Recap of Tuesday

• 214: managing complexity, from programs to systems
  - Threads and concurrency
  - Object-oriented programming
  - Analysis and modeling
  - Design

• Object-oriented programming organizes code around concepts
  - Objects contain state and behavior
  - Methods capture behavior, fields capture state
  - Classes as template for objects
  - As we will see, this organization allows
    • Greater reuse of concepts
    • Better support for change when concepts vary
An Object-Oriented Set Library

• We communicate with objects by sending them messages
  ▪ Or, equivalently, invoking their methods

• What messages should we be able to send to a set?
  ▪ *Hint:* think about mathematical set operations
An Object-Oriented Set Library

- We communicate with objects by sending them messages
  - Or, equivalently, invoking their methods

- What messages should we be able to send to a set?
  - Hint: think about mathematical set operations

- Let’s design an interface to a (functional) set object
  - Equivalent to header files in C
  - But now we are listing the messages understood by an object
    - Java interfaces may not have (instance) fields

```java
interface IntSet {
    /** does the IntSet contain element? */
    boolean contains(int element);
    /** is the IntSet a subset of otherSet? */
    boolean isSubsetOf(IntSet otherSet);
}
```
Implementing Set

- An implementation of an interface is defined using a `class`
  - Provides method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
    - May also define additional methods and/or data fields
  - The class is a `subtype` of the interfaces it implements

- Trivial example: an empty set

```java
class EmptySet implements IntSet {

    /** does the IntSet contain element? */
    boolean contains(int element) {
    }
}
```
Implementing Set

interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}

class EmptySet implements IntSet {
    /** does the IntSet contain element? */
    boolean contains(int element) {
        return false;
    }
}

- Trivial example: an empty set

error: method isSubsetOf from interface IntSet is not implemented
Implementing Set

- An implementation of an interface is defined using a **class**
  - Provides method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
  - May also define additional methods and/or data fields
  - The class is a **subtype** of the interfaces it implements

- Trivial example: an empty set

```java
class EmptySet implements IntSet {
    /** does the IntSet contain element? */
    boolean contains(int element) {
        return false;
    }
    /** is the IntSet a subset of otherSet? */
    boolean isSubsetOf(IntSet otherSet) {
        return true;
    }
}
```
Using an EmptySet

```java
class EmptySet implements IntSet {
    /** does the IntSet contain element? */
    boolean contains(int element) { return false; }
    /** is the IntSet a subset of otherSet? */
    boolean isSubsetOf(IntSet otherSet) { return true; }
}
```

IntSet s = new EmptySet();
boolean f = s.contains(0); // false
boolean t = s.isSubsetOf(s); // true

The **receiver**, an implicit argument, called `this` inside the method

The method **name**. Identifies which method to use, of all the methods the receiver’s class defines

Method **arguments**, just like function arguments

Allocates memory for the EmptySet
Typechecking client code

```java
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}
class EmptySet implements IntSet { ... }
```

1. The `new` expression has type `EmptySet` implements `IntSet`

2. OK to assign an `EmptySet` to an `IntSet`, because `EmptySet` implements `IntSet`

IntSet `s = new EmptySet();`

3. `s` has type `IntSet`. We check that `IntSet` defines a `contains` method.

4. The `contains()` method in `IntSet` accepts an `int` argument so the actual argument is OK

5. `contains()` returns a `boolean`, which we can assign safely to `f`

```java
boolean f = s.contains(0); // false
```
Typechecking: What Could Go Wrong?

interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}
class EmptySet implements IntSet {
    ...
}

1. Can’t instantiate an interface; its methods are undefined.

2. Can’t assign an IntSet to an EmptySet because IntSet is not a subtype of (i.e. does not implement) EmptySet

3. s has type EmptySet. But EmptySet does not define a contains method

4. Even if we spell contains correctly, the method takes an int argument, and String is not a subtype of int

5. contains() returns a boolean, which is not a subtype of int (unlike in C)

int f = s.contains("hello"); // false

EmptySet s = new IntSet();
Executing client code

```
IntSet s = new EmptySet();
boolean f = s.contains(0);
boolean t = s.isSubsetOf(s); // true
```
Executing client code

Look at the object s points to. It keeps track of its class: EmptySet

Method Stack

```java
class EmptySet implements IntSet {
    boolean contains(int element) {
        return false;
    }
    ...
}
```

IntSet s = new EmptySet();
boolean f = s.contains(0);
boolean t = s.isSubsetOf(s); // true

EmptySet defines contains(); we call this method implementation
IntSet s = new EmptySet();

boolean f = s.contains(0);

boolean t = s.isSubsetOf(s); // true
Implementing a Singleton Set

- Several classes can implement the same interface
  - Instances of these classes can all work together
  - A key strength of objects compared to alternatives such as ADTs

```java
class SingletonSet implements IntSet {
    int member;

    SingletonSet(int element) {
        member = element;
    }

    boolean contains(int e) {
        return member == e;
    }

    boolean isSubsetOf(IntSet otherSet) {
        return otherSet.contains(member);
    }
}
```

A field stores the member of the set

A constructor method initializes the fields
Implicit Constructors

- If you don’t define a constructor, Java generates one for you
  - It has no return type and is named after the class
    - Just like all constructors
  - It has no arguments
  - Fields (if any) are initialized to default values
    - 0 for numeric values
    - false for boolean variables
    - null for reference (pointer) variables

```java
class EmptySet implements IntSet {
    /** This is equivalent to the auto-generated constructor */
    public EmptySet() {}

    public boolean contains(int element) { return false; }
    public boolean isSubsetOf(IntSet otherSet) {
        return true;
    }
}
```
class SingletonSet implements IntSet {
    int member;

    SingletonSet(int element) { member = element; }
    boolean contains(int e) { return member == e; }
    boolean isSubsetOf(IntSet otherSet) {
        return otherSet.contains(member);
    }
}

// client code
SingletonSet s = new SingletonSet(5);
if (s.member <= 5) s.member++;
Hiding Fields

```java
class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }
    public boolean contains(int e) { return member == e; }
    public boolean isSubsetOf(IntSet otherSet) {
        return otherSet.contains(member);
    }
}
```

// client code
SingletonSet s = new SingletonSet(5);
if (s.member <= 5) s.member++;

Note: all methods in an interface are implicitly `public`

Discussion: when is it useful to have a `private` method?
Using Sets Together

```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?
Using Sets Together

Method Stack

```
main()
s1
s2
temp
```

```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?
Using Sets Together

Method Stack

main()
s1
s2
temp
e : EmptySet
s : SingletonSet
member = 5

IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));

What does this program print?
Using Sets Together

**Dynamic Dispatch:** determine which method to call based on the runtime class of the object.

**Polymorphism** ("many forms"): Sets can take two forms, and the behavior of a set depends on which form it takes.

```java
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

- **Method Stack**
  - `main()`
    - `s1`
    - `s2`
    - `temp`

- `s : SingletonSet`
  - `member = 5`

- **s1 points to s.**
  - `s` is of class `SingletonSet`
  - `SingletonSet.contains()` is called, printing true

- **s2 points to e.**
  - `e` is of class `EmptySet`
  - `EmptySet.contains()` is called, printing false
Adding Unions

```java
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
    IntSet union(IntSet otherSet);
}

class UnionSet implements IntSet {
    private IntSet set1;
    private IntSet set2;
    public UnionSet(IntSet s1, IntSet s2) {
        set1 = s1; set2 = s2; }
    public boolean contains(int elem) {
        return ...
    }
    public boolean isSubsetOf(IntSet otherSet) {
        return ...
    }
    public IntSet union(IntSet otherSet) {
        return ...
    }
}
```
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
    IntSet union(IntSet otherSet);
}

class UnionSet implements IntSet {
    private IntSet set1;
    private IntSet set2;
    public UnionSet(IntSet s1, IntSet s2) {
        set1 = s1; set2 = s2;
    }
    public boolean contains(int elem) {
        return set1.contains(elem) || set2.contains(elem);
    }
    public boolean isSubsetOf(IntSet otherSet) {
        return set1.isSubsetOf(elem) && set2.isSubsetOf(elem);
    }
    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet);
    }
}
When we refer to a locally-declared field or method, we are implicitly looking in the receiver object this.

The this keyword refers to the current object.

class UnionSet is a **Composite**—an object that groups other objects, while behaving just like the objects it groups. For example, you can make a UnionSet out of UnionSets.

```java
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
    IntSet union(IntSet otherSet);
}

class UnionSet implements IntSet {
    private IntSet set1;
    private IntSet set2;
    public UnionSet(IntSet s1, IntSet s2) {
        this.set1 = s1; this.set2 = s2; }
    public boolean contains(int elem) {
        return set1.contains(elem) || this.set2.contains(elem); }
    public boolean isSubsetOf(IntSet otherSet) {
        return set1.isSubsetOf(elem) && set2.isSubsetOf(elem); }
    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet); }
}
```
Another Look at Interfaces
Contracts and Clients

- Contract of service provider and client
  - Interface specification
  - Functionality and correctness expectations
  - Performance expectations
  - Hiding of respective implementation details
  - “Focus on concepts rather than operations”
interface IntSet {
    /** @return true if element is in this set */
    boolean contains(int element);

    /** @return true if otherSet is a subset of this set */
    boolean isSubsetOf(IntSet otherSet);

    /** @return a new set representing the union of this set
     *     and otherSet */
    IntSet union(IntSet otherSet);
}
Java interfaces and classes

Object-orientation

1. Organize program functionality around kinds of abstract “objects”
   • For each object kind, offer a specific set of operations on the objects
   • Objects are otherwise opaque
     • Details of representation are hidden
   • “Messages to the receiving object”

2. Distinguish interface from class
   • Interface: expectations
   • Class: delivery on expectations (the implementation)

3. Explicitly represent the taxonomy of object types
   • This is the “inheritance hierarchy”
     • A square is a shape
Implementation of interfaces

- Classes can implement one or more interfaces.

```java
public class SingletonSet implements IntSet, Cloneable {...}
```

**Semantics**
- **Must provide code** for all methods in the interface(s)

**Best practices**
- Define an interface whenever there may be multiple implementations of a concept
- Variables should have **interface type**, not class type
  ```java
  int sum(UnionSet set) { ... // preferably no
  int sum(IntSet set) { ... // yes!
  ```
Classes and Interfaces

```java
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}

class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }

    public boolean contains(int e) { return member == e; }
    public boolean isSubsetOf(IntSet otherSet) {
        return otherSet.contains(member);
    }

    // OK to define additional public methods in the class
    public int getMember() { return member; }
}
```
Two ways to put a new empty list into a variable

```
IntSet s = new SingletonSet(4);
SingletonSet ss = new SingletonSet(3);
int i = ss.getMember();    // OK
int j = s.getMember();    // error: no method getMember in IntSet
```
Object Identity & Object Equality
Object identity vs. equality

• There are two notions of equality in OO
  - The *same object*. References are the same.
  - Possibly different objects, but equivalent content
    • From the client perspective!! The actual internals might be different

```
String s1 = new String ("abc");
String s2 = new String ("abc");
```

• There are two string objects, s1 and s2.
  • The strings are are equivalent, but the references are different

```
if (s1 == s2) { *same object* } else { *different objects* }
if (s1.equals(s2)) { *equivalent content* } else { *not* }
```

• An interesting wrinkle: *literals*

```
String s3 = "abc";
String s4 = "abc";
```

• These are true: s3==s4. s3.equals(s2). s2 != s3.
Encore: Polymorphism Example 2
Functional Lists of Integers

- Some operations we expect to see:
  - **create** a new list
    - empty, or by adding an integer to an existing list
  - return the **size** of the list
  - **get** the $i^{th}$ integer in the list
  - **concatenate** two lists into a new list

- **Key questions**
  - How to **implement** the lists?
    - Many options
      - Arrays, linked lists, etc
      - How to hide the details of this choice from client code?
        - Why do this?
  - How to state **expectations**?
    - A variable $v$ can reference a list of integers
Interfaces – stating **expectations**

- The IntList interface

```java
public interface IntList {
    int size();
    int get(int n);
    IntList concatenate(IntList otherList);
    String toString();
}
```

- The declaration for `v` ensures that any object referenced by `v` will have implementations of the methods `size`, `get`, `concatenate`, and `toString`

```java
IntList v = ...

int len = v.size();
int third = v.get(2);
System.out.println(v.toString());
```
## Implementing lists

- Two options (among many):
  - Arrays
    
    
    
    | 1 | 3 | 7 | 5 | 11 | 13 | 6 | 42 |
    |
    |
    |
    |
    |
    |
    |

  - Linked lists

    
    
    
    | 1 | 3 | 9 | 4 | empty |
    |
    |
    |
    |
    |

- Operations:

  - create a new **empty** list
  - return the **size** of the list
  - return the $i^{th}$ integer in the list
  - create a list by adding to the **front**
  - concatenate two lists into a new list

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>create a new <strong>empty</strong> list</td>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>return the <strong>size</strong> of the list</td>
<td>const</td>
<td>linear</td>
</tr>
<tr>
<td>return the $i^{th}$ integer in the list</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>create a list by adding to the <strong>front</strong></td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>concatenate two lists into a new list</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
An inductive definition

- The size of a list $L$ is
  - 0 if $L$ is the empty list
  - $1 + \text{size of the tail of } L$ otherwise
Implementing Size

```java
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...
}
```

```
public class IntListCell implements IntList {
    public int size() {
        return 1 + next.size();
    }
    ...
}
```
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...
}

Base case

public class IntListCell implements IntList {
    private int value;
    private IntListCell next;

    public int size() {
        return 1 + next.size();
    }
    ...
}

Type is wrong! May be a cell or an empty list!

Inductive case
List Representation (FIXED!)

```java
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...
}
```

**Base case**

```java
public class IntListCell implements IntList {
    private int value;
    private IntList next;

    public int size() {
        return 1 + next.size();
    }
    ...
}
```

**Inductive case**

Interface type provides needed flexibility.
List Constructors

```java
public class EmptyIntList implements IntList {
    public EmptyIntList() {
        // nothing to initialize
    }
    ...
}

public class IntListCell implements IntList {
    public IntListCell(int val, IntList next) {
        this.value = val;
        this.next = next;
    }

    private int value;
    private IntList next;
    ...
}
```

Java gives us this default constructor for free if we don’t define any constructors.
In main(...)

IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
Some Client Code

```java
In main(…) IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);

public IntListCell(int value, IntList next) {
    // value is 5, next is emptyList
    this.value = value; // this is fiveList
    this.next = next;
}
```

In main(...)

IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
Some Client Code

In main(...)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
Implementing Concatenate

```java
public class EmptyIntList implements IntList {
    public IntList concatenate(IntList other) {
        return other;
    }
    ...
}

public class IntListCell implements IntList {
    public IntList concatenate(IntList other) {
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
    ...
}
```

**Two concatenate methods – which do we use?**
In main(...)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
Method dispatch (simplified)

Example:

```
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList);
```

- **Step 1 (compile time):** determine what type to look in
  - Look at the static type (IntList) of the receiver (fourList)

- **Step 2 (compile time):** find the method in that type
  - Find the method in the class with the right name
    - Later: there may be more than one such method

```
IntList concatenate(IntList otherList);
```

- Keep the method only if it is *accessible*
  - e.g. remove private methods
- Error if there is no such method
Method dispatch (simplified)

Example:

List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList);

- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the heap and get its class

- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method we found statically (step 2)

    public IntList concatenate(IntList other) {
      IntList newNext = next.concatenate(other);
      return new IntListCell(value, newNext);
    }

- Invoke the method
Some Client Code

```java
class IntListCell {
    public IntList concatenate(IntList other) {
        // this is fourList, other is fiveList
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
}

List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?
```
A Question for You!

1. What concatenate method is called next?

2. What does the final heap look like?

```java
class IntListCell {
    public IntList concatenate(IntList other) {
        // this is fourList, other is fiveList
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
}
```

List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?
In main(…)
List emptyList = \texttt{new} EmptyIntList();
List fiveList = \texttt{new} IntListCell(5, emptyList);
List fourList = \texttt{new} IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?

\begin{itemize}
\item emptyList : EmptyIntList
\item fiveList : IntListCell value = 5 next
\item fourList : IntListCell value = 4 next
\item fourFive : IntListCell value = 4 next
\end{itemize}

\textbf{Answers}

\textit{fourList.next} points to an object of class \texttt{EmptyIntList}. Therefore \texttt{EmptyIntList.concatenate()} is called.
Toad’s Take-Home Messages

- **OOP** – code is organized code around *kinds of things*
  - **Objects** correspond to things/concepts of interest
  - Objects embody:
    - State – held in **fields**, which hold or reference data
    - Actions – represented by **methods**, which describe operations on state
    - **Constructors** – how objects are created
  - A **class** is a family of similar objects
  - An **interface** states expectations for classes and their objects
  - Polymorphism and Encapsulation as key concepts
    - Allow different implementations behind a common interface

- **Objects reside in the heap**
  - They are accessed by **reference**, which gives the objects **identity**
  - **Dispatch** is used to choose a method implementation based on the **class** of the **receiver**
  - Equivalence (**equals**) does not mean the same object (==)