Intermediate Java

Geoff Matrangola
geoff@matrangola.com

Overview

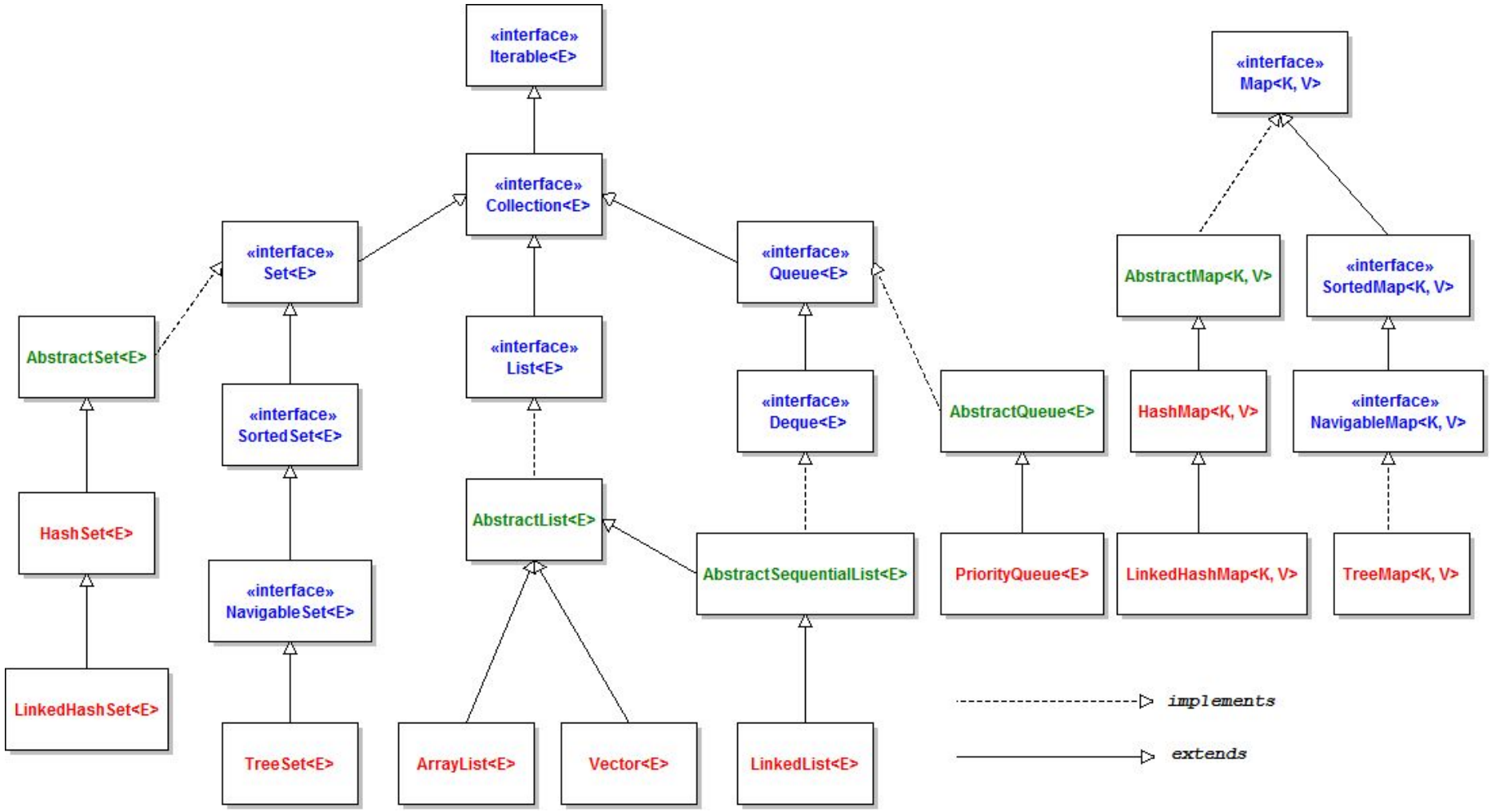
- Instructor: Geoff Matrangola geoff@matrangola.com
- Company: DevelopIntelligence
- Java Knowledge Expectations
  - Edit, compile, run Java Code
  - String manipulation, arrays, basic types
  - Control structure (if, while, etc)
  - Basic OOP
- Class Objectives
  - Closer Look at Collections
  - Date Time API
  - Lambda Expressions
  - Generics
  - Concurrency

Java Collections

Collections

- Base Class *Collection* (sub of *Iterable*) https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
- Primary Interfaces
  - Set - Not Ordered, no duplicates
  - List - Ordered, Duplicates entries permitted, indexed by position
  - Map - Not Ordered, custom index
- Common Specialized interfaces and implementations
  - HashSet
  - ArrayList
  - LinkedList
  - HashMap
  - TreeMap

Collection Classes UML



https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram

Primary Collections

- Set
  - HashSet
  - TreeSet
- List
  - LinkedList
  - ArrayList
- Map
  - HashMap
  - TreeMap
- Queues
  - Dequeue
  - PriorityQueue
- General
  - Collection Interface
  - Iterator and Iterable
  - Enumerators

Java Lambdas

Lamda Objectives

- Outline the purpose of lambda expressions
- Read and write lambda expressions in Java

Why Lambdas?

- Programs benefit from flexible behavior; e.g:
  - Sort these using this ordering policy
  - Remove elements unless they match this criterion
- Historically, “variable behavior” has different approaches:
  - Pointer to function (e.g. C, C++)
  - Interpretable source code (e.g. SQL)
  - Object implementing an interface (e.g. Java)
  - Anonymous inner classes (e.g. Java)
  - “Code as data” or “First Class” functions (e.g. Lisp, functional languages, Java 8 Lambdas)

Copyright DevelopIntelligence LLC

What Are Lambdas?

- In essence a Java lambda behaves like a pointer to a function
- Java is object oriented, and statically type-safe, so lambdas are essentially code that compiles to a pointers to a object that implements a single-method interface
  - With a lot less textual “clutter”

Copyright DevelopIntelligence LLC

From Classes to Lambdas

```
public class Filter {
  public static <E> void filterList(List<E> data,
                                    Test<E> t) {
    Iterator<E> iterator = data.iterator();
    while (iterator.hasNext()) {
      if (!t.test(iterator.next()))
        iterator.remove();
      }
  }
```

Copyright DevelopIntelligence LLC

From Classes to Lambdas

```
public static void main(String [] args) {
    List<String> ls = new LinkedList<>();
    ls.addAll(Arrays.asList(
      "Alice", "Bob", "Maverick", "Trent"));
    System.out.println("Before: " + ls);
    filterList(ls, new LongerThan5());
    System.out.println("After: " + ls);
  }
}
```

Copyright DevelopIntelligence LLC

From Classes to Lambdas

```
public interface Test<E> {
    boolean test(E e);
}
```

Note: only one method is declared in this interface

```
public class LongerThan5 implements Test<String> {
    @Override
    public boolean test(String s) {
        return s.length() > 5;
    }
}
```

Copyright DevelopIntelligence LLC

From Classes to Lambdas

- Previously:

```
public class LongerThan5 implements Test<String> {
    @Override
    public boolean test(String s) { // ->
        return s.length() > 5;
    }
}
```

- Becomes:

```
(s) -> s.length() < 5
```

Copyright DevelopIntelligence LLC

Lambda General Format

- Lambdas provide behavior that implements a method in an interface
  - Commonly a generic interface

- Syntax defines:
  - Argument list
  - `->`
  - Behavior

Copyright DevelopIntelligence LLC

Type Inference

- Java compiler attempts to decide the type of the lambda based on the context
  - Lambda defined in method argument must satisfy the requirements of the method
    - Overloaded methods can cause ambiguity
  - Lambda defined in initialization of variable must satisfy the type of the variable
  - Lambdas frequently are used to implement generic types, in which case the generic type variables are inferred from the context too
- Inference isn't always possible

Copyright DevelopIntelligence LLC

Lambda Argument Syntax

- Argument list is generally enclosed in parentheses
  - Types do not generally need to be specified

    ```
    (s,t,u) -> s + t / u
    ```

  - Types may be specified for the ***entire*** argument list to resolve type inference ambiguity

    ```
    (long s, long t, int u) -> s + t / u
    ```

  - Zero arguments use empty paretheses

    ```
    () -> (int)(Math.random() * 1000)
    ```

  - Single argument situations allow the parentheses to be dropped

    ```
    s -> System.out.println("Value is " + s)
    ```

Copyright DevelopIntelligence LLC

Expression Lambda Syntax

- Simple lambdas may be expressed using a single expression to the right of ->
  - These are called “Expression Lambdas”
  - Note that no semicolon is used to terminate the expression

```
(s)  -> s * 2
```

Copyright DevelopIntelligence LLC

Complex Lambda Behavior Syntax

- For more complex lambda behavior, a block may be used

```
(s,t) -> {
  int rv = 0;
  for (int i = s; i < t; i++) {
    rv += i;
  }
  return rv;
}
```

- If a traditional-form method would require a return statement, then the block form lambda requires a return statement too

Copyright DevelopIntelligence LLC

Lambdas And Closure

- Lambda expressions can refer to variables in enclosing scopes provided their lifetimes are suitable
- Rules are as variable access in inner classes
  - Fields of enclosing class or object are accessible
  - Method locals must be “effectively `final`”
- The design merits of this technique are highly debatable
  - Sometimes, very good, other times less desirable
  - As a general guide, try to avoid “side effects”

Copyright DevelopIntelligence LLC

Functional Interfaces

- Lambdas can only be used to implement interfaces that have a single abstract method
- Such interfaces are called “Functional Interfaces”
- The annotation `@FunctionalInterface` tells the compiler to verify that this interface defines exactly one abstract method
  - `@FunctionalInterface` is not required to allow use in a lambda; the annotation only serves to warn if we accidentally created more than one abstract method

Copyright DevelopIntelligence LLC

Java 8 API Functional Interfaces

- Package `java.util.function` defines many functional interfaces

Interface	Method
`Predicate<T>`	`boolean test(T t)`
`Supplier<T>`	`T get()`
`Consumer<T>`	`void accept(T t)`
`Function<T,R>`	`R apply(T t)`
`BiFunction<T,U,R>`	`R apply(T t, U u)`

This is the “right” interface for the “Test” defined in the earlier example

Copyright DevelopIntelligence LLC

Functional Interfaces And Primitives

- Most functional interfaces are generic, e.g. `BiFunction<T,U,R>` defines `R apply(T t, U u)`
- But, generics are incompatible with primitive data types
- So, several functional interfaces are defined for primitive types, `int`, `long`, and `double`
  - But `float` is generally ignored; use `double` instead, or define your own interface

Copyright DevelopIntelligence LLC

Method References

- Method references allow pre-existing methods to be conveniently used where a lambda expression would be applicable

Copyright DevelopIntelligence LLC

Method Reference Example

```
public static <E> E executeBinaryOp(
    E e1, E e2, BinaryOperator<E> op)
{
  return op.apply(e1, e2);
}
```

```
executeBinaryOp("Jim", " Jones", (s,t)->s.concat(t))

executeBinaryOp("Jim", " Jones", String::concat)
```

Copyright DevelopIntelligence LLC

Method Reference Invocation

- Methods for references can be instance ***or*** static
- The interface `BinaryOperator<T>` requires two arguments (and returns a single value)
- Suppose the arguments are `s` and `t`
- The method reference `String::concat` will cause invocation as `s.concat(t)`
- However, a static method `String joinStrings(String a, String b)` will be invoked as `joinStrings(s, t)`

Copyright DevelopIntelligence LLC

Method References Afterthought

- The use of method references effectively allows any arbitrary method to be used to implement a functional interface, without the original method or its defining class knowing anything about that interface
  - This is an aspect of “duck typing” in Java, although it happens at compile time and only relates to functional interface behavior, not to objects as a whole

Copyright DevelopIntelligence LLC

Lab Exercise

- Create a `Customer` class representing a customer of our retail outlet. Give the customer a name, a credit limit, a credit balance, and optionally a `Set<String>` representing the items this customer has purchased from us
- Give the customer a `toString` method to allow easy textual representation
- In a `main` method create a `List<Customer>` with a few sample customers in it
- Print the list out

Copyright DevelopIntelligence LLC

Lab Exercise

- Write a generic filter method, similar to the one in the example, that takes a list and “filter behavior” and creates a new list that contains only the items that pass the test of the filter
- Think about what interface the “filter behavior” should implement? Use the standard Java 8 interfaces

Copyright DevelopIntelligence LLC

Lab Exercise

- Arrange for the main method to filter the list according to several criteria, printing the list each time
- Suggested criteria are:
  - Customers who have a credit balance greater than 1000
  - Customers who have a credit balance greater than their credit limit
  - Customers whose names begin with a particular letter
  - Customers who buy a particular item
- Optional: implement one of these criteria without using lambdas

Copyright DevelopIntelligence LLC

Streams

Streams - Overview

- Introduced in Java 8
- Non destructive of source data.
- Works well with Lambdas and method references

Creating Streams

- Empty Stream
  - `Stream<String> streamEmpty = Stream.empty();`
- Stream from Collection
  - `Collection<String> collection = Arrays.asList("Pippa", "Natty", "Oscar");`
  - `Stream<String> streamOfCollection = collection.stream();`
- Stream from VarArgs (array)
  - `Stream<String> streamOfArray = Stream.of("Pippa", "Natty", "Oscar");`
- Stream from Array
  - `String[] dogs = new String[]{"Pippa", "Natty", "Oscar"};`
  - `Stream<String> streamOfArrayFull = Arrays.stream(dogs);`
  - `Stream<String> streamOfArrayPart = Arrays.stream(dogs, 1, 3);`
- Stream from Builder
  - `Stream<String> streamBuilder =`
  - `Stream.<String>builder().add("Pippa").add("Natty").add("oscar").build();`

Stream Generators

- Generate Stream with the generate() method and a lambda and limit with the limit() method
  - ```
    Stream<String> streamGenerated =
    ```
  - ```
        Stream.generate(() -> "woof").limit(5);
    ```
- Use Iterate method
  - Like counters, etc. Can be used for indexes
  - ```
    Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
    ```
    ```

Primitives in Streams

- Built-in lowercase types (int, float, long, double) are not always treated as first-class objects in Java
- Stream style *for*-loop replacement is possible by using special interfaces *IntStream, LongStrea, and DoubleStream*
  - IntStream.*range*(1, 500)
  - LongStream.*rangeClosed*(1, 20);

Stream Terminal Operations

- Perform some operation on the entire Stream and produce final output
  - forEach()
  - toArray()
  - reduce()
  - collect()
  - min()
  - max()
  - count()
  - anyMatch()
  - allMatch()
  - noneMatch()
  - findFirst()
  - findAny()
- Predicate - used in many of the operations
  - `boolean all = dogStream.allMatch(p -> p.length() == 5);`

Terminal Operations, terminal

Terminal operations can only be used one per stream.

```
LongStream longStream = LongStream.rangeClosed(1, 20);

longStream.forEach(
      p -> System.out.println("i=" + p)
);

long sum = longStream.reduce(0, (a,b) -> a+b);
System.out.println("sum = " + sum);
```

```
i=19
i=20
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or
closed
        at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
        at java.util.stream.LongPipeline.reduce(LongPipeline.java:438)
        at com.developintelligence.demo.Main.main(Main.java:16)
```

Stream Intermediate Operations

- Perform some operation on the stream and pipeline to next operation
- Returns a new stream
- Executed when the terminal operation is run (lazy)
- Examples
  - filter
  - map
  - flatmap
  - peak
  - distinct
  - sorted
  - limit

Stream Processing Performance

- Place the operations that will reduce the size of the set in the beginning of the chain.
- *filter(), distinct(), and skip()*
- For example, filter before you map.

Parallel Stream

- Uses multiple threads to balance core use
- Executes operations in parallel
- Create with
  - List.parallelStream()
  - Stream.parallel()
- Rules
  - stateless operation at the element level
  - non-interfering operation- data source cannot be affected
  - An associative operation is an operation in which the result is not affected by the order of operand
- forEach vs. forEachOrdered

More Generics

Generics Classes Overview

- List<Course> courses = new ArrayList<Course>
- Diamond Operator: List<Course> courses = new ArrayList<>
- Naming Convention
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types
- Nesting:

  Map<String, List<Course>> coursesByInstructor = new HashMap<>

-

Generic Methods Overview

```
public static <T> void printList(String title, Iterable<T> thingy) {
   System.out.println(title);
   for (T o : thingy) {
      System.out.println(o.toString());
   }
}
```

Bounded Type

```
public static <T, V extends Iterable<T>> void printList2(String title, V items) {
   System.out.println(title);
   for (T item : items) {
      System.out.println(item);
   }
}
```

Wildcard Generics

```
public static int findMaxId(List<? extends IdBean> items) {
   return items.stream().mapToInt(IdBean::getId).max().getAsInt();
}
```

Time API

Objectives

- Work with absolute points in time
- Work with periods of time
- Compare points in time
- Modify points in time based on time increments
- Modify points in time based on calendar aspects
- Convert dates and times to and from text
- Work with local dates and times

Copyright DevelopIntelligence LLC

Representing Time

- The package `java.time` provides a comprehensive set of tools for handling time and date
  - Dates, potentially in many calendar systems
  - Dates and times, including timezones
  - Points in time without reference to any calendar system
  - Durations of time
  - Means for converting points in time between representations
  - Means for moving dates and times around by durations
  - Parsing and formatting times and dates

Copyright DevelopIntelligence LLC

Absolute Points In Time

- `ZonedDateTime` and Instant represent an unambiguous point in time
  - Instant is just the moment in time
  - `ZonedDateTime` includes a notion about how this will be interpreted / presented, i.e. the time zone
- Both have nanosecond nominal accuracy

Copyright DevelopIntelligence LLC

Creating ZonedDateTime

- `ZonedDateTime` can be constructed in several ways using static factory-type methods

now	Current instant, in the system default time zone
of(y, m, dom, h, m, s, ns, tz)	Instant specified by the arguments
of	Instant & timezone specified by arguments
ofInstant(inst, tz)	Instant & timezone specified by arguments
ofLocal	A LocalDate & Time, with a timezone
parse	Instant & timezone specified in text

Copyright DevelopIntelligence LLC

Creating An Instant

- `Instant` can be created using static factory-type methods

now	Current instant
ofEpochSecond	Instant n seconds (& nanosecs) after Jan 1 1970 epoch
ofEpochMilli	Instant n milliseconds after Jan 1 1970 epoch
parse	Instant & timezone specified in text

Copyright DevelopIntelligence LLC

Representing Relative Time

- `Duration` and `Period` represent time differences:
  - 3 hours 15 minutes
  - 1 year and a day
  - 197.28 seconds
- `Duration` is “machine” based
  - 1 day is 24 hours
- `Period` is “calendar” based
  - 1 day might vary from 24 hours, depending on daylight saving etc.

Copyright DevelopIntelligence LLC

Limitations Of Relative Time

- Relative time represented by `Period` can be specified in terms of years, weeks, hours, etc.
- However, a period of 60 days cannot readily be converted to 2 months
  - Because a period, without reference to a starting point, doesn't know what calendar month(s) it covers
- Similarly, 365 days cannot be readily converted into 1 year as it might be a leap year
- `Period` has a `normalized` method, but this will only normalize elements up to days.

Copyright DevelopIntelligence LLC

Time Units

- Several features of the Date Time API allow / require the use of time units to clarify a request
- `ChronoUnit` is an enumerated type that defines these, e.g.:
  - `ChronoUnit.CENTURIES`
  - `ChronoUnit.DAYS`
  - `ChronoUnit.HOURS`
  - Etc.
- These can be used, for example, in creating a `Duration`:
  - `Duration d = Duration.of(3, ChronoUnit.HOURS);`

Copyright DevelopIntelligence LLC

Comparing Times

- Both `Instant` and `ZonedDateTime` (and others not yet introduced) implement `Temporal`
- The time difference between two `Temporal` values can be determined using
  `Duration.between(t1, t2)`
- Time "elements" between two points can be computed using `ChronoUnit`, e.g.:

```
long h = ChronoUnit.HOURS.between(t1, t2);
```

- Several other `between` methods exist in other classes, with more specific applications

Copyright DevelopIntelligence LLC

Modifying Times

- `ZonedDateTime` can create a derived date using `plusXxx` and `minusXxx` methods

```
ZonedDateTime today = ZonedDateTime.now();
ZonedDateTime tomorrow = today.plusDays(1);
ZonedDateTime nextWeek = today.plusWeeks(1);
ZonedDateTime lastYear = today.minusYears(1);
Duration d4 = Duration.ofHours(77);
ZonedDateTime later = today.plus(d4);
```

- Note: ***most date / time API elements are immutable***, so modification behaviors create new objects; don't forget to store them!

Copyright DevelopIntelligence LLC

More Time Modification

- ZonedDateTime also allows adjusting single elements of the represented time; e.g.:

```
ZonedDateTime here =
  ZonedDateTime.of(2015, 3, 8, 1, 55, 0, 0,
    ZoneId.of("America/Denver"));

ZonedDateTime ny =
  before.withZoneSameInstant(
    ZoneId.of("America/New_York"));
```

- Refers to the same moment, in a different time zone

Copyright DevelopIntelligence LLC

More Time Modification

- The withXxx methods also allow changing the time

```
ZonedDateTime fiveAm = today.withHour(5);
```

Copyright DevelopIntelligence
LLC

Advanced Date Modification

- Humans often make date modification in less purely mathematical terms, e.g. “a week on Friday”, or “on the third Monday of the month”
- Changes such as these are supported by the `TemporalAdjuster` interface, along with utility implementations available from the `TemporalAdjusters` class

Copyright DevelopIntelligence LLC

Advanced Date Modification

```
ZonedDateTime janOne =
  ZonedDateTime.of(2015, 1, 1, 0, 0, 0, 0,
    ZoneId.of("America/Denver"));

ZonedDateTime firstFriday = janOne.with(
    TemporalAdjusters.dayOfWeekInMonth(1,
                                       DayOfWeek.FRIDAY));

ZonedDateTime nextMonth = janOne.with(
    TemporalAdjusters.firstDayOfNextMonth());
```

Copyright DevelopIntelligence LLC

Formatting And Parsing

- The `DateTimeFormatter` class is a configurable tool for formatting date/time objects as text, and parsing text into date/time objects
- Many ISO standard formats are supported directly as constants, but arbitrary formats can be created from template strings

Copyright DevelopIntelligence LLC

Using a DateTimeFormatter

- `DateTimeFormatter` has several static methods for creating formatter objects suitable for different data / time object types

```
DateTimeFormatter dtf =
  DateTimeFormatter.ofLocalizedDateTime(
    FormatStyle.MEDIUM);
ZonedDateTime now = ZonedDateTime.now();
System.out.println("> " + dtf.format(now));
```

- Produces something like:

```
May 4, 2015 1:17:26 PM
```

Copyright DevelopIntelligence LLC

Using a DateTimeFormatter

- Parse operations:

Parsing creates “internal” objects for data/time.

```
TemporalAccessor ta =
  dtf.parse("Jul 20, 1969 8:18:00 PM");

ZonedDateTime landing =
  LocalDateTime.from(ta).atZone(ZoneId.of("UTC"));

System.out.println("> " + landing);
```

If desired, convert to a specific date/time type

Copyright DevelopIntelligence LLC

Arbitrary Date / Time Formats

- If the ISO standard date / time formats do not suit, `DateTimeFormatter` allows specification using template text (much like `printf`)

```
DateTimeFormatter dtf2 =
   DateTimeFormatter.ofPattern(
     "HH:mm:ss MMMM d, yyyy");
```

Format characters are defined in the API documentation

```
System.out.println(dtf2.format(now));
```

- Produces something like:

```
14:09:28 March 20, 2013
```

- This formatter can parse too

Copyright DevelopIntelligence LLC

Arbitrary Date / Time Formats

- The format specification characters can typically be repeated.

  Repetitions are interpreted as changing the width and/or style of the representation

  - “yy” □ 15
  - “yyyy” □ 2015
  - “e” □ 2
  - “ee” □ 02
  - “eee” □ Mon
  - “eeee” □ Monday

The API documentation indicates the format variations that are possible

Copyright DevelopIntelligence LLC

Local Points In Time

- The date / time API can also describe dates and times in the local time zone
  - These might be simpler to work with, but might be inconvenient if the program is later modified to a global audience
- Three classes:
  - `LocalDate`
  - `LocalTime`
  - `LocalDateTime`

Copyright DevelopIntelligence LLC

Local Points In Time

- Local time/dates support most of the conversions and adjustments that are applicable to `ZonedDateTime`
- They can be converted to `ZonedDateTime` given a time zone:

```
zdt = ldt.atZone(zoneId);
```

- They can be extracted from `ZonedDateTime` or `Instant`

```
ldt = LocalDateTime.ofInstant(inst, zoneId);
ldt1 = zdt.toLocalDateTime();
```

Copyright DevelopIntelligence LLC

Limitations Of Local Date / Time

- Local date and time objects are missing some time information
  - They have no timezone
  - A `LocalDate` has no time
  - A `LocalTime` has no date
- Some processes, including data extraction, and formatting, might try to access these missing items
  - This will throw an exception
- Determine if an item is available using the `isSupported` methods

Copyright DevelopIntelligence LLC

Using Legacy Date / Time Objects

- Several methods exist facilitating using the new date / time API with code already using older APIs

Conversion Class & Method Name
`Calendar.toInstant`
`GregorianCalendar.toZonedDateTime`
`GregorianCalendar.from(ZonedDateTime)`
`Date.from(Instant)`
`Date.toInstant`
`TimeZone.toZoneId`

Copyright DevelopIntelligence LLC