texttt{si.size()}}$; | $\text{\texttt{si} = \{ 49, 3, 70 \}}$
$n = -45843$ |
| | $\text{\texttt{si} = \{ 49, 3, 70 \}}$
$n = 3$ |
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**

```java
void add(Set<T> s)
```

- Adds to `this` all elements of `s` that are not already in `this`, also removing just those elements from `s`.
- Updates: `this`, `s`
- Ensures:
  
  \[
  this = \#this \text{ union } \#s \text{ and } s = \#this \text{ intersection } \#s
  \]
void add(Set<T> s)

• Adds to this all elements in s that are not already in this, also removing just those elements from s.
• Updates: this, s
• Ensures:
  this = #this union #s and
  s = #this intersection #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>.
Example

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

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_1.add(s2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${1, 2, 3, 4}$</td>
<td>${3, 4, 5, 6}$</td>
<td></td>
</tr>
<tr>
<td>${1, 2, 3, 4, 5, 6}$</td>
<td>${3, 4}$</td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
\text{this} = \#\text{this} \setminus s \quad \text{and} \\
\text{remove} = \#\text{this} \text{ intersection } s
\]
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:
  
  this = #this
  remove = #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>(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>
<tr>
<td>(s_3 = s_1.remove(s_2));</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</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>
<tr>
<td>$s_3 = s_1.remove(s_2);$</td>
<td>$s_1 = { 1, 2 }$</td>
</tr>
<tr>
<td></td>
<td>$s_2 = { 3, 4, 5, 6 }$</td>
</tr>
<tr>
<td></td>
<td>$s_3 = { 3, 4 }$</td>
</tr>
</tbody>
</table>
In other words, this “conserves” all elements of \(s1\) and \(s2\); they all wind up in some Set\(<T>\) rather than being “lost”.

\[
s3 = s1.remove(s2),
\]

\[
s1 = \{ 1, 2 \}
\]

\[
s2 = \{ 3, 4, 5, 6 \}
\]

\[
s3 = \{ 3, 4 \}
\]
isSubset

boolean isSubset(Set<T> s)

• Reports whether this is a subset of s.
• Ensures:

    isSubset = this is subset of s
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean b =</td>
<td>$s1 = { 2, 4 }$</td>
</tr>
<tr>
<td>s1.isSubset(s2);</td>
<td>$s2 = { 1, 2, 3, 4 }$</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s_1 = { 2, 4 }$ $s_2 = { 1, 2, 3, 4 }$</td>
</tr>
<tr>
<td>boolean $b =$</td>
<td></td>
</tr>
<tr>
<td>s1.isSubset(s2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_1 = { 2, 4 }$ $s_2 = { 1, 2, 3, 4 }$ $b = true$</td>
</tr>
</tbody>
</table>
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| ```
boolean b =
    s1.isSubset(s2);
``` |
| \( s1 = \{ 3, 4, 5 \} \)
\( s2 = \{ 1, 2, 3, 4 \} \) |
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s1 = { 3, 4, 5 }$</td>
</tr>
<tr>
<td></td>
<td>$s2 = { 1, 2, 3, 4 }$</td>
</tr>
<tr>
<td>boolean $b =$</td>
<td></td>
</tr>
<tr>
<td>$s1$.isSubset($s2$);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s1 = { 3, 4, 5 }$</td>
</tr>
<tr>
<td></td>
<td>$s2 = { 1, 2, 3, 4 }$</td>
</tr>
<tr>
<td></td>
<td>$b = false$</td>
</tr>
</tbody>
</table>
Iterating Over a \texttt{Set}

- Suppose you want to do something with each of the elements of a \texttt{Set<T> 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 with x
    s.add(x);
}
```

Recall that `newInstance` returns a new object of the same **object type** (dynamic 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 \texttt{x} each time through the loop body
  - It is reasonably efficient (making no copies of elements of type \texttt{T}, though it does use \texttt{removeAny} and \texttt{add}, and these could be slow)
Iterating With `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 `removeAny` and `add`, and these could be slow)

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

Standard

SetKernel

Iterable

Set

Set1L

Set2

Set3
One More Interface

SetKernel

Standard

Iterable

Set

Set1L

Set2

Set3

Iterable has a contract for one method: iterator
iterator

Iterator<T> iterator()

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

• Ensures:

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

Iterator<T> iterator()

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

• Ensures:

$$\text{entries(~this.seen \land \sim \text{this.unseen}) \land |\sim \text{this.seen} \land \sim \text{this.unseen}| = |\text{this}|}$$

The `iterator` method returns an iterator over a set of elements of type `T`. It ensures that the set of elements seen (`~this.seen`) and unseen (`~this.unseen`) equals the set of elements in the iterator (`this`).

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

Iterator<T> iterator()

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

• Ensures:

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

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 `Set<T> s`:

```
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`. 
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 (`SetKernel` in this case) contains the contract for the iterator method, as specialized for the type `Set<T>`
- This contract specifies the order in which the elements are seen
Two new mathematical variables are involved in the contract:

– The string of $T$ called $\sim\text{this}.\text{seen}$ contains, in order, those values already “seen” in the for-each loop iterations up to any point.

– The string of $T$ called $\sim\text{this}.\text{unseen}$ 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:

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

and

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

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

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

The concatenation of the string of T values already seen and the values not yet seen...
iterator

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

• Ensures:

\[
\text{entries}(\neg \text{this.seen} \times \neg \text{this.unseen}) = \text{this}
\]
\[
\text{and}
\]
\[
|\neg \text{this.seen} \times \neg \text{this.unseen}| = |\text{this}|
\]

The finite set of \( T \) of values already seen and not yet seen...
Iterator\langle T \rangle\ it = \ldots

- Returns an iterator over a set of elements of type \( T \).
- Ensures:
  
  \[
  \text{entries}(\neg t.h.is\ seen \times \neg t.h.is\ unseen) = t.h.is\ 
  \text{and}
  \]
  
  \[
  |\neg t.h.is\ seen \times \neg t.h.is\ unseen| = |t.h.is|\n  \]

The \textit{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:

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

and

\[
\vert \sim \text{this.seen} \ast \sim \text{this.unseen} \vert = \vert \text{this} \vert
\]
Iterating With iterator

• The 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 s
  – If what you want to do to each element is to change it (when T is a mutable type), then the approach does not work because the loop body should not change x
  – It may be more efficient than using removeAny (i.e., it also makes no copies of elements of type T, though it does use iterator methods to carry out the for-each loop, and these could be slow)
Resources

• OSU CSE Components API: \texttt{Set}
  – \url{http://web.cse.ohio-state.edu/software/common/doc/}

• Java Libraries API: \texttt{Iterable} and \texttt{Iterator}
  – \url{http://docs.oracle.com/javase/8/docs/api/}