Java 8

Part One:

Date and Time Classes

Interfaces

Default Methods

Lambdas

DATE AND TIME

The java.time Package

- Existing time packages were inadequate.
  - Not thread-safe
  - Poor design- e.g., months start at 1, days start at 0
- The new java.time package addresses deficiencies.

The java.time Package

- Designed to be clear and intuitive
- Threadsafe
- Immutable
  - Most classes are immutable!
  - Create new instances with of, from, or with
  - No constructors, no setters

Immutable!

- Objects in the java.time package are immutable.
  - You cannot change them by invoking a method.
  - Methods will return a new object that you can then save in a variable.
- This is just like Strings:

```
String name = “Jessica”;

name.toUpperCase();
// returns JESSICA but the value is not used so  it is lost; name still stores Jessica

name = name.toUpperCase();
// name now stores JESSICA
```

Method Naming Conventions

Prefix	Method Type	Use
`of`	static factory	Creates an instance where the factory is primarily validating the input parameters, not converting them.
`from`	static factory	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
`parse`	static factory	Parses the input string to produce an instance of the target class.
`format`	instance	Uses the specified formatter to format the values in the temporal object to produce a string.
`get`	instance	Returns a part of the state of the target object.
`is`	instance	Queries the state of the target object.
`with`	instance	Returns a copy of the target object with one element changed; this is the immutable equivalent to a `set` method on a JavaBean.
`plus`	instance	Returns a copy of the target object with an amount of time added.
`minus`	instance	Returns a copy of the target object with an amount of time subtracted.
`to`	instance	Converts this object to another type.
`at`	instance	Combines this object with another.

https://docs.oracle.com/javase/tutorial/datetime/overview/naming.html

Local Dates

- Local dates and times have no associated time zone information
  - Example: April 7, 2016
- This is different from zoned dates and times:
  - Example: April 7, 2016, 18:00:00 PST
- Think about this like looking at the calendar on the wall.
- It is recommended to use local dates unless you really need to represent an absolute instance of time.

The LocalDate Class

- A LocalDate object has a year, month, and day of the month.
- Static methods `now` and `of` create LocalDate objects.

```
LocalDate today = LocalDate.now();

LocalDate stPattys = LocalDate.of(2016, 3, 17);

LocalDate holiday = LocalDate.of(2016, Month.JULY, 4);
// MONTH is an enum
```

"Changing" LocalDates

- LocalDate objects are **immutable**.
- You can invoke a method to create and return an entirely new object.
  - plusDays(int)
  - plusWeeks(int)
  - minusMonths(int)
  - minusYears(int)
  - plus(Duration) or minus(Duration)
  - plus(1, ChronoUnit.WEEKS)
- Other useful methods:
  - isBefore and isAfter
  - isLeapYear
  - until(LocalDate)
    - returns a Period

Duration

- A time-based amount of time
  - Time in nanoseconds, seconds, minutes, hours, or days (day = 24 hours)

```
Duration d = Duration.of(6, ChronoUnit.HOURS);
Duration meeting = Duration.ofHours(3);
Duration timeToAnswer = Duration.ofSeconds(30);
long hours = d.get(ChronoUnit.HOURS);
```

ChronoUnit

- Standard units of date and time

Enum Constants

Enum Constant	Description
CENTURIES	Unit that represents the concept of a century.
DAYS	Unit that represents the concept of a day.
DECADES	Unit that represents the concept of a decade.
ERAS	Unit that represents the concept of an era.
FOREVER	Artificial unit that represents the concept of forever.
HALF_DAYS	Unit that represents the concept of half a day, as used in AM/PM.
HOURS	Unit that represents the concept of an hour.
MICROS	Unit that represents the concept of a microsecond.
MILLENNIA	Unit that represents the concept of a millennium.
MILLIS	Unit that represents the concept of a millisecond.
MINUTES	Unit that represents the concept of a minute.
MONTHS	Unit that represents the concept of a month.
NANOS	Unit that represents the concept of a nanosecond, the smallest sup
SECONDS	Unit that represents the concept of a second.
WEEKS	Unit that represents the concept of a week.
YEARS	Unit that represents the concept of a year.

ChronoUnit

- The “between” method
  - ChronoUnit.VALUE.between(moment1, moment2)

```
long numDays = ChronoUnit.DAYS.between(date1, date2)
long seconds = ChronoUnit.SECONDS.between(instant1,
instant2)
```

ChronoUnit

- Not all ChronoUnits are supported by all types
  - Example: Instant objects do not support ChronoUnit.YEARS
  - Example: Period objects do not support ChronoUnit.SECONDS
- This will be a runtime error, not a compiler error.

Period

- A date-based amount of time values (years, months, days)
  - Time in days, months, or years

```
Period p = Period.of(years, months, days);
Period days = Period.ofDays(31);
Period months = Period.ofMonths(6);
long years = p.get(ChronoUnit.YEARS);
```

Instant

- An Instant object is a point on a time line.
- Instant objects can be used as timestamps.

```
Instant rightNow = Instant.now();
```

Instant

- Use a Duration to get the difference between two instants.

```
Instant start = Instant.now();
// code here
Instant end = Instant.now();
Duration timeElapsed =
    Duration.between(start, end);
long seconds = timeElapsed.getSeconds();
long minutes = timeElapsed.toMinutes();
```

"Changing" Instants and Durations

- Instant and Duration objects are **immutable**.
- You can invoke a method to create and return an entirely new object.
  - plusSeconds(int)
  - plusMinutes(int)
  - minusDays(int)

```
Instant timeA = Instant.now();
Instant timeB = timeA.plusSeconds(60);
```

LocalTime

- A Local object has an hour, minute second.
- Think about this like looking at the clock on the wall.
- Static methods `now` and `of` create LocalTime objects.

```
LocalTime rightNow = LocalTime.now();
LocalTime javaClass = LocalTime.of(18, 10);
          // with or without the second
```

"Changing" LocalTimes

- LocalTime objects are **immutable**.
- You can invoke a method to create and return an entirely new object.
  - plusHours(int)
  - plusMinutes(int)
  - minusSeconds(int)
  - plus(Duration) or minus(Duration)
- Other useful methods:
  - isBefore and isAfter

The LocalDateTime Class

- Immutable class that stores a date **and** time.
- Methods and usage are similar to LocalDate and LocalTime.

DayOfWeek and Month Enums

- DayOfWeek values
  - MONDAY (value 1) through SUNDAY (value 7)
- Month values
  - JANUARY (value 1) through DECEMBER (value 12)
- method: getDisplayName(TextStyle.FULL, Locale.getDefault());
  - // (or TextStyle.NARROW, TextStyle.SHORT)

Parsing from String to Date

- LocalDate can parse Strings in the format “YYYY-MM-DD”
  - This is DateTimeFormatter.ISO_LOCAL_DATE
  - LocalDate date = LocalDate.parse(“2016-04-07”);
- LocalTime can parse Strings in the format “hh:mm:ss”
  - Example: LocalTime time = LocalTime.parse(“18:10:00”); // with or without seconds
- You can also use a DateTimeFormatter to specify a different formatting.
  - DateTimeFormatter formatter = DateTimeFormatter.ofPattern(“LLL-DD-YYYY”);
  - LocalDate date = LocalDate.parse(“May-05-1999”, formatter);

Formatting for Output

- You can also use the DateTimeFormatter to format for output.
  - DateTimeFormatter formatter = DateTimeFormatter.ofPattern(“LLL-DD-YYYY”);
  - System.out.println(myDate.format(formatter));

All letters 'A' to 'Z' and 'a' to 'z' are reserved as pattern letters. The following pattern letters are defined:

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini; A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	week-of-month	number	3
a	am-pm-of-day	text	PM
h	clock-hour-of-am-pm (1-12)	number	12
K	hour-of-am-pm (0-11)	number	0
k	clock-hour-of-am-pm (1-24)	number	0
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30
s	second-of-minute	number	55
S	fraction-of-second	fraction	978
A	milli-of-day	number	1234
n	nano-of-second	number	987654321
N	nano-of-day	number	1234000000
V	time-zone ID	zone-id	America/Los_Angeles; Z; -08:30
z	time-zone name	zone-name	Pacific Standard Time; PST
O	localized zone-offset	offset-O	GMT+8; GMT+08:00; UTC-08:00;
X	zone-offset 'Z' for zero	offset-X	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	zone-offset	offset-x	+0000; -08; -0830; -08:30; -083015; -08:30:15;
Z	zone-offset	offset-Z	+0000; -0800; -08:00;
p	pad next	pad modifier	1
'	escape for text	delimiter	
''	single quote	literal	'
[	optional section start		
]	optional section end		
#	reserved for future use		
{	reserved for future use		
}	reserved for future use		

Practice

- Review the Time/Date example.

INTERFACES

Pre-Java 8

- Interfaces can only contain:
  - public static final constants
  - public abstract methods

Default Methods

- You can now provide a concrete implementation of an interface method- called a *default method*.
- Benefits
  - Flexibility: Interfaces can now be extended without breaking existing code.
  - Default methods can replace the need for adapter classes.
- To make a method default, include the `default` keyword in the method header and then implement the method.

Practice

- Review the Person, Employee, Intern, and Consultant classes and the HRProcessor interface.
  - Imagine this as part of a much larger system with many more classes!
- Add a new method (review) to the HRProcessor interface.
  - What happens to all my existing classes? Yikes!
  - Modify to make it a default method.

Default Methods

- This issue is essentially what motivated Java to add default methods.
- They wanted to add a whole lot of functionality to existing interfaces, but they didn’t want to break existing codes.
- Default methods to the rescue!

Using Default Methods

- If an interface has a default method, an implementing class has three options.

1. Don’t mention the method at all in the class.
   - The class then inherits and uses the default implementation.
2. Override the method.
   - The class defines a method to override (replace) the default implementation.
3. Re-declare the method as abstract.
   - The method becomes abstract in the class (making the class also abstract).
   - All child classes **must** now override the method.

Conflicts

- With these changes, conflicts are now possible.
- Before Java 8, multiple interfaces could have the same method name and it did not matter.
  - If a class implemented multiple methods with the same name, it just implemented one method in the class.
- Now, two interfaces could have the same method name and they could both be default methods that have an implementation.
  - Which to use?
- Example: Review the CollegeProcessor interface.

Resolving Conflicts

1. Superclasses win over interfaces.
   - If a superclass has a concrete method and an interface has a default method, the child class uses the superclass concrete method.
   - The default interface method is effectively ignored.
2. Interfaces conflicts must be explicitly resolved.
   - If either or both interfaces has a default method, the method **must** be overridden in the implementing class.
   - This is enforced at compile time.
   - You can choose to implement by accessing the default version: `InterfaceA.super.defaultMethodName();`
     - Note: your class must explicitly implement the interface to use this syntax (it cannot just inherit the “implements” from a parent).

Static Methods

- Interfaces can now also have static (implemented) methods.
- Pre-Java 8, we had interfaces and companion classes that provided utility methods.
  - Collection interface and Collections class with static methods (Collections.sort(…))
  - Path interface and Paths class with static methods (Paths.get(…))
- We could now include these static methods directly in the interface.
  - The existing Java classes won’t be changed, but moving forward we can create new classes in this way.

Default Methods… Why?

- Although of course you can now use default methods in your own programming, they were primarily introduced to support the other changes made in Java 8.
- Namely, streams… which require lambdas…

LAMBDA EXPRESSIONS

Terminology: Functional Interface

- A functional interface is an interface with a single abstract method.
- Example: Comparator<T>
  - One abstract method: public int compare(T obj1, T obj2)
- Example: ActionListener (from Swing)
  - One abstract method: public void actionPerformed(ActionEvent event)
- Example: EventHandler<T extends Event> (from JavaFX)
  - One abstract method: public void handle(T event)
  - This the class we send to buttons:
    button.setOnAction(myEventHandlerObject)

Pre-Lambdas

- We write a class that implements the functional interface.
- We create an object of that class.
- We invoke the method on that object.

- This is true object-oriented design!
- But… it's pretty cumbersome when all we *really* want is that one method!

Practice

- Review the Student class and how it implements Comparable
  - Orders by id
- Create some student objects, put them in the list, sort them.
- Write two comparators for the Student class
  - Sort by last name, first name
  - Sort by registration date
- Re-sort using the comparators.

- Review the single button GUI that changes the color of a rectangle.

Anonymous Classes

- Pre-Java 8, we could use an *anonymous class* to simplify things a little bit.
- With an anonymous class, you can declare and instantiate a class at the same time.
- You define the class as an expression.
  - The expression is part of another statement.

- This is better because we do not need to create a *named* class, but the syntax is still pretty cumbersome.

Anonymous Classes

- Example:

```
Comparator<MyClass> comparator = new Comparator<MyClass>() {

        public int compare(MyClass m1, MyClass m2) {
                return value;
         }
};
```

Practice

- Modify the sorting invocation to use an anonymous class.
- Modify the GUI to use an anonymous class.

Lambdas

- Lambdas allow aspects of *functional programming* to be included in Java (an *object-oriented* language).
- Lambdas allow us to create anonymous *methods*.

- For a functional interface, all we really care about is that one method- a *function*- rather than an object.
- We want to pass a *function* around, rather than an object.

- Lambdas allow you to specify code that will be executed later.

Lambda Syntax

(Argument List) -> Body

Parameters -> Expression

- Argument List/Parameters
  - ( ) are optional if there is only one argument
  - Types are optional if they can be inferred
- Body
  - Can be a single expression
    - The expression is evaluated **and** returned (the “return” keyword is optional)
  - Can be a statement block
    - The body is must be surrounded with { }
    - Return statement must be explicit

Simple Lambda Examples

```
(double a, double b) -> a / b;

( ) -> { System.out.println(“We have a problem.”); }

String s1 -> {
        String s2 = s1.toUpperCase();
        System.out.println(s2);
        return s2;
        }
```

Lambdas and Functional Interfaces

- You can use a lambda any place an object of a functional interface is expected.
  - Example: a method takes a Comparator? You can send a lambda.
  - Example: a method takes an ActionListener? You can send a lambda.
  - Example: a method takes an EventHandler? You can send a lambda.
- Send in a lambda to replace the one abstract method of the functional interface.
  - Example:
    - the Comparator<MyClass> method is compare(MyClass m1, MyClass m2)
    - the lambda is (MyClass m1, MyClass m2) -> { code that goes inside compare }

Using Lambdas to Replace Functional Interfaces

- Think about the *method* you care about inside the functional interface.
  - The one abstract method
- That's what you're replacing with the lambda.
  - What are the parameters to that method?
    - These are the lambda parameters
  - What code would normally go inside your method?
    - This the lambda body expression

Practice

- Replace the comparator code with lambdas.
- Replace the GUI listener code with lambdas.

Variable Scope

- A lambda expression can *capture* the value of a variable in the enclosing scope as long as it is *effectively final*.
- What does this mean?
  - Lambdas can safely use and update instance data variables.
  - For local variables or formal parameters, the variable must be *effectively final*.
- Effectively final means you do not have to declare it as final, but you cannot change its value.
- Using `this` inside of a lambda refers to the `this` parameter of the method that creates the lambda.

- Example: add a count to be updated.

Practice

- Review the GUI conversion program.
  - Implement a method to handle the button click.
    - Link to the method using a lambda.
  - Write a factory class to create Converter objects
    - First, use the old approach of creating named classes that implement the interface.
    - Then, convert to use lambdas.
    - Then think if you can streamline even more!

Lambdas are just the beginning…

- In addition to allowing lambdas, Java 8 added an entire new set of functionality for manipulating collections called the Stream API.
- This functionality uses lambdas to radically improve how we work with collections.
- But let’s take one step at a time… so this week, just focus on the lambda syntax. Next week, we’ll see it in action with Streams to really see how useful it can be!