Functional Java
(+ Other Java 8 Features)

Java 8 - Key Features

- Functional Programming / Related Changes
  - Effectively Final - already discussed in previous lectures
  - Default / Static Interface Methods
  - Lambda Expressions
  - Streams API
- Optional - help for `null` safety
- DateTime API - more fluent API (and it's immutable!)
- Improved Type Inference - see here
- Unsigned Arithmetic Support
- Many Other Changes
  - See here

Default / Static Interface Methods

- Prior to Java 8, adding method definition(s) to an interface created breaking source compatibility [1].
  - This made augmenting libraries (especially heavily depended upon libraries like Collection interfaces) difficult to evolve.
- Java 8 allows developers to define methods on interfaces which have default implementations (like methods within an abstract class).
  - Updating an existing interface with a default method allows all implementations of the interface to be source compatible.
- Java 8 also allows developers to define static methods within an interface.
  - This allows developers to tie relevant code directly to where it's applicable (i.e., instead of having a `Collections` class, these methods could be directly written as static methods within the `Collection` interface).

[1] For description of source and binary compatibility, see here

Static Interface Methods

- Can add static logic to interfaces
  - By convention in Java, prior to Java 8, one would make a utility class to perform this logic. For instance, the utility methods for interface `Collection` is called `Collections`.

```
1  public interface List<T> extends java.util.List<T> {
2
3      static <T> void reverse(List<T> list) {
4          for (int i = 0, mid = list.size() >> 1, j = list.size() - 1; i < mid; i++, j--) {
5              swap(list, i, j);
6          }
7      }
8
9      static <T> void swap(List<T> list, int i, int j) {
10         list.set(i, list.set(j, list.get(i)));
11     }
12
```

Default Interface Methods

- Can add default methods to interfaces

```
1  public interface List<T> extends java.util.List<T> {
2
3      static <T> void swap(List<T> list, int i, int j) {
4          list.set(i, list.set(j, list.get(i)));
5      }
6
7      default void reverse() {
8          for (int i = 0, mid = size() >> 1, j = size() - 1; i < mid; i++, j--) {
9              swap(this, i, j);
10         }
11     }
12
13     // essentially making an optional method;
14     // adding to List but not forcing an implementation, possible if desired
15     default void shuffle() {
16         throw new UnsupportedOperationException();
17     }
```

Default Interface Methods - Inheritance

- When you extend an interface with default methods, you have the following options:
  - **Ignore**; the subinterface inherits the default method (including its implementation)
  - **Redefine**; redefining the method with a new default implementation overrides the default method from the super interface
  - **Redeclare**; re-declaring the method on the subinterface makes the method abstract (the default method from the super interface is now abstract for subinterface implementations)
- This logic is the same as normal inheritance in Java (including ‘redeclare’ for abstract classes).

Default Interface Methods - Multiple Inheritance Resolution

- In Java, one can extend from only one class but can extend from multiple interfaces.
  - Now, with default methods, a class can inherit more than one method with the same signature
- If a class inherits the same method signature from multiple super types, it must resolve the conflict according to the following rules:
  - **Classes win**. A method defined in a class or superclass takes priority over any default method declaration.
  - Else, **subinterfaces win**. The most specific subinterface default method implementation wins (If interface B extends A, B is more specific than A and so B's default methods take precedence).
  - Finally, if still ambiguous, the **inheriting class must explicitly choose**. This is done via overriding the method.

Default Interface Methods - Multiple Inheritance Resolution

- Examples

```
1  public interface Vehicle {
2
3      Integer getNumberOfWheels();
4
5  }
```

```
1  public class AbstractCar {
2
3      private static final Integer DEFAULT_NUMBER_OF_WHEELS = 4;
4
5      public Integer getNumberOfWheels() {
6          return DEFAULT_NUMBER_OF_WHEELS;
7      }
8
9  }
```

```
1  public interface Car {
2
3      Integer DEFAULT_NUMBER_OF_WHEELS = 4;
4
5      Integer DEFAULT_NUMBER_OF_DOORS = 4;
6
7      default Integer getNumberOfWheels() {
8          return DEFAULT_NUMBER_OF_WHEELS;
9      }
10
11     default Integer getNumberOfDoors() {
12         return DEFAULT_NUMBER_OF_DOORS;
13     }
14
15 }
```

Default Interface Methods - Multiple Inheritance Resolution

- What method of '`getNumberOfWheels`' is invoked?

```
1  public class Bmw extends AbstractCar implements Vehicle, Car {
2
3      public static void main(String[] args) {
4          Bmw bmw = new Bmw("Speedster");
5          System.out.printf("Number of wheels? %d%n", bmw.getNumberOfWheels());
6      }
7
8      private final String name;
9
10     public Bmw(String name) {
11         this.name = name;
12     }
13 }
```

Default Interface Methods - Multiple Inheritance Resolution

- What method of ‘`getNumberOfDoors`’ is invoked on `SportsCar`?

```
1  public interface Sedan extends Car {
2
3      Integer DEFAULT_NUMBER_OF_DOORS = 2;
4
5      @Override default Integer getNumberOfDoors() {
6          return DEFAULT_NUMBER_OF_DOORS;
7      }
8
```

```
1  public class SportsCar implements Sedan {
2
3      public static void main(String[] args) {
4          SportsCar sportsCar = new SportsCar();
5          System.out.printf("Number of doors? %d%n", sportsCar.getNumberOfDoors());
6      }
7
8  }
```

Default Interface Methods - Multiple Inheritance Resolution

- What method of ‘`getNumberOfDoors`’ is invoked on Suburban?

```
1  public interface Suv {
2
3      Integer DEFAULT_NUMBER_OF_DOORS = 4;
4
5      default Integer getNumberOfDoors() {
6          return DEFAULT_NUMBER_OF_DOORS;
7      }
```

```
1  public class Suburban implements Suv, Car {
2
3      public static void main(String[] args) {
4          Suburban suburban = new Suburban();
5          System.out.printf("Number of doors? %d%n", suburban.getNumberOfDoors());
6
7      }
```

Functional Programming

- What is it? Programming using functions where functions produce no side effects. [1]
  - Expanding this; a function must produce no side effects, never mutate data and is referentially transparent [2].
- Is this possible in Java (even with constructions from Java 8)?
  - Not precisely. Instead, Java has *functional-style* of programming.

Functional Style in Java

- A function can only mutate local variables (those defined within the function)
- Not all methods in Java are referentially transparent (e.g. `Random.nextInt`)

[1] More like a mathematical definition of function (`sin`, `log`, etc); see here for more context.

[2] Invoking it with the same input should always produce the same output.

Functional Programming Concepts

- ☕ **First Class Functions** - use functions like other values [1]
- ☕ **Higher Order Functions** - take functions as parameters and return functions [2]
- ☕ **Currying** - where a function of two (or more) arguments is seen as a function taking one argument and returning another function (or functions) of one argument [e.g. f(x,y) = (g(x)) (y)] [3]
- ☕ **Partial Application** - fixing some arguments of a function producing a function with smaller arity [e.g. $f(x,y) = (g_{partial}(5))\ (y)$] [4]
- 😞 **Tail Recursion** - because loops introduce mutation, recursion is leveraged. To avoid stack memory proportional to input, use tail recursion which is where the recursive call happens as the final act of the procedure [5]
- 🔨 **Persistent Data Structures** - immutable structures which has a notion of a previous structure once updated [6]
- 🔨 **Lazy Evaluation** - invocation of function only happens when needed [7]
- 😞 **Pattern Matching** - matching based on structure (not to be confused with regex) [8]
- 🔨 **Memoization (i.e. caching)** - because of referential transparency, function results can be cached [9]
- 😞 **Algebraic Data Types** - type formed by combining other types [10]

☕ Indicates support in Java

😞 Indicates no support in Java

🔨 Indicates programmer could create

(i.e. not natively a part of the language)

Lambda Expressions

- Java's name for adding functions as first class values.
  - The name derives from the lambda calculus
- A **lambda expression** in Java is a concise representation of an anonymous function (specifically a functional interface) which can be passed around [1]
  - Anonymous - the function does not have a name
  - Function - it is not associated with a particular class or method (but is an implementation of a functional interface)
  - Passed Around - it is a first class value
  - Concise - new syntax is introduced in Java 8 to minimize boilerplate code

[1] modified from Java 8 In Action - Urma, Fusco & Mycroft; page 40

Functional Interface

- Concept introduced in Java 8; it is an interface specifying exactly one abstract method.
  - Note, the concept is new for Java 8 but many existing interfaces already meet the definition (e.g. `Runnable`)
- All functional interfaces can be implemented via
  - A class (as we've seen)
  - An anonymous class (as we've seen)
  - A lambda expression (new to Java 8)
- Java 8 introduces an annotation `@FunctionalInterface` to provide metadata for those interfaces being defined as functional interfaces
  - For every functional interface you define, annotate with `@FunctionalInterface`. You do this for the same reason as using `@Override`, the programming intent can then be verified by the compiler.

Lambda Expression Syntax

- Lambda expressions follow two patterns of syntax.
  - (parameters) -> expression
  - (parameters) -> { statements; }
- **Parameters**
  - A comma delimited list of named parameters surrounded by parenthesis. The type is optional but if included comes prior to the named parameter (e.g. **String value**)
  - If only one parameter is used, the parenthesis are optional.
- **Arrow**
  - Separates the parameters from the body
- **Body**
  - Either a single **expression** or one to many **statements**.
    - Note, an expression is evaluated for the return type. If you explicitly specify '`return`' you must enclose in braces (as it's then considered a statement)
- Note, the return type is not explicitly specified but inferred.

Lambda Expression Examples

```
1  public static void main(String[] args) {
2
3      Function<?, ?> function = (String value) -> System.out.printf("%s%n", value);
4
5      function = (value) -> System.out.printf("%s%n", value);
6
7      function = value -> System.out.printf("%s%n", value);
8
9      Runnable voidFunction = () -> Thread.currentThread().interrupt();
10
11     BiFunction<?, ?, ?> biFunction = (String value, String another) -> {
12         System.out.printf("%s%n", value);
13         System.out.printf("%s%n", another);
14         return String.format("%s %s", value, another);
15     };
16
17     biFunction = (value, another) -> String.format("%s %s", value, another);
18
19 }
```

Method References

- Java 8 provides a more compact syntax for referring to methods, called method references.
- Syntax leverages a double colon and is either
  a. Type::MethodName
  b. Object::MethodName
- There are four incarnations of this syntax
  1. **Static Method** [uses syntax (a)] - e.g. to reference the static method `atan` on `Math` class - `Math::atan`
  2. **Instance Method** of **arbitrary type** [uses syntax (a)] - e.g. to reference the instance method `isEmpty` on `String` class - `String::isEmpty`
  3. **Instance Method** of **existing object** [uses syntax (b)] - e.g. to reference the instance method `toUpperCase` on an object named '`professorName`' of type `String` - `professorName::toUpperCase`
  4. **Constructors** [uses syntax (a)] - technically the same as #2 but you use the 'new' keyword; e.g. `String::new`

Method Reference Examples

```
1  public static void main(String[] args) {
2
3      DoubleBinaryOperator biFunction = Math::pow;
4
5      List<String> values = new ArrayList<>();
6      values.sort(String::compareTo);
7
8      Function<?, ?> function = System.out::equals;
9
10     Runnable voidFunction = Thread.currentThread()::interrupt;
11
12     Function<String, Bmw> bmwConstructor = Bmw::new;
13
14 }
```

Method Reference - Overloading

- What's the problem with the following code?

```
1   Function<Integer, String> stringifier = Integer::toString;
```

- Overloading issues; there are multiple overloaded `toString` methods on `Integer`. Compiler is not 'smart' enough to figure out which to use. Can resolve (this particular one, not all) by explicitly referencing the `Object` type.

```
Function<Integer, String> stringifier = Object::toString;
```

Provided Functional Interfaces

- Java 7 and below already defined many of these (e.g. `Runnable`, `Callable`, `Iterable`, `AutoClosable`, `Comparable`, etc)
- Java 8 defines many new useful ones; including
  - Function<T, R> T -> R
  - BiFunction<T, U, R> (T, U) -> R
  - Predicate<T> T -> boolean
  - BiPredicate<L, R> (L, R) -> boolean
  - Consumer<T> T -> void
  - BiConsumer<T, U> (T, U) -> void
  - Supplier<T> () -> T
  - UnaryOperator<T> T -> T
  - BinaryOperator<T> (T, T) -> T
- Primitive variants available as well. Why?

Other Lambda Considerations

- Target Type
  - The compiler goes through a type checking process to determine the lambda expression is valid within its context. The expected type is called the target type.
- Type Inference
  - Because the target type can be deduced from the type checking process, the parameter types of a lambda can be inferred. This is similar to the type inference introduced in Java 7 for the "diamond" syntax (i.e., `List<String> values = new ArrayList<>();`)
- Exceptions
  - Lambda expressions (i.e. functional interfaces) can throw checked exceptions.
- Local Variables (clojures)
  - Just like anonymous classes prior to Java 8, local variables outside of the scope of the anonymous class (i.e. lambda) can be referred to provided they are `final` (or effectively final). Unlike the notion of clojure in other languages, in Java captured free variables cannot be modified (as they are `final`).
- Void Compatibility Rule
  - If a lambda has an expression as it's body and the expression returns a value it can still be used in the context of a 'void' functional interface.
    - e.g. : `Consumer<Integer> consumer = integer -> set.add(integer);`

Example Usage - Execute Around Pattern

- What's wrong with the following code?

```
1  public List<Student> getTopStudents() {
2      List<Student> top = new ArrayList<>(students.size() / 4);
3      for (Student student : students) {
4          double average = getAverage(student);
5          if (average >= TOP_GRADE_AVG) {
6              top.add(student);
7          }
8      }
9      return top;
10 }
11
12 public List<Student> getBottomStudents() {
13     List<Student> bottom = new ArrayList<>(students.size() / 4);
14     for (Student student : students) {
15         double average = getAverage(student);
16         if (average <= BOTTOM_GRADE_AVG) {
17             bottom.add(student);
18         }
19     }
20     return bottom;
```

Stream API

- The Stream API is a way to declaratively modify data.
  - The data can be contained within a Java Collection but it is not limited to this and can be other data like that within an array or from I/O.
- How is this different than interacting with a Collection?
  - A collection is a data structure in memory holding all its elements at once (an object has to be created before being placed into a Collection). Think of this as a CD holding songs (the CD holds only the songs added to it and all songs are present when the CD is created).
  - A stream is a conceptual data structure where elements are only computed on demand. Think of this as a streaming music service like Spotify (songs are available on demand).
- Streams have declarative features which are possible because of the functional aspects added within Java 8.
  - This provides new patterns of interacting with data. These new patterns can often be terser, more descriptive and also more efficient.

Stream Features

- Pipelining
  - Many stream functions return another Stream. This allows operations to be chained together. This pipelining can be exploited to perform operations lazily and even short circuit operations.
- Internal Iteration
  - Iteration is done within the Stream (is transparent to the programmer). This is in contrast to the Collection API. This internal iteration allows exploiting parallelism transparently as well as makes interaction by the developer less verbose.

Stream Operations

- Streams have two types of operations.
  - **Intermediate** - produce another Stream. This is how pipelining works.
  - **Terminal** - produce an end result (not a Stream). This not only starts but also completes the Stream (i.e. the Stream processing is lazy and doesn't start until the terminal operation- allowing for optimization of the intermediate operations).
- Note, streams can only be consumed once.
- A Stream interaction follows the following pattern:
  - The **data to perform operations** on. Every `Collection` now has a `stream()` and `parallelStream()` method to return a `Stream` into the `Collection` data.
  - One to many **intermediate operations** (the pipeline)
  - A single **terminal operation**

Stream Operations

Intermediate

Operation	Return Type	Argument
`filter`	Stream<T>	Predicate<T>
`map`	Stream<R>	Function<T, R>
`limit`	Stream<T>	
`sorted`	Stream<T>	Comparator<T>
`distinct`	Stream<T>	

Terminal

Operation	Purpose
`forEach`	Consumes each element applying a lambda to each. Returns `void`
`count`	Returns the number of elements in the stream, as a `long` type
`collect`	Reduces the stream via the `Collector` interface. Many default implementations exist for `Collector` including `toList`, `toSet`, `toMap`, `groupingBy`, et al. See `Collectors`

Null Safety - `Optional`

- The `null` value is 😱
  - Causes many errors - NullPointerException is by far the most common exception in Java[1]
  - Avoiding it causes code bloat
  - Meaningless and untyped - represents the wrong way to refer to the absence of a type in the Java type system. Moreover, it carries no type information (can be assigned to any type).
  - Exposes pointers - Java hides all pointers from developers but the null pointer
- `Optional` type added to help resolve this
  - Modeled after similar type in Guava and in the Scala language

[1] Java 8 In Action - Urma, Fusco & Mycroft; page 228

Null Unsafe or Code Bloat

- Code accessing null references
  - Terse and easy to read
  - BUT unsafe

```
1  public class PartyPlanning {
2
3      private final Party party;
4
5      public PartyPlanning(Party party) {
6          this.party = party;
7      }
8
9      public Double getPlannerCostUnsafe() {
10         return party.getPlanner().getCost().getAmount();
11     }
12
13 }
```

- Code accessing null references
  - Safe
  - BUT verbose and bloated

```
1  public class PartyPlanning {
2
3      private final Party party;
4
5      public PartyPlanning(Party party) {
6          this.party = party;
7      }
8
9      public Double getPlannerCost() {
10         if ((party == null) || (party.getPlanner() == null)
11             || (party.getPlanner().getCost() == null)) {
12             return null;
13         }
14         return party.getPlanner().getCost().getAmount();
15     }
16
17 }
```

Optional to the Rescue

- The `java.util.Optional` type provides a type safe solution
  - Includes convenience methods; like `get`, `isPresent`, `orElse`, et al
  - Mimics Stream like functionality; `map`, `flatMap`, `filter`

```
1  public class PartyPlanning {
2
3      private final Party party;
4
5      public PartyPlanning(Party party) {
6          this.party = party;
7      }
8
9      public Optional<Party> getOptParty() {
10         return Optional.ofNullable(party);
11     }
12
13     public Optional<Double> getPlannerCost() {
14         return getOptParty().flatMap(Party::getOptPlanner).flatMap(Planner::getOptCost).flatMap(Cost::getOptAmount);
15     }
16 }
```

Other Java 8 Features

- DateTime API - more fluent API (and it's immutable!)
  - Based on JSR-310 (inspired by Joda DateTime API)
  - Fluent API which is easy to understand
  - Review Tutorial here
- Others
  - See here

Additional Reading

- **Java 8 In Action** - Urma, Fusco & Mycroft
  - ISBN-13: 978-1449370770
  - ISBN-10: 1449370772