CS 5044
Object-Oriented Programming with Java

Q&A Session

Functional Programming (FP) in general

- FP concepts were fundamental in some of the earliest programming languages
  - FP ideally works with purely stateless functions and immutable data
    - Functions process existing data and return new (entirely deterministic) data
    - There are never any side effects or mutable variables
  - "Objects" in FP are just structures (collections) of tagged immutable primitives
    - Other than primitives, pretty much all data is some kind of object or array
    - Importantly, functions are considered data as well
      - However, the data is not encapsulated within any functions or objects
- FP doesn't produce programs that run as a recipe of specific steps
  - Data represents state at some point in time, then functions are applied to the data
    - Functions are evaluated, either with existing or new data, generating more new data
    - Old data can be discarded (thus effectively replaced by new data) but not mutated
  - FP programs have more in common with spreadsheets than with imperative programs
- This paradigm basically turns object-oriented programming inside-out
  - Much of this is completely incompatible with object-oriented programming, yet...
    - Several of these features are still entirely relevant, and even missed when absent
    - The FP approaches fit extremely well in parallel/distributed environments

Functional Programming in Java

- Three key FP concepts were introduced, starting with Java 8 (2014):
  - Lambda Expressions
    - Much cleaner syntax for defining anonymous inner classes of functional interfaces
    - Allows functions (methods) to be treated much more readily as data (but not really)
  - Method References
    - Simplified syntax for certain lambda expression where parameters can be inferred
  - Aggregate Operations
    - Stream-based processing of entire collections, without any explicit loops
- All of the above rely upon a specific definition of a "functional" interface
  - Any interface that has exactly one abstract method (that isn't already declared in `Object`)
  - By convention, we annotate such interfaces as `@FunctionalInterface`
    - This asks the compiler to enforce the definition
- Lambda Expressions and Method References are very common in GUI code
  - In Swing, these provide nice enhancements to substantially clean up verbose syntax
  - In JavaFX, these constructs are considered fundamental to the event model

Events: Classic approach

- From ButtonDemo Homework

```
public class ButtonDemo extends JPanel implements ActionListener
{
    JButton button;

    // ... other methods ...

    public ButtonDemo()
    {
        // other code in constructor
        button.addActionListener(this); // add *this* as the listener
    }

    public void actionPerformed(ActionEvent ae) // method called in *this* upon click
    {
        button.setText("Ouch!");
    }
}
```

Events: Named inner class

- From ButtonDemo Homework

```
public class ButtonDemoNamedInner extends JPanel // <-- Does NOT need to implement ActionListener now
{
    JButton button;

    // ... other methods ...

    public ButtonDemoNamedInner()
    {
        // other code in constructor

        public class ButtonClickListener implements ActionListener // <-- *named* inner class
        {
            @Override
            public void actionPerformed(ActionEvent ae)
            {
                button.setText("Ouch!");
            }
        }
        ButtonClickListener myListener = new ButtonClickListener(); // <-- new instance of named inner class
        button.addActionListener(myListener); // <-- add instance as the listener
    }

    // no actionPerformed() method is needed in this class
}
```

Events: Named inner class (anonymous instance)

- From ButtonDemo Homework

```
public class ButtonDemoNamedInner extends JPanel
{
    JButton button;

    // ... other methods ...

    public ButtonDemoNamedInner()
    {
        // other code in constructor

        public class ButtonClickListener implements ActionListener // <-- *named* inner class
        {
            @Override
            public void actionPerformed(ActionEvent ae)
            {
                button.setText("Ouch!");
            }
        }
        button.addActionListener(new ButtonClickListener()); // <-- add *anonymous* instance
    }
}
```

Events: Anonymous inner class

- From ButtonDemo Homework

```
public class ButtonDemoAnonInner extends JPanel
{
    JButton button;

    // ... other methods ...

    public ButtonDemoAnonInner()
    {
        // other code in constructor

        button.addActionListener(new ActionListener()   // <-- *anonymous* inner class
        {
            @Override
            public void actionPerformed(ActionEvent ae)
            {
                 button.setText("Ouch!");
            }
        });
    }
}
```

Events: Lambda expression

- From ButtonDemo Homework

```
public class ButtonDemoAnonInner extends JPanel
{
    JButton button;

    // ... other methods ...

    public ButtonDemoAnonInner()
    {
        // other code in constructor

        button.addActionListener((ActionEvent ae) ->     // <-- lambda expression
        {
            button.setText("Ouch!");
        });
    }
}
```

Events: Lambda expression

- From ButtonDemo Homework

```
public class ButtonDemoAnonInner extends JPanel
{
    JButton button;

    // ... other methods ...

    public ButtonDemoAnonInner()
    {
        // other code in constructor

        button.addActionListener((ActionEvent ae) ->     // <-- lambda expression
        {
            button.setText("Ouch!");
        });
    }
}
```

- Usually simplified even further:

```
        button.addActionListener(ae -> button.setText("Ouch!"));
```

  - see next slide for available options

Lambda parameters, braces, and semicolons

- Lambda expression parameters
  - Most basic form is the same as for the normal method being implemented:
    - `(int i, String s) -> some expression using i and s`
  - If parameter types can be inferred, the types may be omitted:
    - `(i, s) -> some expression using i and s`
  - If only one parameter, and its type can be inferred, you may omit the parentheses:
    - `i -> some expression using i`
  - If no parameters, you must use empty parentheses:
    - `() -> some expression`
- Lambda expression braces and semicolons
  - Most basic form includes braces, as with the normal method (here shown on one line):
    - Semicolons are required as usual when braces are present:
    - `(int i, String s) -> { String t = s.toLowerCase(); int j = t.length() + i; return t + j; }`
  - If only one statement, the braces and the trailing semicolon may be omitted:
    - Often we delegate to another method (see next slide):
    - `(int i, String s) -> doSomething(i, s)`
    - If statement is an expression, it is implicitly returned (if method return is not `void`):
    - `(int i, String s) -> s.toUpperCase() + i`

From lambda expressions to method references

- Further syntax simplification is available, but only if...
  - The lambda expression calls one method, and all lambda parameter types can be inferred
- Four variants of method references exist, covering many common cases:
  - Instance method, called on the first lambda parameter
    - Reference is class::method (where *class* is the type of the first lambda parameter)
      - `(x, y, z) -> x.someMethod(y, z)`
      - `SomeClass::someMethod`
  - Instance method, called on an explicit object
    - Reference is object::method
      - `(i, j) -> someObject.someMethod(i, j)`
      - `someObject::someMethod`
  - Static method
    - Reference is class::method
      - `(a, b) -> SomeClass.someMethod(a, b)`
      - `SomeClass::someMethod`
  - Constructor
    - Reference is class::new
      - `(s, t) -> new SomeClass(s, t)`
      - `SomeClass::new`

Aggregate operations (streams)

- Streams act as a pipeline of operations to be executed on demand
  - Iteration is entirely implicit, and may take advantage of parallel execution
  - Most evaluation is "lazy" meaning operations are only executed as needed
  - Elements within the stream are never mutated (but can be replaced or removed)
- Summary of aggregate operations
  - Source operations
    - Usually generated from existing collections or arrays via `stream()` or `Stream.of()`
    - Streams can also be generated via lazy `Stream.iterate()` and other (less common) ways
  - Intermediate operations
    - Act on a stream and return a stream, continuing the pipeline
      - Transforming operations include: `map()`
      - Reducing operations include: `filter()`, `limit()`
      - Increasing operations include: `concat()`, `flatMap()`
  - Terminal operations
    - Act on a stream, but do not return a stream; these define the end of the pipeline
      - Often we want to generate a new collection via `collect()` or aggregation via `reduce()`
        - » Aggregations via `sum()`, `min()`, `average()`, etc. are common special cases of `reduce()`
      - Sometimes we use `forEach()` to intentionally process side-effects for each element

Example: DotsAndBoxes getScores() via streams

- Here's an approach using streams and aggregate operations
  - Stream the Box values of the boxGrid
  - Transform (map) to a stream of owners
  - Retain (filter) the non-null elements
  - Collect in a Map with duplicates
    - Keys: the owner elements
    - Values: List<Integer>
      - Each element treated as 1
      - Aggregate by summation
- Code is denser, but arguably much cleaner than an iterative approach
- We can trivially use parallel processing:
  - Change stream() to parallelStream()
  - Change toMap() to toConcurrentMap()
  - There are a few other considerations
    - ...but that's another course!

```
public Map<Player, Integer> getScores() {
    return boxGrid.values().stream()
        .map(Box::getOwner)
        .filter(Objects::nonNull)
        .collect(toMap(identity(), p -> 1, Integer::sum));
}
```

Example: AI findBestPlacement() via streams

- Here's how you could implement the TetrisAI solution using streams:

```
public Placement findBestPlacement(Board board, Shape shape) {
    return shape.getRotationSet().stream()
        .flatMap(rot -> Stream.iterate(0, c -> c + 1)
                        .limit(Board.WIDTH - shape.getWidth(rot) + 1)
                        .map(col -> new Placement(rot, col)))
        .min((a, b) -> Integer.compare(
                        calculateCost(board, shape, a),
                        calculateCost(board, shape, b)))
        .orElse(null);
}
```

- Substantially less verbose than the traditional approach (with no local variables)
  - However, for most Java developers, it isn't nearly as straightforward to read
- Note that this particular implementation is computationally inefficient
  - The cost calculation of the best-so-far placement is repeated for every alternative
  - It can be fixed easily with a few minor modifications, but this serves as a nice reminder:
    - We can write poorly-performing code in many different ways
- As in the previous example, we can trivially use parallel processing
  - Again, a few additional considerations are required

Other stream examples (Map <Term, Set<Course>>)

```
Map<Term, Set<Course>> timetable = /* courses by term */

long numOnlineInstructorsIn2019 =
    timetable.entrySet().stream()
        .filter(e -> e.getKey().getYear() == 2019)
        .flatMap(e -> e.getValue().stream())
        .filter(Course::isOnline)
        .map(Course::getInstructor)
        .distinct()
        .count();
```

Equivalent non-stream code:

```
List<Course> courses2019 = new ArrayList<>();
for (Term term : timetable.keySet()) {
    if (term.getYear() == 2019) {
        for (Course course : timetable.get(term)) {
            courses2019.add(course);
        }
    }
}
Set<Person> instructors = new HashSet<>();
for (Course course : courses2019) {
    if (course.isOnline()) {
        Person prof = course.getInstructor();
        if (!instructors.contains(prof)) {
            instructors.add(prof);
        }
    }
}
long numOnlineInstructorsIn2019 = instructors.size();
```

Other stream examples (Map <Term, Set<Course>>)

```
Map<Term, Set<Course>> timetable = /* courses by term */
```

```
long numOnlineInstructorsIn2019 =
    timetable.entrySet().stream()
        .filter(e -> e.getKey().getYear() == 2019)
        .flatMap(e -> e.getValue().stream())
        .filter(Course::isOnline)
        .map(Course::getInstructor)
        .distinct()
        .count();
```

```
Map<String, Integer> fall2019OnlineCapacityByDept =
    timetable.keySet().stream()
        .filter(t -> t.getYear() == 2019 &&
                     t.getSemester() == Semester.FALL)
        .flatMap(t -> timetable.get(t).stream())
        .filter(Course::isOnline)
        .collect(toMap(
            Course::getDept,
            Course::getCapacity,
            Integer::sum
        ));
```

```
double averageFall2019OnlineCapacityPerDept =
    fall2019OnlineCapacityByDept.values().stream()
        .mapToDouble(i -> i)
        .average()
        .orElse(0);
```

```
Set<Person> instructorsWithTopAllTimeOnlineCapacity =
    timetable.values().stream()
        .flatMap(Set::stream)
        .filter(Course::isOnline)
        .collect(toMap(
            Course::getInstructor,
            Course::getCapacity,
            Integer::sum
        ))
        .entrySet().stream()
        .collect(groupingBy(
            Entry::getValue,
            TreeMap::new,
            mapping(Entry::getKey, toSet())
        ))
        .lastEntry().getValue();
```

How to incorporate functional features into solutions

- Ideally we should use the most appropriate tools for the tasks
  - So-called "pure" functional approaches are certainly possible, however...
    - You couldn't then take advantage of the object-oriented roots of the language
    - In this case you'd probably be better off using FP languages, such as Scala or Clojure
      - Both of these run on the JVM, and can interact with all Java libraries
- Lambda expressions and method references are very nice syntax simplifications
  - Feel free to use them wherever appropriate (but don't go out of your way to do so)
- Stream processing has significant trade-offs that should be carefully considered
  - Pros:
    - Great fit for certain types of relatively common problems
    - Often requires significantly less code, which tends to be far less fragile
      - Stateless deterministic algorithms are generally much easier to test as well
    - Can usually be made to work in parallel with very little effort
      - Note that this doesn't necessarily ensure better performance
  - Cons:
    - Still considered a niche feature; not nearly as widely known/used as classic iterations
    - Code can be somewhat difficult to understand, especially if you're not used to it

Project 6: Action listeners (classic technique)

- Three "new game" menu items, each of which needs an `ActionListener` assigned
- This works exactly as expected, but it's extremely verbose, fragile, and awkward
  - Each item needs an action command String, which is then detected in the listener
  - Don't use this approach!
- As an alternative, we could create additional classes, but that makes it even worse
  - Don't use this approach!
- Is there a better way?
  - Let's try using anonymous inner classes

```
public class DABPanel extends JPanel implements ActionListener {
    private JMenuBar createMenuBar() {
        // ... other code ...
        JMenuItem menuItemNewGame2 = new JMenuItem();
        menuItemNewGame2.setActionCommand("start2");
        menuItemNewGame2.addActionListener(this);

        JMenuItem menuItemNewGame3 = new JMenuItem();
        menuItemNewGame3.setActionCommand("start3");
        menuItemNewGame3.addActionListener(this);

        JMenuItem menuItemNewGame4 = new JMenuItem();
        menuItemNewGame4.setActionCommand("start4");
        menuItemNewGame4.addActionListener(this);
        // ... other code ...
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if ("start2".equals(ae.getActionCommand())) {
            startGame(2);
        } else if ("start3".equals(ae.getActionCommand())) {
            startGame(3);
        } else if ("start4".equals(ae.getActionCommand())) {
            startGame(4);
        }
    }
```

Project 6: Action listeners (anonymous inner class)

- No need for DABPanel to implement `ActionListener` now
- Each action is in immediate proximity of the component that takes the action
- No need for switching based on actions; each is self-contained
- Fairly verbose now; lots of repetition that seems like it should not be needed
- Not a bad approach, but there is still a better way...
  - Java 8 to the rescue!

```
public class DABPanel extends JPanel {
    private JMenuBar createMenuBar() {
        // ... other code ...
        JMenuItem menuItemNewGame2 = new JMenuItem();
        menuItemNewGame2.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                startGame(2);
            }
        });
        JMenuItem menuItemNewGame3 = new JMenuItem();
        menuItemNewGame3.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                startGame(3);
            }
        });
        JMenuItem menuItemNewGame4 = new JMenuItem();
        menuItemNewGame4.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                startGame(4);
            }
        });
        // ... other code ...
    }
```

Project 6: Action listeners (lambda expression)

- Use a Lambda Expression to define the anonymous inner class for each `ActionListener`
- Advantages:
  - Significantly less code
  - No action commands
  - No separate method required
  - No interface to implement
- Disadvantages:
  - Syntax is newer and may be somewhat confusing
- Overall a much cleaner and less error-prone approach

```
public class DABPanel extends JPanel {
    private JMenuBar createMenuBar() {
        // ... other code ...
        JMenuItem menuItemNewGame2 = new JMenuItem();
        menuItemNewGame2.addActionListener(e -> startGame(2));

        JMenuItem menuItemNewGame3 = new JMenuItem();
        menuItemNewGame3.addActionListener(e -> startGame(3));

        JMenuItem menuItemNewGame4 = new JMenuItem();
        menuItemNewGame4.addActionListener(e -> startGame(4));
        // ... other code ...
    }
}
```

Project 6: Action listener (classic technique)

- The drawButton needs an `ActionListener` assigned
- This works exactly as expected, but it's extremely verbose, fragile, and awkward
  - We need yet another unique action command String
  - Don't use this approach!
- As an alternative, we could create additional classes, but that makes it even worse
  - Don't use this approach!
- Let's skip the anonymous inner class and go straight to a lambda expression

```
public class DABPanel extends JPanel implements ActionListener {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.setActionCommand("draw");
        drawButton.addActionListener(this);
        // ... other code ...
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if ("draw".equals(ae.getActionCommand())) {
            handleDrawButton(ae);
        }
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }

}
```

Project 6: Action listener (lambda expression)

- Much better now
- Improvements are same as for new game menu items
- Can we do even more?
  - Yes, but only slightly...

```
public class DABPanel extends JPanel {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.addActionListener(ae -> handleDrawButton(ae));
        // ... other code ...
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }
}
```

Project 6: Action listener (method reference)

- With a Method Reference, the compile can infer all of the parameters and types
- Works exactly like the lambda expression, with just a bit less verbosity
- Not much of an improvement, but sometimes even small simplifications can make a big difference

```
public class DABPanel extends JPanel {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.addActionListener(this::handleDrawButton);
        // ... other code ...
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }
}
```