Object Oriented Programming

(CSC 302)

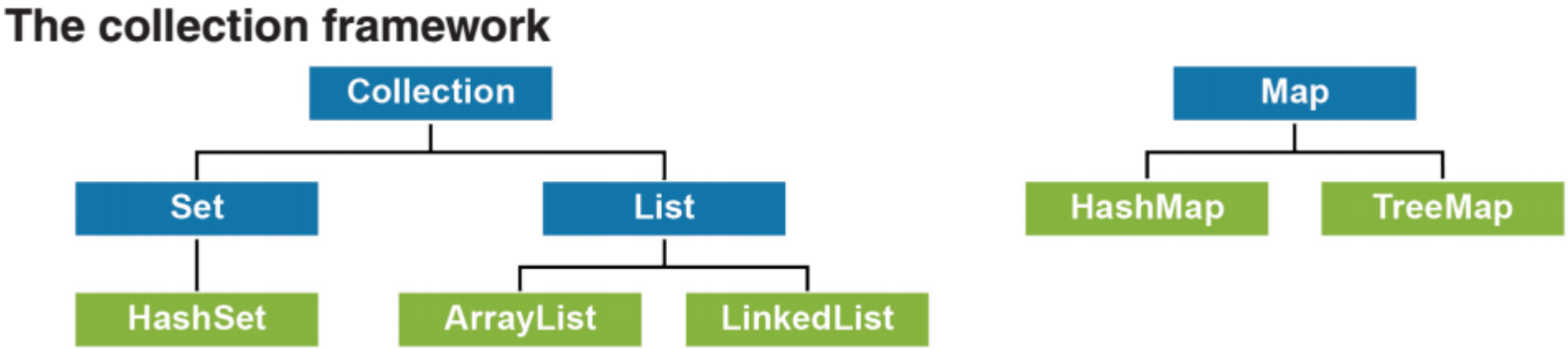
Lecture Note 6

Collections and Generic

4th December, 2019

Introduction to Collections and Generics

- The Java collection framework is interface based, which means that each class in the collection framework implements one of the interfaces defined by the framework.
- The collection framework consists of two class hierarchies: **Collection** and **Map**. A collection stores individual objects as elements. A map stores key-value pairs where you can use a key to retrieve a value (element).



- **Collection interfaces**
  - **Collection** - Defines the basic methods available for all collections.
  - **Set** - Defines a collection that does not allow duplicate elements.
  - **List** - Defines a collection that maintains the sequence of elements in the list. It accesses elements by their integer index and typically allows duplicate elements.
  - **Map** - Defines a map. A map is similar to a collection. However, it holds one or more key-value pairs instead of storing only values (elements). Each key-value pair consists of a key that uniquely identifies the value, and a value that stores the data.
- **Common collection classes**
  - **ArrayList** - More efficient than a linked list for accessing individual elements randomly. However, less efficient than a linked list when inserting elements into the middle of the list.
  - **LinkedList** - Less efficient than an array list for accessing elements randomly. However, more efficient than an array list when inserting items into the middle of the list.
  - **HashSet** - Stores a set of unique elements. In other words, it does not allow duplicates elements.
  - **HashMap** - Stores key-value pairs where each key must be unique. In other words, it does not allow duplicate keys, but it does allow duplicate values.
  - **TreeMap** - Stores key-value pairs in a hierarchical data structure known as a tree. In addition, it automatically sequences elements by key.

Introduction to Generics

- Generics refers to a feature that lets you create typed collections. A typed collection is a collection that can hold only objects of a certain type. This feature was introduced with Java 5.
- To declare a variable that refers to a typed collection, you list the type in angle brackets (<>) following the name of the collection class.
- When you use a constructor for a typed collection, you can specify the type variable in angle brackets following the constructor name. The type variable can't be a primitive type such as *int* or *double*, but it can be a wrapper class such as **Integer** or **Double**. It can also be a user-defined class such as **Student**.
- Beginning with Java 7, you can omit the type from within the brackets that follow the constructor if the compiler can infer the type from the context. This empty set of brackets is known as the **diamond operator**.
- If you do not specify a type for a collection, the collection can hold any type of object. However, the Java compiler will issue warning messages whenever you access the collection to warn you that type checking can't be performed for the collection.

- How to use the ArrayList class
- java.util.ArrayList
  - The **ArrayList** class uses an array internally to store the elements in the list.
  - The capacity of an array list automatically increases whenever necessary.
  - When you create an array list, you can use the default starting capacity of 10 elements, or you can specify the starting capacity.
  - If you know the number of elements that your list needs to be able to store, you can improve the performance of the ArrayList class by specifying a starting capacity that's just over that number of elements.
- Common constructors of the ArrayList class
  - **ArrayList<E>()** - Creates an empty array list of the specified type with the default capacity of 10 elements.
  - **ArrayList<E>(**intCapacity**)** - Creates an empty array list of the specified type with the specified capacity.
- Code that creates an array list of String objects
  - With the default starting capacity of 10 elements

```
ArrayList<String> codes = new ArrayList<>();
```

  - With a specified starting capacity of 200 elements

```
ArrayList<String> codes = new ArrayList<>(200);
```

- How to add and get elements
  - Common methods of the ArrayList class
  - **add(**object**)** - Adds the specified object to the end of the list.
  - **add(**index, object**)** - Adds the specified object at the specified index position.
  - **get(**index**)** - Returns the object at the specified index position.
  - **size()** - Returns the number of elements in the list.
- Code that adds three elements to an array list

```
codes.add("jsp"); codes.add("mysql"); codes.add(0, "java");
```

- Code that gets the last element

```
int lastIndex = codes.size() - 1;
String lastCode = codes.get(lastIndex);
```

- Code that gets and displays each element of an array list

```
for (String code : codes) {
    System.out.println(code);
}
```

- How to replace, remove, and search for elements
- More methods of the ArrayList class
  - **clear()** - Removes all elements from the list.
  - **contains(**object**)** - Returns true if the specified object is in the list.
  - **indexOf(**object**)** - Returns the index position of the specified object.
  - **isEmpty()** - Returns true if the list is empty.
  - **remove(**index**)** - Removes the object at the specified index position and returns that object.
  - **remove(**object**)** - Removes the specified object and returns a boolean value that indicates whether the operation was successful.
  - **set(**index, object**)** - Sets the element at the specified index to the specified object.
  - **toArray()** - Returns an array containing the elements of the list.
- Code that replaces an element

```
codes.set(2, "android");
```

- Code that removes an element

```
String code = codes.remove(1);
```

- Code that searches for an element

```
String searchCode = "android";
if (codes.contains(searchCode)) {
        System.out.println("This list contains: '" + searchCode + "'");
}
```

- How to store primitive types in an array list
- ArrayList does not store primitive values by default, however, you can use the wrapper classes to store primitive values.
- All primitive types have corresponding wrapper classes. For example, the Integer class is the wrapper class for the int type, the Double class is the wrapper class for the double type, and so on.
- To store a primitive type in a collection, you can specify its wrapper class as the type for the collection. Then, the compiler automatically converts the primitive value to its wrapper type when adding values to the collection. This feature is known as **autoboxing**.
- To get a primitive type from a collection, you don't need to do anything because the compiler automatically gets the primitive value from the wrapper class for you.
- Code that stores primitive types in an array list

```
ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(1); numbers.add(2); numbers.add(3);
```

- Code that gets a primitive type from an array list

```
int firstNumber = numbers.get(0);
```

Differences between Arrays and Collections

- An array is a Java language feature. Collections are classes in the Java API.
- Collection classes have methods that perform operations that arrays do not provide.
- Arrays are fixed in size. Collections are variable in size.
- Arrays can store primitive types. Collections cannot.
- Indexes are almost always required to process arrays. Collections are usually processed without using indexes.

Student Management System

How to work with dates and times APIs

- Before Java 8
- If you need your application to work with older versions of Java, you can use the Date, Calendar, and **GregorianCalendar** classes of the **java.util** package to work with dates and times. These classes are available to all modern versions of Java. To format these dates and times, you can use the **DateFormat** or **SimpleDateFormat** classes of the **java.text** package. These classes are also available to all modern versions of Java.
- However, these classes have some design flaws. To start, they are not thread-safe. In addition, most programmers consider the design of these classes to be unintuitive. For example, in the Date class, years start at 1900, months start at 1, and days start at 0. Finally, these classes don't make it easy to localize your application for parts of the world that don't use the Gregorian calendar. For example, when using these classes, it is not easy to write code that supports the Lunar calendar.
- As a result, Java 8 introduces a new date/time API that fixes some of the problems with the old date/time API that was used prior to Java 8.
- This is the version of date/time API we are going to use in this lesson.

- How to create date and time objects
  - The java.time package
  - **LocalDate** - A class for working with dates but not times.
  - **LocalTime** - A class for working with times but not dates.
  - **LocalDateTime** - A class for working with dates and times.
- Enumeration
  - **Month** - An enumeration that contains the months in the year (JANUARY through DECEMBER).
  - **DayOfWeek** - An enumeration that contains the days of the week (MONDAY through SUNDAY).
- Static methods of the LocalDate, LocalTime, and LocalDateTime classes
  - **now()** - Returns an appropriate object for the current local date, time, or date/time.
  - **of**(parameters) - Return an appropriate object for the specified date, time, or date/time parameters. For a date, you can specify the year, month, and day parameters. For a time, you can specify the hour, minute, second, and nanosecond parameters. However, the second and nanosecond parameters are optional and default to 0.
  - **parse**(string) - Returns an appropriate object for the specified string. For a date, you specify a string in the form of YYYY-MM-DD. For a time, you specify a string in the form of HH:MM:SS.NNNNNNNNN. However, the seconds and nanoseconds are optional and default to 0. For a date/time, the date and time are separated by a "T".

- Code that creates date/time objects
  - LocalDate currentDate = LocalDate.now();
  - LocalTime currentTime = LocalTime.now();
  - LocalDateTime currentDateTime = LocalDateTime.now();
  - LocalDate halloween1 = LocalDate.of(2015, Month.OCTOBER, 31);
  - LocalDate halloween2 = LocalDate.of(2015, 10, 31);
  - LocalTime startTime1 = LocalTime.of(14, 32);
  - LocalTime startTime2 = LocalTime.of(14, 32, 45);
  - LocalTime startTime3 = LocalTime.of(14, 32, 45, 123456789);
  - startDateTime = LocalDateTime.of(2015, 10, 31, 14, 32);
  - LocalDate halloween3 = LocalDate.parse("2015-10-31");
  - LocalTime startTime4 = LocalTime.parse("02:32:45");
  - LocalDateTime startDateTime2 = LocalDateTime.parse("2015-10-31T02:32:45.123456789");

- How to get date and time parts
  - The LocalDateTime class supports all of the get methods.
  - The LocalDate class only supports the get methods relevant to getting dates.
  - The LocalTime class only supports the get methods relevant to getting times.
  - LocalDate, LocalTime, and LocalDateTime objects are immutable. As a result, these get methods do not have corresponding set methods.
- Methods for getting parts of date and time objects
  - **getYear()** - Returns the current year as an in integer.
  - **getMonth()** - Returns the current month as a Month object.
  - **getMonthValue()** - Returns the current month as an integer between 1 and 12.
  - **getDayOfMonth()** - Returns the current day of the month as an integer.
  - **getDayOfYear()** - Returns the current day of the year as an integer.
  - **getDayOfWeek()** - Returns the current day of the week as a **DayOfWeek** object.
  - **getHour()** - Returns the current hour of the day as an integer in 24-hour format.
  - **getMinute()** - Returns the current minute of the hour as an integer.
  - **getSecond()** - Returns the current second of the minute as an integer.
  - **getNano()** - Returns the current nanosecond of the second as an integer.

- Code that gets the parts of a LocalDateTime object
- // Assume a current date/time of October 31, 2015 14:32:45.898000000

```
int year = currentDateTime.getYear();
Month month = currentDateTime.getMonth();
int monthValue = currentDateTime.getMonthValue();
int day = currentDateTime.getDayOfMonth();
int dayOfYear = currentDateTime.getDayOfYear();
DayOfWeek dayOfWeek = currentDateTime.getDayOfWeek();
int hour = currentDateTime.getHour();
int minute = currentDateTime.getMinute();
int second = currentDateTime.getSecond();
int nano = currentDateTime.getNano();
```

- How to compare dates and times
  - You can use these methods to compare LocalTime, LocalDate, and LocalDateTime objects.
  - **isBefore(**dateTime**)** - Returns true if the date or time is before the other specified date or time. Otherwise, this method returns false.
  - **isAfter(**dateTime**)** - Returns true if date or time is after the other specified date or time. Otherwise, this method returns false.
  - **compareTo(**dateTime**)** - Returns a negative value if the date or time is before the other specified date or time, a positive value if the date or time is after, and 0 if they are equal.
- Code that uses the isBefore method

```
LocalDate currentDate = LocalDate.now();
LocalDate halloween = LocalDate.of(2015, Month.OCTOBER, 31);
      if (currentDate.isBefore(halloween)) {
            System.out.println("Current date is before Halloween.");
      }
}
```

- Code that uses the isAfter method

```
LocalTime currentTime = LocalTime.now();
LocalTime startTime = LocalTime.of(15, 30);
if (currentTime.isAfter(startTime)) {
    System.out.println("Current time is after start time.");
}
```

- Code that uses the compareTo method

```
LocalDate currentDate = LocalDate.now();
LocalDate halloween = LocalDate.of(2015, Month.OCTOBER, 31);
if (currentDate.compareTo(halloween) < 0) {
    System.out.println("Current date is BEFORE Halloween.");
} else if (currentDate.compareTo(halloween) > 0) {
    System.out.println("Current date is AFTER Halloween.");
} else if (currentDate.compareTo(halloween) == 0) {
    System.out.println("Current date is Halloween.");
}
```

- **How to adjust date/time objects**
  - To adjust a date/time object, you can use the with methods.
  - These methods create a new object from the existing date/time object. In other
  - words, they don’t alter the existing date/time object.
  - The LocalDateTime class supports all of the methods in the table.
  - The LocalDate class only supports the methods relevant to dates.
  - The LocalTime class only support the methods relevant to times.
  - These methods can throw a **DateTimeException** for arguments that are out of range.
  - The **withMonth** method may change the day if the current day stored in the object is out of range for the new month.

- **Methods for adjusting dates and times**
  - **withDayOfMonth(**day**)** - Returns a new object based on the original with the day of month changed to day.
  - **withDayOfYear(**dayOfYear**)** - Returns a new object based on the original with the month and day set to dayOfYear (1 to 365).
  - **withMonth(**month**)** - Returns a new object based on the original with the month changed to month.
  - **withYear(**year**)** - Returns a new object based on the original with the year changed to year.
  - **withHour(**hour**)** - Returns a new object based on the original with the hour changed to hour.
  - **withMinute(**minute**)** - Returns a new object based on the original with the minute changed to minute.

- An example that changes the month to December

```
LocalDate date = LocalDate.of(2015, 10, 20);
LocalDate newDate = date.withDayOfMonth(31);
```

- An example that throws an exception due to an invalid day of month

```
LocalDateTime dateTime1 = LocalDateTime.parse("2015-02-28T15:30");
LocalDateTime newDateTime1 = dateTime1.withDayOfMonth(29);
// Throws a DateTimeException because 2015 is not a leap year.
// As a result, February only has 28 days
```

- An example that quietly changes the day of month

```
LocalDateTime dateTime2 = LocalDateTime.parse("2015-10-31T15:30");
LocalDateTime newDateTime2 = dateTime2.withMonth(2);
// Does not throw an exception, but quietly changes the day to 28
// because there are only 28 days in February 2015.
```

- How to add or subtract a period of time
  - To adjust the date or time forward or backward, you can use the plus and minus methods with the **ChronoUnit (java.time.temporal)** enumeration to specify a period of time. However, it's usually easier to use the shortcut methods such as the **plusWeeks** or **minusWeeks** methods.
  - Shortcut methods exist for all of the ChronoUnit constants shown in this figure.
  - The LocalDateTime class supports all of the methods and ChronoUnit constants. The LocalDate and LocalTime classes only support the methods and constants that are relevant to them.
  - All of the plus and minus methods return new objects of the same type. In other words, they do not alter the existing objects.
  - You can use method chaining to perform multiple calculations in a single statement.
- Methods for adding or subtracting time
  - **plus(**long, chronoUnit**)** - Returns a new object after adding the specified amount of time.
  - **minus(**long, chronoUnit**)** - Returns a new object after subtracting the specified amount of time.
- Common constants of the ChronoUnit enumeration
  - ChronoUnit.YEARS, ChronoUnit.MONTHS, ChronoUnit.WEEKS,
  - ChronoUnit.DAYS, ChronoUnit.HOURS, ChronoUnit.MINUTES, ChronoUnit.SECONDS

- Code that adds three weeks to the current date/time
  - LocalDateTime newDateTime = currentDateTime.plus(3, ChronoUnit.WEEKS);
- Code that subtracts three weeks from the current date/time
  - LocalDateTime newDateTime = currentDateTime.minus(3, ChronoUnit.WEEKS);
- Code that adds three hours to the current time
  - LocalTime newTime = currentTime.plus(3, ChronoUnit.HOURS);
- Shortcut methods for adding or subtracting weeks
  - **plusWeeks(**long**)** - Returns a new object after adding the specified number of weeks.
  - **minusWeeks(**long**)** - Returns a new object after subtracting the specified number of weeks.
- Shortcut methods for the previous three examples
  - LocalDateTime newDateTime = currentDateTime.plusWeeks(3);
  - LocalDateTime newDateTime = currentDateTime.minusWeeks(3);
  - LocalTime newTime = currentTime.plusHours(3);
- Code that uses method chaining
  - LocalDateTime newDateTime = currentDateTime.plusWeeks(3).plusHours(3);

- How to format dates and times
  - You can use the **ofLocalized** methods to format a date and time for the locale of the system that the application is running on.
  - You can use the **FormatStyle** constants to specify the format style for the date and time.
  - If you attempt to use the LONG or FULL style to format a time, the compiler throws a **DateTimeException**.
  - If none of the formats included with the API meets your needs, you can write your own custom formatters.
- A class and an enumeration for formatting dates and times
  - java.time.format.DateTimeFormatter
  - java.time.format.FormatStyle
- Common static methods of the DateTimeFormatter class
  - **ofLocalizedDate(**dateStyle**)** - Returns a DateTimeFormatter object for the date, but not time.
  - **ofLocalizedTime(**timeStyle**)** - Returns a DateTimeFormatter object for the time, but not date.
  - **ofLocalizedDateTime(**dateTimeStyle**)** - Returns a DateTimeFormatter object for the date and time.
  - **ofLocalizedDateTime(**dateStyle, timeStyle**)** - Returns a DateTimeFormatter object for the date and time, but with different formatting styles used for each.

- A common method of the DateTimeFormatter class
  - **format(**dateTime**)** - Returns a String object for the formatted date/time.
- Constants of the FormatStyle enumeration

Constant	Date Example	Time Example
FormatStyle.SHORT	10/31/15	6:30 PM
FormatStyle.MEDIUM	Oct 31, 2015	6:30:00 PM
FormatStyle.LONG	October 31, 2015	DateTimeException
FormatStyle.FULL	Saturday, October 31, 2015	DateTimeException

- Code that uses the same style to format both the date and time
  - LocalDateTime currentDateTime = LocalDateTime.now();
  - DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
  - String currentDateTimeFormatted = dtf.format(currentDateTime);
- Code that uses a separate style for the date and time
  - DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG, FormatStyle.SHORT);
  - String currentDateTimeFormatted = dtf.format(currentDateTime);