



COMP 303

Lecture 12

Unit testing II

Winter 2025

slides by Jonathan Campbell

© Instructor-generated course materials (e.g., slides, notes, assignment or exam questions, etc.) are protected by law and may not be copied or distributed in any form or in any medium without explicit permission of the instructor. Note that infringements of copyright can be subject to follow up by the University under the Code of Student Conduct and Disciplinary Procedures.

Announcements

- My office hours moved to tomorrow (will change weekly)
 - Always check sheet.
- Proposal grades coming this Fri/Sat
- Project server

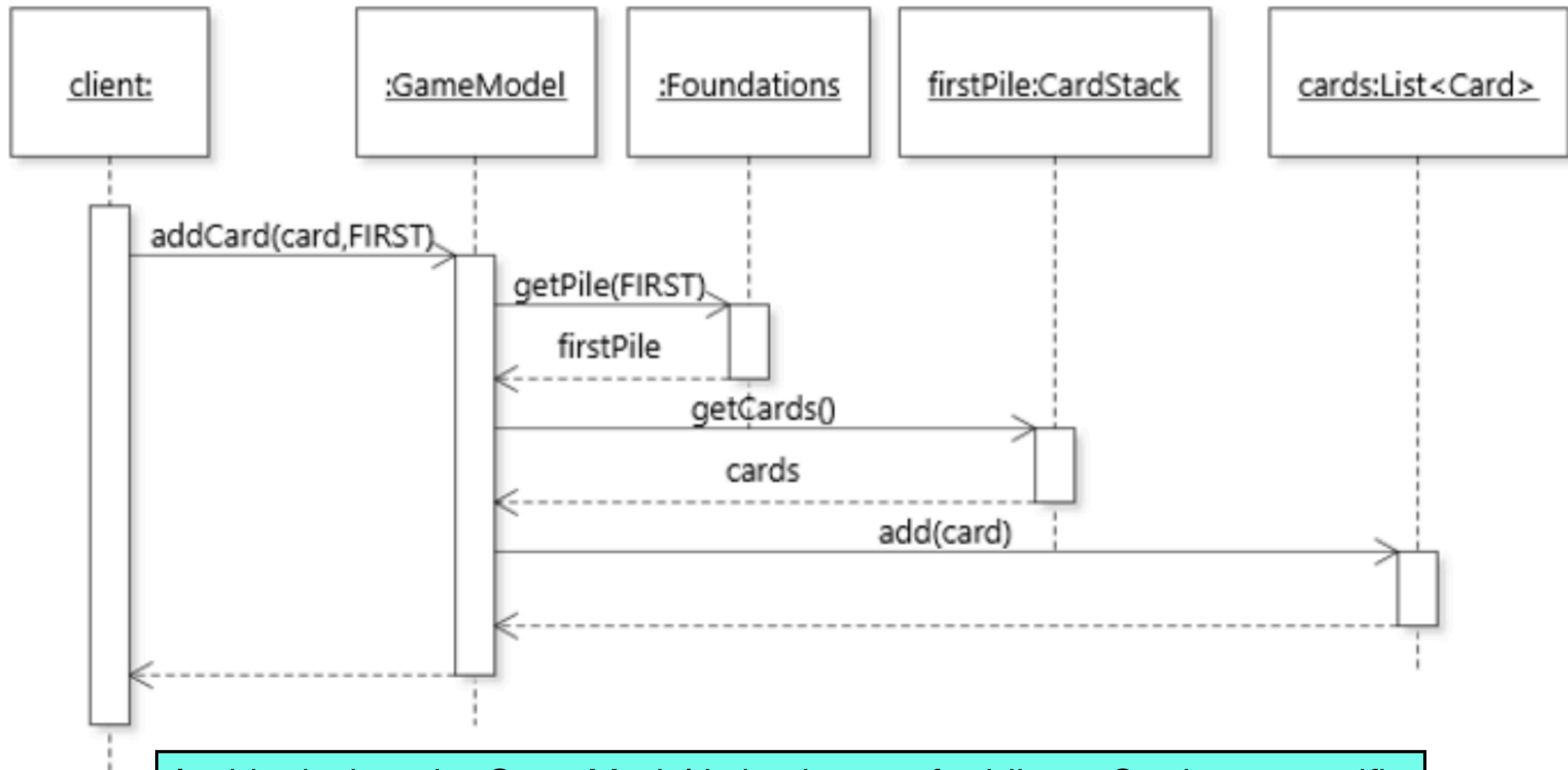
Unit testing

- Unit testing
 - Introduction to unit testing (last class)
 - **Metaprogramming (reflection) and the Class<T> type**
 - **Stubs & test coverage**
 - Unit tests in Python (next class)

Recap

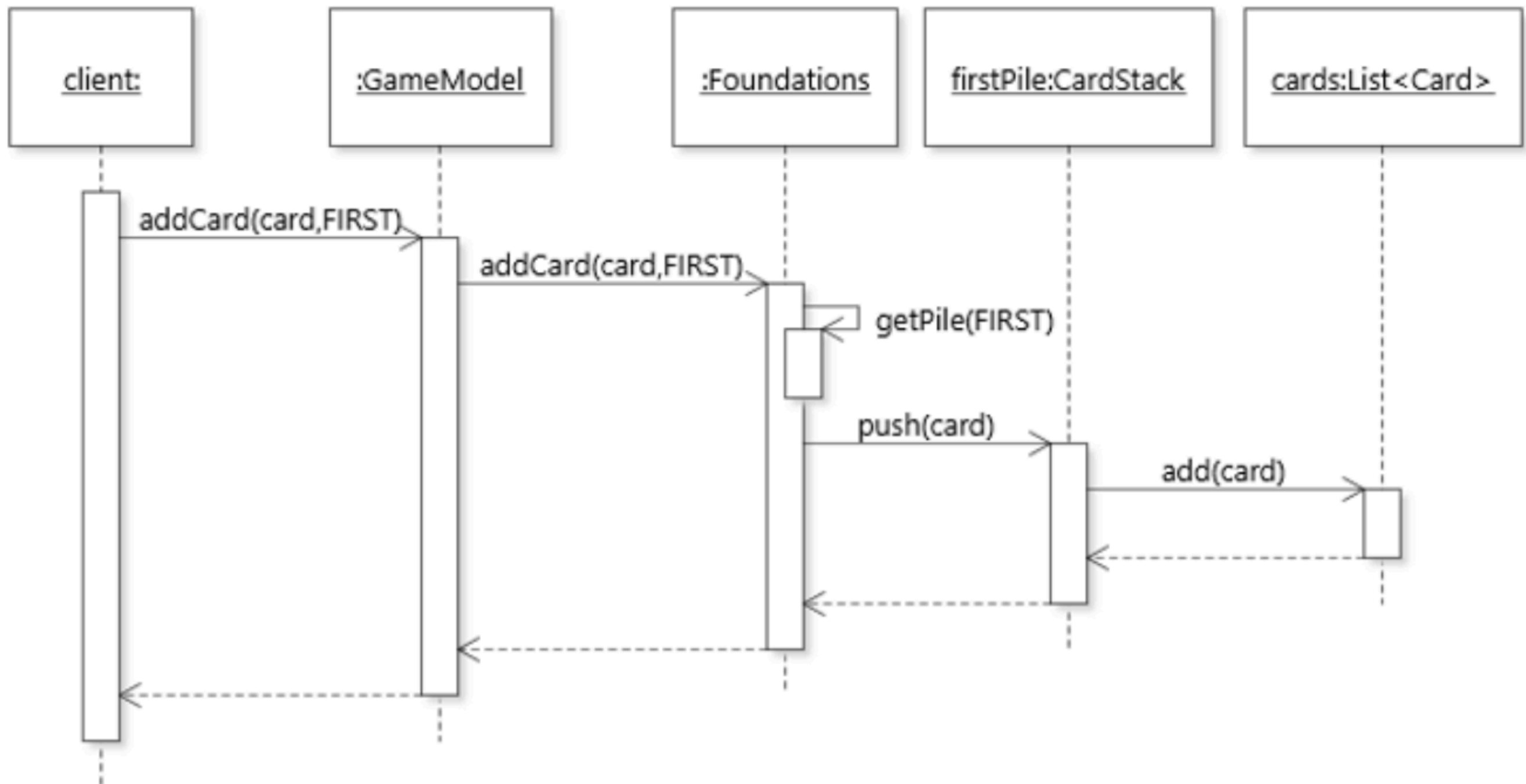
Delegation chains

```
aFoundations.getPile(FIRST).getCards().add(pCard);
```



In this design, the GameModel is in charge of adding a Card to a specific List<Card>, even though that field is stored several layers deep.

Law of Demeter



Law of Demeter

- Code in a method should only access:
 - the instance variables of its implicit parameter (this);
 - the arguments passed to the method;
 - any new object(s) created within the method;
 - (if need be) globally available objects.

Testing & unit testing

- Testing: Check that code works properly.
- Unit testing: Write **little** tests, one per every behaviour / edge case ("unit") of a method.
- When we change the method later on, we can re-run all the tests that we've written, to make sure it still works.

Components of a unit test

- UUT: The **unit under test**. E.g., the method.
- Input data: The arguments to the method. Also, the implicit argument (this/self).
- Expected result ("oracle"): what the method should return.

Unit testing frameworks

- Automatic software testing is typically done using a unit testing framework.
- These frameworks automate running the tests, reporting the results of tests, and have other nice things.
- In Java, the most popular such framework is **JUnit**. We will cover the basics of it.

Unit tests in JUnit

```
public class AbsTest {  
    @Test  
    public void testAbs_Positive() {  
        assertEquals(5, Math.abs(5));  
    }  
    @Test  
    public void testAbs_Negative() {  
        assertEquals(5, Math.abs(-5));  
    }  
    @Test  
    public void testAbs_Max() {  
        assertEquals(Integer.MAX_VALUE,  
            Math.abs(Integer.MIN_VALUE));  
    }  
}
```

@Test: indicates that the annotated method should be run as a unit test

assert method: Will check the given argument(s), and report a failure if appropriate.


Output of unit tests

Finished after 0.094 seconds


Runs: 3/3  Errors: 0  Failures: 1



▼  AbsTest [Runner: JUnit 5] (0.010 s)

 testAbs_Max (0.007 s)

 testAbs_Positive (0.000 s)

 testAbs_Negative (0.001 s)

 Failure Trace



  java.lang.AssertionError: expected: <2147483647> but was: <-2147483648>

 at designbook/chapter5.AbsTest.testAbs_Max(AbsTest.java:36)













 at java.base/java.util.stream.ForEachOps\$ForEachOp\$OfRef.accept(ForEachOps.java:183)

Unit testing II

Organizing tests

- We typically put all tests for a class inside its own class. E.g., to test the class Deck, we would put the tests in a class TestDeck.
- We then put all these test classes in their own folder, and try to mimic the folder hierarchy of the original code.
- Classes with the same package name are considered to be in the same package scope, and thus can refer to non-public (but not private) members of other classes with the same package name.

Organizing tests

- ▼  > JetUML [JetUML master]
 - ▼  src
 - >  ca.mcgill.cs.jetuml
 - >  ca.mcgill.cs.jetuml.annotations
 - ▼  ca.mcgill.cs.jetuml.application
 - >  ApplicationResources.java
 - >  Clipboard.java
 - ▼  test
 - >  ca.mcgill.cs.jetuml
 - ▼  ca.mcgill.cs.jetuml.application
 - >  TestApplicationResources.java
 - >  TestClipboard.java

Organizing tests

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
<code>public</code>	✓	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✓	✗
<i>no modifier</i>	✓	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗	✗

Metaprogramming

- Typically, when we write code, we operate on data that represent real-world objects (like Cards, or bank records, or shapes, etc.).
- When we write code that operates on other code (e.g., writing code to test other classes, methods, fields, etc.), it is called **metaprogramming**.
- In Java, metaprogramming is called **reflection** ("the ability to see yourself in the mirror").

Introspection

- Introspection is often the first step in metaprogramming.
- It is the process of obtaining a reference to an object that represents a piece of code. E.g., obtaining a reference to a field, method or class which you want to analyze.

The Class<T> type

- Class<T> is the primary mechanism for introspection in Java.
- Every Java class is represented at runtime by an instance of Class<T>, where T is the class name.
- An object of type Class<T> holds metadata about the class that it represents; e.g., all the methods/fields inside.

Obtaining a Class<T> object

- If we know exactly the class for which we want the Class<T> object, we can simply use a class literal:

```
Class<Card> cardClass1 = Card.class;
```

- Or, if we already have an instance of the class, we can call getClass on the instance to get the Class:

```
Card card = Card.get(Rank.ACE, Suit.CLUBS);  
Class<?> cardClass2 = card.getClass();
```

Obtaining a Class<T> object

- If we only have the name of the class as a String, we can call `Class.forName`:

```
try {  
    String fullyQualifiedName = "cards.Card";  
    Class<Card> cardClass =  
        (Class<Card>) Class.forName(fullyQualifiedName);  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

- Because the String we pass in may not correspond to a valid class, we have to catch the appropriate exception.

Obtaining a Class<T> object

- Here's an example that passes a string to Class.forName, and uses Class<?>, where ? is a type wildcard, meaning that the class could be of any type (Card, Deck, etc.).

```
public static void main(String[] args) {  
    try {  
        Class<?> theClass = Class.forName(args[0]);  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
}
```

Class<T> objects

- Instances of Class are unique:

```
System.out.println(cardClass1 == cardClass2);  
// prints True
```

Using a Class<T> object

```
for (Method method : String.class.getDeclaredMethods()) {  
    System.out.println(method.getName());  
}
```


Using metaprogramming

- With metaprogramming in Java, we can:
 - change the accessibility of class members;
 - set field values;
 - create new instances of objects;
 - invoke methods;
 - and more.

Creating a new instance

```
try {  
    Card card1 = Card.get(Rank.ACE, Suit.CLUBS);  
  
    Constructor<Card> cardConstructor =  
        Card.class.getDeclaredConstructor(Rank.class, Suit.class);  
    cardConstructor.setAccessible(true);  
  
    Card card2 = cardConstructor.newInstance(Rank.ACE, Suit.CLUBS);  
  
    System.out.println(card1 == card2);  
} catch (ReflectiveOperationException e) {  
    e.printStackTrace();  
}
```

get the constructor

make it public

call the constructor

Modifying fields

```
try {  
    Card card = Card.get(Rank.TWO, Suit.CLUBS);  
  
    Field rankField = Card.class.getDeclaredField("aRank");  
    rankField.setAccessible(true);  
  
    rankField.set(card, Rank.ACE);  
  
    System.out.println(card);  
} catch (ReflectiveOperationException e) {  
    e.printStackTrace();  
}
```

get the field

make it public

set it to Ace for the given card

Basic principles for unit tests

- **speed:** unit tests should avoid things like intensive device I/O or network access, because they are run often.
- **independence:** each unit test should execute in isolation; it should not depend on another test running first, because we often like to execute a single test, and testing frameworks don't guarantee the order of execution of tests

Basic principles for unit tests

- **repeatable**: unit tests should produce the same result in different environments; they should not depend on environment-specific properties.
- **focus**: tests should exercise and verify a slice of code execution behaviour that is narrow as reasonably possible, to be able to pinpoint exactly what is wrong.
- **readability**: the UUT, input data and oracle (expected result), and the rationale, should all be easily identifiable.

Example method to test

```
public class FoundationPile {  
    public boolean isEmpty() { ... }  
    public Card peek() { ... }  
    public Card pop() { ... }  
    public void push(Card pCard) { ... }  
  
    public boolean canMoveTo(Card pCard) {  
        assert pCard != null;  
        if (isEmpty()) {  
            return pCard.getRank() == Rank.ACE;  
        }  
        else  
        {  
            return pCard.getSuit() == peek().getSuit() &&  
                pCard.getRank().ordinal() == peek().getRank().ordinal()+1;  
        }  
    }  
}
```

Returns true if we can move the input card to the top of the pile respecting the rules of Solitaire.

Example test

```
public class TestFoundationPile {  
    @Test  
    public void testCanMoveTo_Empty() {  
        FoundationPile emptyPile = new FoundationPile();  
        Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);  
        Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);  
        assertTrue(emptyPile.canMoveTo(aceOfClubs));  
        assertFalse(emptyPile.canMoveTo(threeOfClubs));  
    }  
}
```

Respects the unit test principles:
fast, independent, focused, repeatable, readable.

Example test

```
public class TestFoundationPile {  
    @Test  
    public void testCanMoveTo_Empty() {  
        FoundationPile emptyPile = new FoundationPile();  
        Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);  
        Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);  
        assertTrue(emptyPile.canMoveTo(aceOfClubs));  
        assertFalse(emptyPile.canMoveTo(threeOfClubs));  
    }  
}
```

Respects the unit test principles:
fast, independent, focused, repeatable, readable.

Problem: only checks a single path through the method:
moving a card onto an empty pile.

Example test

```
public class TestFoundationPile {  
    @Test  
    public void testCanMoveTo_NotEmptyAndSameSuit() {  
        Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);  
        Card twoOfClubs = Card.get(Rank.TWO, Suit.CLUBS);  
        Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);  
        FoundationPile pileWithOneCard = new FoundationPile();  
        pileWithOneCard.push(aceOfClubs);  
        assertTrue(pileWithOneCard.canMoveTo(twoOfClubs));  
        assertFalse(pileWithOneCard.canMoveTo(threeOfClubs));  
    }  
}
```

This test checks a different path: when the pile already contains an Ace of Clubs.

Example test

```
public class TestFoundationPile {  
    @Test  
    public void testCanMoveTo_NotEmptyAndSameSuit() {  
        Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);  
        Card twoOfClubs = Card.get(Rank.TWO, Suit.CLUBS);  
        Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);  
        FoundationPile pileWithOneCard = new FoundationPile();  
        pileWithOneCard.push(aceOfClubs);  
        assertTrue(pileWithOneCard.canMoveTo(twoOfClubs));  
        assertFalse(pileWithOneCard.canMoveTo(threeOfClubs));  
    }  
}
```

This test checks a different path: when the pile already contains an Ace of Clubs.

Problem: lots of **DUPLICATED CODE** (anti-pattern).

Example test

Fields of the test class will always be reset to initial values when a test executes (re-instantiation).

```
public class TestFoundationPile
{
    private static final Card ACE_CLUBS = Card.get(Rank.ACE, Suit.CLUBS);
    private static final Card TWO_CLUBS = Card.get(Rank.TWO, Suit.CLUBS);
    private static final Card THREE_CLUBS = Card.get(Rank.THREE, Suit.CLUBS);
    private FoundationPile aPile = new FoundationPile();
    @Test
    public void testCanMoveTo_Empty() {
        assertTrue(aPile.canMoveTo(ACE_CLUBS));
        assertFalse(aPile.canMoveTo(THREE_CLUBS));
    }
    @Test
    public void testCanMoveTo_NotEmptyAndSameSuit() {
        aPile.push(ACE_CLUBS);
        assertTrue(aPile.canMoveTo(TWO_CLUBS));
        assertFalse(aPile.canMoveTo(THREE_CLUBS));
    }
}
```

These baseline objects that are later used in tests are called the **test fixture**.

(similar to how a physical fixture in a lab holds something)

Tests are now much more readable.

Testing for exceptional conditions

```
class FoundationPile {  
    boolean isEmpty() { ... }  
  
    /*  
    * @return The card on top of the pile.  
    * @pre !isEmpty()  
    */  
    Card peek() { ... }  
}
```

This method uses design by contract (preconditions specified), so we should not write a unit test for this condition, since the behaviour is unspecified.

Testing for exceptional conditions

```
class FoundationPile {  
    boolean isEmpty() { ... }  
    /*  
    * @return The card on top of the pile.  
    * @throws EmptyStackException if isEmpty()  
    */  
    Card peek() {  
        if (isEmpty()) {  
            throw new EmptyStackException();  
        }  
        ...  
    }  
}
```

This method throws an exception under some condition.
We should write a unit test to verify this case.
(If it doesn't throw the exception, then it is faulty.)

Testing for exceptional conditions

```
@Test
public void testPeek_Empty() {
    assertThrows(EmptyStackException.class, () -> aPile.peek());
}
```

Lambda expression

Encapsulation & unit testing

- Sometimes, when writing a unit test, we may need some functionality that is not part of the interface of the class being tested.
- For example, we may need to calculate the size of a FoundationPile, but there is no method defined in the class to do that for us.
 - So, instead of adding such a method to the class, we can add the method to the testing class (TestFoundationPile), so that the actual class doesn't have methods that are only needed in testing (i.e., so that it doesn't get "polluted").

References

- Robillard ch. 5.3-5.7 (p. 104-117)
 - Exercises #1-7: <https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter5.md>

Coming up

- Next lecture:
 - More about unit testing
 - Inheritance