Contents

12. Working with Selected classes from the Java API

1. Create and manipulate Strings
2. Manipulate data using the StringBuilder class and its methods
3. Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period
4. Declare and use an ArrayList of a given type
5. Write a simple Lambda expression that consumes a Lambda Predicate expression

12.1 Create and manipulate Strings

12.1.1 Creating Strings

Java uses the `java.lang.String` class to represent character strings. Strings such as `"1234"` or `"hello"` are really objects of this class. In the Java world, `String` objects are usually just called **"strings"**. I talked about the basics of strings briefly in the '**Using Operators'** chapter. I also showed you how to create strings by directly typing the value within double quotes and by using constructors defined in the `java.lang.String` class. As you may recall, all strings are stored in heap space but when you create a string directly without using the constructor, that string is stored in a special area of heap space known as the **"string pool"**.

`String` is a **final** class, which means it cannot be extended. It extends `Object` and implements `java.lang.CharSequence`.

Creating strings through constructors

The `String` class has several constructors but for the purpose of the exam, you only need to be aware of the following:

1. **String()** - The no-args constructor creates an empty String.
2. **String(String str), String(StringBuilder sb)** - Create a new String by copying the sequence of characters currently contained in the passed String or StringBuilder objects.
3. **String(byte[] bytes)** - Creates a new String by decoding the specified array of bytes using the platform's default charset.
4. **String(char[] value)**- Creates a new String so that it represents the sequence of characters currently contained in the character array argument.

Note that a string is composed of an array of `char`s. But that does not mean a string is the same as a `char` array. Therefore, you cannot apply the array indexing operator on a string. Thus, something like `char c = str[0];`, where `str` is a `String`, will not compile.

Creating strings through concatenation

The second common way of creating strings is by using the concatenation (, i.e., +) operator:

```
String s1 = "hello ";
String s12 = s1 + " world"; //produces "hello world"
```

The + operator is overloaded in such a way that if either one of its two operands is a string, it converts the other operand to a string and produces a new string by joining the two. There is no restriction on the type of operands as long as one of them is a string.

The way + operator converts the non-string operand to a string is important:

1. If the non-string operand is a reference variable, the `toString()` method is invoked on that reference to get a string representation of that object.
2. If the non-string operand is a primitive variable or a primitive literal value, a wrapper object of the same type is created using the primitive value and then a string representation is obtained by invoking `toString()` on the wrapper object.
3. If the one of the operands is a `null` literal or a null reference variable, the string `"null"` is used instead of invoking any method on it.

The following examples should make this clear:

```
String s1 = "hello ";
String s11 = s1 + 1; //produces "hello 1"

String s12 = 1 + " hello"; //produces "1 hello"

String s2 = "" + true; //produces "true";

double d = 0.0;
String s3 =  "-"+d +"-"; //produces "-0.0-"

Object o = null;
String s4 = "hello "+o; //produces "hello null". No NullPointerException here.
```

Just like a mathematical expression involving the + operator, string concatenation is also evaluated from left to right. Therefore, while evaluating the expression `"1"+2+3`, `"1"+2` is evaluated first to produce `"12"` and then `"12"+3` is evaluated to produce `"123"`. On the other hand, the expression `1 + 2 +"3"` produces `"33"`. Since neither of the operands to `+` in the expression `1 + 2` is a `String`, it will be evaluated as a mathematical expression and will therefore, produce integer `3`. `3 + "3"` will then be evaluated as `"33"`.
Remember that to elicit the overloaded behavior of the + operator, at least one of its operands must be a `String`. That is why, the following statements will not compile:

```
String x = true + 1;

Object obj = "string";
String y = obj + obj; //even though obj points to a String at runtime, as far as the
    compiler is concerned, obj is an Object and not a String
```

Since the `toString` method is defined in the `Object` class, every class in Java inherits it. Ideally, you should override this method in your class but if you do not, the implementation provided by the `Object` class is used. Here is an example that shows the benefit of overriding `toString` in a class:

```
class Account{
   String acctNo;
   Account(String  acctNo){
      this.acctNo = acctNo;
   }
```

```
   //overriding toString
   //must be public because it is public in Object
   public String toString(){
      return "Account["+acctNo+"]";
   }
}

public class TestClass{
   public static void main(String[] args){
       Account a = new Account("A1234");
       String s = "Printing account  - "+a;
       System.out.println(s);
   }
}
```

The above code produces the following output with and without overriding `toString`:

```
Printing account  - Account[A1234]
```

and

```
Printing account  - Account@72bfaced
```

Observe that when compared to `Object`'s `toString`, `Account`'s `toString` generates a meaningful string. Since the `Object` class has no idea about what a class represents, it just returns a generic string consisting of the name of the class of the object, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. Don't worry, you will not be asked to predict this value in the exam. Just don't get scared if you see such a value in the exam.

On a side note, the `print/println` methods that we have often used also behave just like the + operator with respect to generating a string representation of the object that is passed to them. For example, when you call `System.out.print(acct);` where acct refers to an `Account` object, the print method invokes `toString` on that `Account` object and prints the returned string.

The += operator

In the chapter on operators,we saw that `+=` is a compound operator. It applies the `+` operator on the two operands and then assigns the result back to the variable on the left side. As is the case with the `+` operator, the string concatenation behavior of `+=` is triggered when the type of any one of its operands is `String`. Here is an example:

```
String s = "1";
s += 2; //expanded to s = s + 2;
System.out.println(s); //prints "12"
```

Furthermore, if the result of the `+=` operator is a string, the type of the operand on the left must be something that can refer to a string, otherwise, the expression will not compile. There are only 4 such types other than `String` - the super classes of `String`, i.e., `CharSequence` and `Object` and, the interfaces that `String` implements, i.e., `Serializable` and `Comparable`. Here is an example:

```
int x = 1;
x += "2"; //will not compile
```

Since the type of one of the operands in the above expression is `String`, the String concatenation behavior of `+=` will be triggered. However, the expression will not compile because you can't assign the resulting object of type `String` to a variable of type `int`.

Observe that if the type of the left operand is `String`, the type of the right operand can be anything because in that case, even if the type of the right operand is not `String`, it will be converted to a string as per the rules discussed above.

Now, can you tell what the following code will print?

```
Object m = 1;
m += "2";
System.out.println(m);
```

It will compile fine and print `"12"`. First, `1` will be boxed into an `Integer` object, which will be assigned to `m`. This assignment is valid because an `Integer` **"is-a"** `Object`. Next, the expression `m += "2"` will be expanded to `m = m + "2"`. Since one of the operands of `+` in this expression is a string, a string concatenation will be performed, which will produce the string `"12"`. This string will be assigned to `m`. The assignment is also valid because a `String` is an `Object`.

12.1.2 String immutability

Strings are immutable. It is impossible to change the contents of a string once you have created it. Let me show you some code that looks like it is altering a string:

```
String s1 = "12";
s1 = s1+"34";
System.out.println(s1); //prints 1234
```

The output of the above code indicates that the string pointed to by `s1` has been changed from `"12"` to `"1234"`. However, in reality, the original string that `s1` was pointing to remains as it is and a new string containing `"1234"` is created, whose address is assigned to `s1`. After the last line of the above code, the string pool will contain three different strings - `"12"`, `"34"`, and `"1234"`.

There are several methods in the `String` class that may make you believe that they change a string but just remember that a string cannot be mutated. Ever. Here are some examples:

```
String s1 = "ab";
s1.concat("cd");
System.out.println(s1); //prints "ab"
s1.toUpperCase();
System.out.println(s1); //prints "ab"
```

In the above code, `s1.concat("cd")` does create a new string containing `"abcd"` but this new string is not assigned to `s1`. Therefore, the first println statement prints `"ab"` instead of `"abcd"`.

The same thing happens with `toUpperCase()`. It does produce a new string containing `"AB"` but since this string is not assigned to `s1`, the second println statement prints `"ab"` instead of `"AB"`. Note that the newly created strings `"abcd"` and `"AB"` will remain in the string pool. The JVM will use them whenever it needs to create a string containing the same characters. But as of now, we don't have any reference that points to these strings.

Mutability

Generally, **mutability** refers to the properties of an object of a class. Thus, an immutable class implies that the instance variables of an object cannot be changed once the instance is created. This is achieved by making instance variables private and having only getter methods for reading their values. For example, the following class is immutable because there is no way to change the contents of a Moo object once it is created:

```
class Moo{
   private String value;
   Moo(String s){ this.value = s; }
   public String getValue(){
      return value;
   }
}
```

Although not important for the exam, you should be aware that interviewers like to dig deeper into the immutability of a class. Consider what will happen if, instead of `String`, the type of the value field is some other class such as `java.util.List`:

```
import java.util.*;
class Moo{
   private List value;
   Moo(List ls){ this.value = ls; }
   public List getValue(){
      return value;
   }
}
```

It is not possible for anyone to change `value` to point to another `List` object. But it is possible for other classes to change the contents of the `List` pointed to by `value` once they get hold of the reference of that List using `getValue()`. To some, this implies that an object of `Moo` is not really immutable. Depending on the requirements of the application, you may want to change the getter method to return a copy of the list:

```
public List getValue(){
    return new ArrayList(value);
}
```

Now, the `List` contained in a Moo object cannot be accessed by any other class and is therefore, immutable but what about the objects contained in the list? Well, you have to draw the line based on the application requirements. Mutability may also refer to the mutability of the class itself. You

can change the behavior of a class by overriding its methods in a subclass. This can be prevented by making the class **final**. As mentioned before, `String` is a good example of a final class.

12.1.3 Manipulating Strings

The `String` class contains several methods that help you manipulate strings. To understand how these methods work, you may think of a string as an object containing an array of chars internally. These methods simply work upon that array. Since array indexing in Java starts with 0, any method that deals with the locations or positions of characters in a string, also uses the same indexing logic. Furthermore, any method that attempts to access an index that is beyond the range of this array throws `IndexOutOfBoundsException`.
Here are the methods that belong to this category with their brief JavaDoc descriptions:

1. `int length()` - Returns the length of this string.

   For example, `System.out.println("0123".length());` prints `4`. Observe that the index of the last character is always one less than the length.

2. `char charAt(int index)` - Returns the char value at the specified index. Throws `IndexOutOfBoundsException` if the index argument is negative or is not less than the length of this string.

   For example, `System.out.println("0123".charAt(3));` prints `3`.

3. `int indexOf(int ch)-` Returns the index within this string of the first occurrence of the specified character. Returns -1 if the character is not found.

   Examples:

   ```
   System.out.println("0123".indexOf('2')); //prints 2
   System.out.println("0123".indexOf('5')); //prints -1
   ```

A design philosophy followed by the Java standard library regarding methods that deal with the starting index and the ending index is that the starting index is always inclusive while the ending index is always exclusive. `String`'s `substring` methods works accordingly:

1. `String substring(int beginIndex, int endIndex)` - Returns a new string that is a substring of this string.

   Examples:
   `System.out.println("123456".substring(2, 4)); //prints 34.`
   Observe that the character at index 4 is not included in the resulting substring.
   `System.out.println("123456".substring(2, 6)); //prints 3456`
   `System.out.println("123456".substring(2, 7)); //throws StringIndexOutOfBoundsException`

2. `String substring(int beginIndex)` - This method works just like the other substring method except that it returns all the characters from beginIndex (i.e. including the character at the beginindex) to the end of the string (including the last character).

   Examples:
   `System.out.println("123456".substring(2)); //prints 3456.`
   `System.out.println("123456".substring(7)); //throws StringIndexOutOfBoundsException.`

Note

The rule about not including the element at the ending index is followed not just by the methods of the String class but also by methods of other classes that have a concept of element positions such as java.util.ArrayList.

The following methods return a new `String` object with the required changes:

1. `String concat(String str)` - Concatenates the specified string to the end of this string.

   Example - `System.out.println("1234".concat("abcd")); //prints 1234abcd`

2. `String toLowerCase()/toUpperCase()` - Converts all of the characters in this String to lower/upper case.

   Example - `System.out.println("ab".toUpperCase()); //prints AB`

3. `String replace(char oldChar, char newChar)` - Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

   Example: `System.out.println("ababa".replace('a', 'c')); //prints cbcbc`

4. `String trim()` - Returns a copy of the string, with leading and trailing whitespace omitted. Example: `System.out.println(" 123 ".trim()); //prints "123" (without the quotes, of course)`

One interesting thing about the String manipulation methods detailed above is that they return the same string if there is no change in the string as a result of the operation. Thus, all of the following print statements print `true` because all of these operations return the same String object:

```
String s1 = "aaa"; //size of this string is 3
System.out.println(s1.substring(0,3) == s1); //prints true because the resulting
    substring is the same as the original string
System.out.println(s1.substring(0) == s1); //prints true because the resulting
    substring is the same as the original string
System.out.println(s1.replace('b', 'c') == s1); //nothing is replaced because there is
    no b in the string
```

```
System.out.println(s1.strip() == s1); //there is nothing to strip at the ends of the
    original string
```

It is very common to invoke these methods in the same line of code by chaining them together:

```
String str = "  hello ";
str = str.concat("world ").trim().concat("!").toUpperCase();
System.out.println(str);
```

The above code prints `HELLO WORLD!`. Note that such chaining is possible only because these methods return a string. You will see a similar chaining of methods in `StringBuilder`/`StringBuffer` classes as well. Finally, here are a few methods that let you inspect the contents of a string:

1. `boolean startsWith(String prefix)`: Returns true if this string starts with the specified prefix.
2. `boolean endsWith(String suffix)`: Returns true if this string ends with the specified suffix.
3. `boolean contains(CharSequence s)`: Returns true if and only if this string contains the specified sequence of char values.
4. `boolean equals(Object anObject)`: Returns true if the contents of this string and the passed string are exactly same. Observe that the type of the parameter is `Object`. That's because this method is actually defined in the Object class and the String class overrides this method. So, you can pass any object to this method, but if that object is not a string, it will return false.
5. `boolean equalsIgnoreCase(String anotherString)`: Compares this String to another String, ignoring case considerations.
6. `boolean isEmpty()`: Returns true if, and only if, length() is 0.

The above methods are fairly self-explanatory and work as one would expect after looking at their names, so, I am not going to talk about them in detail here but I suggest you take a look at their JavaDoc descriptions and write a few test programs to try them out. You will not get any trick questions on these methods in the exam.

12.2 Manipulate data using the StringBuilder class and its methods

12.2.1 Why StringBuilder

`java.lang.StringBuilder` is the mutable sibling of `java.lang.String`. Both the classes directly extend `Object` and implement `CharSequence`. Both are `final` as well.

You may be wondering why we need another class to deal with strings if `String` allows us to do everything that we could possibly want to do with strings! Well, besides being mutable, `StringBuilder` is quite different from `String` due to the fact that `StringBuilder` objects are treated just like objects of other regular classes by the JVM. There is no "string pool" or "interning"

associated with StringBuilder objects. StringBuilder objects are garbage collected just like other objects once they go out of scope, which means they don't keep occupying memory forever. This makes StringBuilder objects more suitable for creating temporary strings that have no use once a method ends, such as, creating lengthy debug messages or building long xml documents. It may be hard to believe but a program can create a large number of String objects pretty quickly. For example, the following trivial code generates an HTML view for displaying a list of names in a browser:

```
public String showPerson(List persons){
   String html = "<h3>Persons</h3>";
   for(Object o : persons){
     Person p = (Person) o;
     html = html +p.getName()+"<br class=\"mynewline\" >";
   }
   return html;
}
```

The above code has the potential to wreak havoc on a program's memory. Depending on the size of the list, it will create a large number of String objects and all of them will sit in the memory for a long time, possibly for the entire life-time of the program. The same method, written using StringBuilder, is a lot more efficient:

```
public StringBuilder showPerson(List persons){
   StringBuilder html = new StringBuilder("<h3>Persons</h3>");
   for(Object o : persons){
     Person p = (Person) o;
     html.append(p.getName()).append("<br class=\"mynewline\" >");
   }
   return html;
}
```

It creates exactly two String objects and exactly one StringBuilder object irrespective of the number of elements in the List. Furthermore, the StringBuilder object will be garbage collected as soon as it goes out of scope.

On the other hand, since String objects are interned, they are more suitable for creating short strings that are used repeatedly in a program (For example, `"<br class=\"mynewline\" >"` in the above code). Also, if you want to use strings in a switch statement, then String is the only option.

12.2.2 StringBuilder API

StringBuilder provides several constructors and methods and the exam expects you to know most, if not all, of them. Let's go over the constructors first:

1. StringBuilder(): Constructs a StringBuilder with no characters in it and an initial capacity of 16 characters. Here, "capacity" refers to the size of an internal array that is used to store the characters. Initially, this array is empty and is filled up as you start adding characters to the StringBuffer. The StringBuilder object automatically allocates a new array with larger

size once this array is full.
The capacity of a StringBuilder is analogous to a bucket of water. It is empty at the beginning and fills up as you add water to it. Once it is full, you need to get a bigger bucket and transfer the water from the smaller bucket to the bigger one.

2. `StringBuilder(CharSequence seq)`: Constructs a StringBuilder that contains the same characters as the specified CharSequence. Recall that `String` implements `CharSequence` as well. Thus, this constructor can be used to create a new StringBuilder with the same data as an existing String or StringBuilder.

3. `StringBuilder(int capacity)`: Constructs a StringBuilder with no characters in it and an initial capacity specified by the capacity argument. If you expect to build a large string, you can specify a big capacity at the beginning to avoid reallocation of the internal storage array later. For example, if you are building an HTML page in a method, you might want to create a StringBuilder with a large initial capacity.
It is important to understand that specifying a capacity does not mean you can store only that many characters in the StringBuilder. Once you fill up the existing capacity, the StringBuilder will automatically allocate a new and larger array to store more characters.

4. `StringBuilder(String str)`: Constructs a StringBuilder initialized to the contents of the specified string. This constructor is actually redundant because of the `StringBuilder(CharSequence seq)` constructor. It exists only for backward compatibility with code written before JDK 1.4, which is when `CharSequence` was first introduced.

Since the whole purpose of having a `StringBuilder` is to have mutable strings, it is no wonder that it has a ton of overloaded append and insert methods. But don't be overwhelmed because all of them follow the same pattern. The append method only takes one argument. This argument can be of any type. The insert method takes two arguments - an `int` to specify the position at which you want to insert the second argument. Both the methods work as follows:

1. If you pass a `CharSequence` (which, again, implies `String` and `StringBuilder`) or a `char[]`, each character of the `CharSequence` or the char array is appended to or inserted in the existing StringBuilder.

2. For everything else, `String.valueOf( ... )` is invoked to generate a string representation of the argument that is passed. For example, `String.valueOf(123)` returns the String `"123"`, which is then appended to or inserted in the existing StringBuilder. In case of objects, valueOf invokes `toString()` on that object to get its string representation.

3. If you pass a `null`, the string `"null"` is appended to or inserted in the existing StringBuilder. No `NullPointerException` is thrown.

4. All of the `append` and `insert` methods return a reference to the same `StringBuilder` object. This makes it easy to chain multiple operations. For example, instead of writing `sb.append(1); sb.insert(0, 2);`, you can write `sb.append(1).insert(0, 2);`

Here are a few examples of how the append methods work:

```
StringBuilder sb = new StringBuilder(100); //creating an empty StringBuilder with an
    initial capacity of 100 characters

   sb.append(true); //converts true to string "true" and appends it to the existing
    string
   System.out.println(sb); //prints true

   sb.append(12.0); //converts 12.0 to string "12.0" and appends it to the existing
    string
   System.out.println(sb); //prints true12.0


   sb.append(new Object()); //calls toString on the object and appends the result to
    the existing string
   System.out.println(sb); //prints true12.0java.lang.Object@32943380
```

And here are a couple of examples to illustrate the insert methods:

```
StringBuilder sb = new StringBuilder("01234");

   sb.insert(2, 'A'); //converts 'A' to string "A" and inserts it at index 2
   System.out.println(sb); //prints 01A234

   sb.insert(6, "hello"); //inserts "hello" at index 6
   System.out.println(sb); //prints 01A234hello
```

In the above code, observe the location at which the string is being inserted. As always, since indexing starts with 0, the first position at which you can insert a string is 0 and the last position is the same as the length of the existing string. If your position argument is negative or greater than the length of the existing string, the insert method will throw an `StringIndexOutOfBoundsException`. The rest of the methods are quite straightforward and work as indicated by their names. To make them easy to remember, I have categorized them into two groups - the ones that return a self-reference (i.e. a reference to the same StringBuilder object on which the method is invoked), which implies they can be chained, and the ones that do not. Methods that return a self-reference are - `reverse()`, `delete(int start, int end)`, `deleteCharAt(int index)`, and `replace(int start, int end, String replacement)`. Remember that **start index** is always **inclusive** and **end index** is always **exclusive**, so, the following code will print `0abcd34` and `0cd34`.

```
StringBuilder sb = new StringBuilder("01234");

   sb.replace(1, 3, "abcd"); //replaces only the chars at index 1 and 2 with "abcd"
   System.out.println(sb); //prints 0abcd34

   sb.delete(1, 3); //deletes only the chars at index 1 and 2
   System.out.println(sb); //print 0cd34
```

Methods that cannot be chained are `int capacity()`, `char charAt(int index)`, `int length()`, `int indexOf(String str)`, `int indexOf(String str, int startIndex)`, `void`

`setLength(int len)`, `String substring(int start)`, `String substring(int start, int end)`, and `String toString()`.
The `setLength` method is interesting. It truncates the existing string contained in the StringBuilder to the length passed in the argument. Thus, `StringBuilder sb = new StringBuilder("01234"); sb.setLength(2);` will truncate the contents of `sb` to `01`.

delete vs substring

It is important to understand the difference between the delete and the substring methods of StringBuilder. The delete methods affect the contents of the StringBuilder while the substring methods do not. This is illustrated by the following code:

```
StringBuilder sb = new StringBuilder("01234");

String str = sb.substring(0, 2);
System.out.println(str+" "+sb);

StringBuider  sb2 = sb.delete(0, 2);
System.out.println(sb2+" "+sb);
```

The above code prints `01 01234` and `234 234`.

Note

Not important for the exam but you should be aware that prior to Java 1.5, the Java standard library only had the `java.lang.StringBuffer` classStringBuffer class to deal with mutable strings. This class is thread safe, which means it has a built-in protection mechanism that prevents data corruption if multiple threads try to modify its contents simultaneously. However, the Java standard library designers realized that `StringBuffer` is often used in situations where this protection is not needed. Since this protection incurs a substantial performance penalty, they added `java.lang.StringBuilder` in JDK 1.5, which provides exactly the same API as `StringBuffer` but without the thread safety features. You may see old code that uses `StringBuffer` but unless you want to modify a string from multiple threads, you don't need to use `StringBuffer`. Code with `StringBuffer` will run a little slower than the code that uses `StringBuilder`.

12.3 Create and manipulate calendar data

12.3.1 Overview of the new Date/Time API

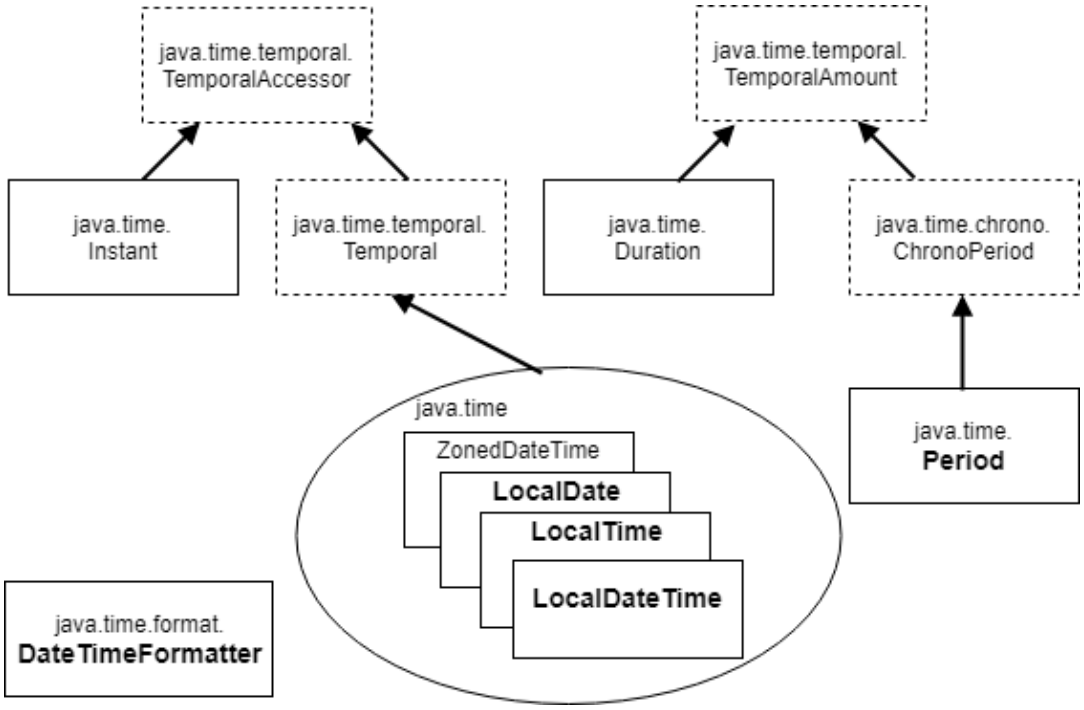
When Java 8 introduced the new Date and Time API, it was Java's third attempt to get the date/time business correctly. The first one was the `java.util.Date` class and the second one was the `java.util.Calandar` class. Both of them were a pain to use and a cause of bugs in Java applications. Given that most non-trivial applications use data/time in some form or the other, you can imagine how important it is to get it right. I only mention this to make you appreciate the fact

that working with date/time is not as simple as it sounds. It is important to get the fundamentals right.

Java 8's Date/Time API is a completely fresh take on managing the date and time. It has no dependency and no relation to the old Date and Calendar classes. Although in real life you will see a lot of code that uses the old date/time classes, I will not talk about them in detail in order to avoid any confusion.

The main cause of complexity in working with time is the fact that machines don't see time the same way as humans do. To machines, time is just a period that has elapsed after a predefined starting point. This point in time is called **"Epoch"** and is set at January 1, 1970 00:00:00.000 at Greenwich. (Greenwich is the name of the town in UK where the Royal Observatory is located. Time at this location was chosen for historical reasons). You can think of it as a long series of milliseconds plotted on an x-axis, where 0 is called January 1, 1970 00:00:00.000 GMT and every millisecond since elapsed moves you forward on this axis. Humans, on the other hand, see time in terms of hours, minutes, and seconds of a specific day, month, and year, at a particular location. For example, to computers, the point 1514786400991 on the time line is 1514786400991 number of milliseconds that have elapsed since the epoch. But to humans, this point could be interpreted as 1AM on January 1st 2018 in New York or 10PM on Dec 31st 2017 in Los Angeles. Old Java classes mix these two very different ideas in the same class. For example, `java.util.Date` actually just stores the number of milliseconds that have elapsed since epoch and has no idea about what date it is. Yet, the name of the class is Date!

The new Date/Time API addresses this concern well and makes it a lot easier to perform routine operations such as date arithmetic and printing. The new date/time framework is packaged in the **java.time** package and has a lot of classes and interfaces. The following diagram shows **a few** important classes and interfaces of this API. Dashed boxes contain interfaces and regular boxes contain classes.



Organization of the new Date/Time API

In the above diagram, five classes are shown in bold font. These are the only classes that are covered in the OCAJP 8 exam. Even though the exam only focuses on these classes, it is important to understand the framework of the related classes as well. Let's take an overview of the packages and classes shown in the above diagram first and then we will go over the details of the classes covered in the exam.

java.time package

This is the main package of the date/time framework. The majority of the application programs only need the classes defined in this package. The classes defined in this package are based on the ISO calendar system (You don't need to know the details of the ISO calendar system). All classes in this package are **immutable** and **thread-safe**.

The most important aspect of these classes is that there is a clear separation between the classes that represent the machine view of the time and the classes that represent the human view of the time.

Instant: This class represents the machine view of time. As discussed before, a computer only cares about the number of milliseconds that have elapsed since Epoch. That is what an Instant contains. It has no notion of AM/PM or time zone. It is a point on the machine view of the timeline.

LocalDate/LocalTime/LocalDateTime/ZonedDateTime: These classes represent the human view of time. An instance of any of these classes represents a point on the human view

of the timeline. Multiple classes are provided so as to avoid complexity where it is not needed. For example, if you just need to capture a date without the time component, you should use **LocalDate**. If you just want the time without the date, you should use **LocalTime** and so on. Furthermore, "local" classes do not contain zone information. If you want to capture zone information, you should use **ZonedDateTime**.

Period: Period represents a date-based amount of time in terms of Days, Months, and Years. It has no notion of hours, minutes, and seconds.

Duration: Duration represents time-based amount of time in terms of Hours, Minutes, and Seconds. It has no notion of days.

> **Note**
>
> Duration is not on the exam, but it is important to understand the difference between Period and Duration. Even though a Period representing 1 day is the same as a Duration representing 24 hours, both are conceptually different. Adding a Period of 1 day to a given date will only increment the day part of the date and will not affect the time component but adding a Duration of 24 hours will add 24 hours to the time part, which in turn, changes the day part as well. For example, adding a Period of 1 Day to the date before day light savings start date, will simply increment the day without affecting the time. Thus, 9th March 2018 10 PM will become 10th March 2018 **10 PM**. But adding a Duration of 24 hours to the same date will change the date to 10th March 2018 **11 PM**, because on this particular day, clocks are moved forward by 1 hour. Therefore, 10PM + 24 hours is not 10PM but 11PM of the next day.

This package also contains two enums - `Month` and `DayOfWeek`. `Month` contains values from `JANUARY` to `DECEMBER` and `DayOfWeek` contains values from `MONDAY` to `SUNDAY`. It is better to use these enums instead of integers while creating date/time objects because they make the code more readable and they also let you avoid the confusion caused by indexing. For example, `LocalDate.of(2018, Month.JANUARY, 10)` is more readable than `LocalDate.of(2018, 1, 10)`.

It appears that the exam only wants to focus on the human view of date/time, which is probably why `Instant` and `Duration` are not included in the objectives.

java.time.format package

This package contains helper classes for converting date/time objects to strings and for parsing strings into date/time classes. **DateTimeFormatter** and **FormatStyle** are the most used classes of this package.

java.time.temporal package

This package contains the base interfaces such as `Temporal`, `TemporalAmount`, `TemporalUnit`, `TemporalField` and `TemporalAccessor` of the Date/Time framework. These interfaces are implemented by various classes of the **java.time** package.

Having common base interfaces for related classes allows applications to operate on the classes uniformly. For example, you can add a `Period` to a `LocalDate` in the same way as you add a `Duration` to a `ZonedDateTime` i.e. using the `add` method on Period or Duration instance. This is possible because both `Period` and `Duration` implement `TemporalAmount`, which declares the `add(Temporal t)` method and both `LocalDate` and `ZonedDateTime` implement `Temporal`.

The exam does not mention this package in the objectives but the whole scheme of things related to date/time manipulation is driven by the interfaces of this package. Many of the methods that you will use while working with date/time classes are actually declared in the interfaces of this package.

If you browse through the JavaDoc of the new Date/Time API, you will see many different classes and many different methods in each of these classes. It is easy to get overwhelmed, but don't worry. The API is actually quite easy to understand once you realize that there are only four types of operations that you can do with these classes: **creating new objects**, **converting them to strings**, **comparing**, and **reading** the state of the objects. There are no methods for updating because the objects of these classes are immutable.

All relevant classes have similar methods to perform these operations. In the following sections, I will explain each of these operations with examples that use only the classes covered in the exam.

12.3.2 Creating date/time objects

The new date/time API provides three distinct ways to create date/time objects. The first and easiest way is to create them by specifying the value for each field of the object. The second is by using the values from an existing date/time object. And the third is by parsing a string. Let's look at each of these ways one by one.

Using the static now/of/from methods

Remember that none of the date/time classes have a public constructor. Therefore, it is not possible to create them using the new operator. If you see something like `new LocalDate()` anywhere in the code, go straight for the option that says, **"compilation fails"**! Instead of having constructors, these classes have **static** factory methods named `now`, `of`, and `from`. All three methods are **public**.

The now method

The **now** method returns an instance of the object that represents the current date or time. For example, `LocalDate.now();` returns a `LocalDate` object set to today's date, `LocalTime.now();` returns a `LocalTime` object set to current time, and `LocalDateTime.now();` returns a `LocalDateTime` object set to today's date and current time. You only need to know the no-args versions of these methods for the exam.

Since `Period` denotes an amount of time and not a point in time, it doesn't have the `now()` method. But it does have a static constant of type `Period` named `ZERO`, which denotes a period of zero.

The of method

The **of** method lets you create an object for any date or time by passing individual components of a date/time. Here is a list of the overloaded **of** methods of the relevant classes:

LocalDate:

```
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
```

LocalTime:

```
static LocalTime of(int hour, int minute)
static LocalTime of(int hour, int minute, int second)
static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
```

LocalDateTime:

```
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int
    second)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int
    second, int nanoOfSecond)
static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute)
static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute,
    int second)
static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute,
    int second, int nanoOfSecond)
static LocalDateTime of(LocalDate date, LocalTime time)
```

As you can see, their signatures are self-explanatory. For example:

```
LocalDate thenLD = LocalDate.of(2018, 1, 10); //10th January 2018
LocalTime someTime = LocalTime.of(10, 53); //10.53 AM
LocalDateTime thenLDT = LocalDateTime.of(2018, 1, 10, 23, 10, 14);
```

There are three points that you should remember about the **of** methods:

1. Indexing for month parameter starts from `1`. In other words, to specify **January**, you must pass `1` (or use the enum value `Month.JANUARY`).

2. They use a **24 hour clock** for the `hour` parameter. Thus, to specify **11 AM**, you need to pass `11` and to specify **11 PM**, you need to pass `23`.

3. They throw `java.time.DateTimeException` (which extends `java.lang.RuntimeException` and is, therefore, an unchecked exception) if the value passed for any field is out of range. For example, `LocalDate.of(2018, 2, 29);` will throw a `DateTimeException` because February of 2018 doesn't have 29 days. Similarly, `LocalTime.of(24, 10);` will also throw a `DateTimeException` because the range of the hour parameter is only 0 to 23.

Period has similar `of` methods:

```
static Period of(int years, int months, int days)
static Period ofDays(int days)
static Period ofMonths(int months)
static Period ofWeeks(int weeks)
static Period ofYears(int years)
```

Here is an example that shows two ways to create a Period with 10 days:

```
Period tenDays = Period.ofDays(10);
tenDays = Period.of(0, 0, 10);
```

The from method

The `from` methods are similar to the `of` methods except that instead of specifying a value for each property individually, you specify another `TemporalAccessor` object. This `TemporalAccessor` object is used as a source for all properties that are required to create the desired object. For example:

```
LocalDateTime ldt = LocalDateTime.now();
LocalDate ld1 = LocalDate.from(ldt);
LocalDate ld2 = LocalDate.from(ld1);
LocalTime lt =  LocalTime.from(ldt);
```

Observe that a `LocalDateTime` has a date as well as a time component and therefore, it is possible to use a `LocalDateTime` to create a `LocalDate` as well as a `LocalTime`. But it is not possible to use a `LocalDate` to create a `LocalDateTime` or a `LocalTime` because `LocalDate` does not have a time component. Thus, the following will throw a `DateTimeException` at run time:

```
LocalDate ld =  LocalDate.now();
LocalDateTime ldt = LocalDateTime.from(ld); //DateTimeException at run time
LocalTime lt = LocalTime.from(ld); //DateTimeException at run time
```

Similarly, you can create a `Period` using another `Period` but not a `Duration` because `Duration` deals with hours, minutes, and seconds, while `Period` needs days, months, and years.

Creating a date/time object using an existing date/time object

Remember that all date/time objects are immutable and therefore, you cannot change them. But you can use the values from an existing date/time to build a new date/time object. This is helpful when the new object that you want to create is slightly different from an existing object. For example, you may already have a `LocalDate` and you want to create a `LocalDateTime` by adding a time component to the same date, or you have a `LocalDateTime` and you want just a `LocalDate`, or you have a `LocalDate` for today and you want a `LocalDate` for tomorrow. The API provides several methods for this purpose.To make them easy to remember, I have categorized these methods on the basis of how they alter the existing values - instance methods that return a new object by altering a field of another object and instance methods that return a new object by adding a new field to an another object.

1. Instance methods that alter an existing field value
Both - a `LocalDate` and a `Period` - have three components: day, month, and year. To modify any of these properties of, you have `plusDays`, `plusWeeks`, `plusMonths`, and `plusYears` and `minusDays`, `minusWeeks`, `minusMonths`, and `minusYears` methods in `LocalDate` as well as in `Period`. Similarly, `LocalTime` has plusHours, plusMinutes, plusSeconds, plusNanos and minusHours, minusMinutes, minusSeconds, minusNanos methods to modify the properties that it supports.

Note that `LocalTime` does not have methods to alter the day, month, and year because `LocalTime` does not have these properties, while `LocalDateTime` has plus and minus methods for all properties of `LocalDate` as well as `LocalTime`.
Example:

```
LocalDate ld1 = LocalDate.now();
LocalDate ld2 = ld1.plusDays(10);
Period p = Period.ZERO.plusDays(10);
```

The **plusXXX** and **minusXXX** methods take only one argument because the name of the property to be updated is implied by their names. There are four more plus and minus methods that do the same thing but in a different way - `plus(long amountToAdd, TemporalUnit unit)`, `plus(TemporalAmount amountToAdd)`, `minus(long amountToSubtract, TemporalUnit unit)`, `minus(TemporalAmount amountToSubtract)`. These methods identify the property to be altered through the arguments that are passed to them.
Example:

```
LocalDate ld1 = LocalDate.now();
LocalDate ld2 = ld1.plus(10, ChronoUnit.DAYS); //add 10 days
LocalDate ld3 = ld1.minus(Period.of(0, 0, 10)); //subtract 0 yrs, 0 months, 10 days
```

Remember from the class hierarchy diagram that I showed earlier that `Period` implements `TemporalAmount`. `ChronoUnit` is an enum that implements `TemporalUnit`. It defines various the units of date/time such as DAYS, MONTHS, YEARS, HOURS, MINUTES, SECONDS, NANOS, and WEEKS.

You need to be careful about using days and months while manipulating dates. Adding a month to a date is not the same as adding 30 days to the date because the number of days in a month depends on the month of the date that you want to manipulate. The following example shows the difference:

```
LocalDate ld = LocalDate.of(2018, 01, 31);
LocalDate ld1 = ld.plusDays(30);
System.out.println(ld1); //prints 2018-03-02
LocalDate ld2 = ld.plus(Period.ofMonths(1));
System.out.println(ld2); //prints 2018-02-28
```

If you try to add to or subtract from a field that is not supported by a particular class, a `DateTimeException` will be thrown. For example, adding hours to a `LocalDate` or adding months to a `LocalTime`, will throw a `DateTimeException` at runtime.

If you want to specify an exact value for a property instead of adding to or subtracting from a property, you can use the `withXXX` methods.

Example:

```
LocalDate ld = LocalDate.now();
LocalDate ld2 = ld.withYear(2019); //change year to 2019
ld2 = ld.withMonth(2); //change month to Feb
ld2 = ld.withDayOfMonth(4); //change date to 4

LocalDateTime ldt = LocalDateTime.now();
ldt = ldt.withYear(2019); //change year to 2019
ldt = ldt.withHour(23); //change hour to 23
```

`Period` has similar **withXXX** methods exception that the names of the methods end with an 's':

```
Period p1 = Period.ZERO.withDays(10);
p1 = p1.withMonths(2);
p1 = p1.withYears(1); //p1 now points to a Period with 1 year, 2 months, and 10 days
```

`Period` also has a **negated** method that returns a new Period with amount for each component negated. For example `Period.of(1, 3, 20).negated();` returns a `Period` of -1 year, -3 months, and -20 days.

2. Instance methods that add new properties to an existing object
What if you have a `LocalDate` and you want a `LocalDateTime` by adding a time component to the existing date? `LocalDate` has several `atTime` methods that let you create a `LocalDateTime` using a `LocalDate`.

Example:

```
LocalDate ld = LocalDate.now();
LocalDateTime ldt = ld.atTime(10, 15); //same date with 10hr 15mins
ldt = ld.atTime(10, 15, 45); //same date with 10hr 15mins 45 secs
```

Similarly, `LocalTime` has `atDate(LocalDate ld)`, which returns a `LocalDateTime` and `LocalDateTime` has `atZone(ZoneId id)`, which returns a `ZonedDateTime`.

```
LocalTime lt = LocalTime.now();
LocalDate ld = LocalDate.now();
LocalDateTime ldt = lt.atDate(ld); //add a LocalDate to LocalTime
```

Note that `LocalDateTime` does not have `atTime` or `atDate` because `LocalDateTime` already has a time as well as a date component. `Period` does not have any **atXXX** method either.

Creating a Date/Time object using the static parse methods

Each of the three date/time classes have two parse methods that return an object of the respective class using information present in the string argument. For example, `LocalDate` has the following two methods:

```
static LocalDate parse(CharSequence text)
static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
```

These are quite straight forward to use:

```
LocalDate ld = LocalDate.parse("2018-02-14");
LocalTime lt = LocalTime.parse("22:10:30");
LocalDateTime ldt = LocalDateTime.parse("2018-02-14T18:12:12");
```

The way these methods parse a given string needs a little explanation. They can't just parse any random string into a date/time. The string that you pass must contain date/time data in a specific format. This format is described by an object of class `java.time.format.DateTimeFormat`. A `DateTimeFormatter` object understands the format in which date/time information is present in a string. While it is possible to design your own format for writing a date, standard formats for writing the date/time strings exist, and `DateTimeFormatter` class provides several readymade `DateTimeFormatter` objects that understand these standard formats. These object can be accessed through public static variables of `DateTimeFormatter` class. You need to be aware of three such variables - `ISO_LOCAL_DATE`, `ISO_LOCAL_TIME`, and `ISO_LOCAL_DATE_TIME` - because when you don't pass a formatter to the parse methods, these are the formatters that the parse methods use internally to parse the string. `LocalDate` uses `ISO_LOCAL_DATE`, `LocalTime` uses `ISO_LOCAL_TIME`, and `LocalDateTime` uses `ISO_LOCAL_DATE_TIME` to parse the given string. Thus, for example, invoking `LocalDate.parse("2018-02-14");` will produce the same result as invoking `LocalDate.parse("2018-02-14", DateTimeFormatter.ISO_LOCAL_DATE);`

For the purpose of the exam, you only need to know that `ISO_LOCAL_DATE` expects strings in `yyyy-MM-dd` format, `ISO_LOCAL_TIME` expects strings in `HH:mm` or `HH:mm:ss` format, and `ISO_LOCAL_DATE_TIME` expects strings in `yyyy-MM-ddTHH:mm:ss` or `yyyy-MM-ddTHH:mm` format (ignoring the nano second part of the time component), where `yyyy` stands for four digit year, `MM` stands for two digit month, `dd` stands for two digit date, `HH` stands for two digit hour, `mm` stands for two digit minute, and `ss` stands for two digit second. You will not be required to parse complicated strings in the exam.

Other formatters that are available in `DateTimeFormatter` are `ISO_DATE`, `ISO_OFFSET_DATE`, `ISO_TIME`, `ISO_OFFSET_TIME`, `ISO_DATE_TIME`, `ISO_OFFSET_DATE_TIME`, and `ISO_ZONED_DATE_TIME`. You do not need to know the formatting details of these formatters, but you do need to know that they exist so that you won't assume an option to be incorrect just because it uses one of these formatters.

The DateTimeFormatter class

Creating a `DateTimeFormatter` to parse a non-standard string is also quite easy. `DateTimeFormatter` has a static `ofPattern(String pattern)` and a static `ofPattern(String pattern, String locale)` method that take the format specification as an argument. For example, if your dates are in **MMM/dd/yy** format, you can create a formatter like this:

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("MMM/dd/yy");
LocalDate ld = LocalDate.parse("Oct/23/19", dtf);
```

```
System.out.println(ld); //prints 2019-10-23
```

Format specification uses predefined characters to denote various components of a date/time. While a detailed understanding of this specification is not required, you should be aware of the meaning of a few basic letters:
`y` is for year
`M` is for month
`d` is for day of month
`H` is for hour of day
`m` is for minute
`s` is for second
`n` is for nanosecond

Parsing a string to create a Period

`Period` also has a parse method but the format of a period string is quite different from date/time. It looks weird to be honest when you look at it for the first time but it actually makes sense. It follows ISO-8601 period format and looks like this - `PnYnMnD` and `PnW`. Here, `n` stands for a number and `P`, `Y`, `M`, `D`, and, `W` are letters that appear in the string. The case of the letters is not important. For example, `P1Y10M3d` means 1 year, 10 months, and 3 days, `P1Y3D` means 1 year and 3 days. `p1Y2w` means 1 year and 2 weeks.

It is now easy to create a `Period` by parsing a string. For example, `Period p = Period.parse("p1Y2w");` creates a `Period` with 1 year and 2 weeks. You can create a `Period` with negative values as well, for example, `"-P2m3d"` and `"P-2m-3d"` will be parsed into the same Period representing -2 months and -3 days.

Again, you will not be asked to parse a complicated string in the exam. The exam does not have trick questions in this area and the basic knowledge given above will be sufficient for you to answer the questions in the exam.

12.3.3 Converting date/time objects to strings

You can get a string representation of a date/time object using either the `toString` method or the `format(DateTimeFormatter dtf)` method.

Just like the single argument `parse` methods, the `toString` method uses the ISO format for generating the string. Here are a few examples:

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

The above code prints:

```
2018-04-29
12:29:08.735
2018-04-29T12:29:08.735
```

Observe that the output for LocalDate does not contain the time component and the output for LocalTime does not contain the date component. Furthermore, the output for LocalDateTime includes a character 'T' that separates the date and the time components.
Here is an example that uses a custom DateTimeFormatter:

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yy MMM dd");
System.out.println(dtf.format(LocalDate.now()));
System.out.println(dtf.format(LocalDateTime.now()));
```

The above code prints:

```
18 Apr 29
18 Apr 29
```

An interesting thing to note in the above output is that no exception is thrown at the third line. The pattern that we used to create the DateTimeFormatter does not include any information for time. Yet, when we used this formatter to format a LocalDateTime, it didn't complain. It simply ignored the time component of the LocalDateTime object. This is totally opposite to how the parse method behaves. If you try to parse a string that has a time component using this formatter, it will throw an exception.

```
LocalDate ld = LocalDate.parse("18 Apr 29 10:10", dtf);
```

The output from the above code is:

```
java.time.format.DateTimeParseException: Text '19 Apr 02 10:10' could not be parsed,
    unparsed text found at index 9
```

By the way, DateTimeParseException extends DateTimeException. Passing a string as an argument to the ofPattern method is an example of "hardcoding". If you want to standardize the format in which you want to print dates across your application, using a hardcoded string is not a good idea because it is prone to typos and misunderstandings. DateTimeFormatter provides a better way to create formatter objects through its static **ofXXX** methods. It has four such methods:

```
static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle
    timeStyle)
static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)
```

The method that should be used depends on the kind of object you want to format. If you have a LocalDate, you should use ofLocalizedDate, if you have a LocalTime, you should use ofLocalizedTime, and if you have a LocalDateTime to format, you should use ofLocalizedDateTime to get the formatter. Using an incompatible formatter, such as using a formatter from ofLocalizedTime to format a LocalDateTime, will cause a DateTimeException to be thrown.

Observe that instead of accepting a string for a format, these methods accept an object of type FormatStyle. FormatStyle is actually an enum with four values: **FULL**, **LONG**, **MEDIUM**, and **SHORT**. You should use a combination of **ofXXX** method and FormatStyle to

create a DateTimeFormatter that satisfies your needs. For example, the following code shows how you can print a date in full as well as in short form:

```
DateTimeFormatter dtf1 = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
DateTimeFormatter dtf2 = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
LocalDate ld = LocalDate.now();
System.out.println(dtf1.format(ld)); //prints Monday, April 30, 2018
System.out.println(dtf2.format(ld)); //prints 4/30/18
```

Similarly, the following code prints a LocalDateTime in short and medium forms:

```
DateTimeFormatter dtf1 = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter dtf2 = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);

LocalDateTime ld = LocalDateTime.now();

System.out.println(dtf1.format(ld)); //prints 4/30/18 1:03 PM
System.out.println(dtf2.format(ld)); //prints Apr 30, 2018 1:03:59 PM
//(Remember that the exact output depends on your default locale)
```

Using FormatStyle.LONG or FormatStyle.FULL for formatting a LocalDateTime or LocalTime will throw an exception at run time because these forms expect time zone information in the time component, which is not present in LocalDateTime or LocalTime. The exam does not expect you to remember the formats in which these formatters print the date/time. You only need to know how standardized formatters can be created using the ofXXX methods mentioned above instead of the ofPattern methods.

Note that Period does not have any format method. To convert a Period into a string, you have to use its toString method. This method generates a String in the same ISO-8601 format that I talked about before. Here is an example:

```
Period p = Period.parse("p13m5w");
System.out.println(p); //prints P13M35D
```

Observe that 5 weeks have been converted into 35 days but 13 months have remained unchanged.

12.3.4 Comparing date/time objects

There are four ways to compare date/time objects.

1. Using the equals method - All date/time classes override the equals method to compare the contents of the two objects. They only return true if the argument is an object of the same class and their contents match exactly.

   ```
   LocalDate ld1 = LocalDate.now(); //assuming today is 2018-04-30
   LocalDate ld2 = LocalDate.parse("2018-04-30");
   System.out.println(ld1.equals(ld2)); //prints true

   LocalDateTime ltd = LocalDateTime.now();
   System.out.println(ltd.equals(ld2)); //prints false
   ```

Recall that the signature of equals is `equals(Object obj)`, therefore, there is no compilation error when you pass an object of a different class to equals.

2. Using the `compareTo` method - All date/time classes implement `java.util.Comparable` interface, which has a `compareTo` method. Unlike the `equals` method, this method returns an `int`. If the date/time passed in the argument is later, this method returns a negative number, if the argument is exactly same, this method returns 0, and a positive number otherwise.

```
LocalDateTime ltd1 = LocalDateTime.now(); //assuming today is 2018-04-30
LocalDateTime ltd2 = LocalDateTime.parse("2018-05-30T10:20");
System.out.println(ltd1.compareTo(ltd2)); //prints -1

LocalDateTime ltd3 = LocalDateTime.parse("2018-03-30T10:20");
System.out.println(ltd1.compareTo(ltd3)); //prints 1
```

   One might expect that the return value would be the difference between the two dates, but the JavaDoc API description for this method does not make any such promise. Thus, you should rely only on the sign of the return value and not on the actual value.

3. Using the `isAfter/isBefore` methods - As their names suggest, these methods check if this date is after or before the date/time passed in the argument.

```
LocalDate ld1 = LocalDate.now(); //assuming today is 2018-04-30
LocalDate ld2 = LocalDate.parse("2018-04-29");
System.out.println(ld1.isAfter(ld2)); //prints true
System.out.println(ld1.isBefore(ld2)); //prints false

LocalDate ld3 = LocalDate.parse("2018-04-29");
System.out.println(ld3.isAfter(ld2)); //prints false
System.out.println(ld3.isBefore(ld2)); //prints false
```

4. Using the `isEqual` method - This method is provided only by `LocalDate` and `LocalDateTime`. It checks if this date is equal to the date passed in the argument.

```
LocalDate ld1 = LocalDate.now(); //assuming today is 2018-04-30
LocalDate ld2 = LocalDate.of(2018, 4, 29);
System.out.println(ld1.isEqual(ld2)); //prints false
System.out.println(ld1.isEqual(ld1)); //prints true
```

You may be wondering why we need the `isBefore/isAfter/isEqual` methods when we already have the `compareTo` and `equals` methods. Although this is beyond the scope of the exam, the difference between them is that `isBefore/isAfter/isEqual` methods do not take chronology, i.e., the calendar system into account while comparing, but `equals` and `compareTo` do. This difference becomes important when you compare a date from, say, a Thai calendar with a date from the regular ISO calendar.

`Period` has only three instance methods that deal with comparison - `equals`, `isZero`, and `isNegative`. Equals only returns true if all three components of the two Periods match. IsZero

only returns true if all the three components of the Period are zero. IsNegative only returns true if any of the three components of the Period is less than zero.

12.3.5 Reading the state of date/time objects

All date/time objects allow reading of their properties through appropriate getters. For example, JavaDoc API description for `LocalDateTime` lists the following getters: `int getDayOfMonth()` : Gets the day-of-month field.
`DayOfWeek getDayOfWeek()` : Gets the day-of-week field, which is an enum DayOfWeek.
`int getDayOfYear()` : Gets the day-of-year field.
`int getHour()` : Gets the hour-of-day field.
`int getMinute()` : Gets the minute-of-hour field.
`Month getMonth()` : Gets the month-of-year field using the Month enum.
`int getMonthValue()` : Gets the month-of-year field from 1 to 12.
`int getNano()` : Gets the nano-of-second field.
`int getSecond()` : Gets the second-of-minute field.
`int getYear()` : Gets the year field.

Besides the above, it also has the following two special `get` methods:

`int get(TemporalField field)` : Gets the value of the specified field from this date-time as an int.
`long getLong(TemporalField field)` : Gets the value of the specified field from this date-time as a long.

I haven't heard of anyone ever getting a question on these two methods but it won't do any harm if you are aware of their existence.

`LocalDate` and `LocalTime` have the same get methods except for the properties that they don't have. For example, `LocalDate` doesn't have `getHour` and `LocalTime` doesn't have `getYear`.

Properties supported by java.time.Period

Period has the following getters:
`int getDays()`: Gets the amount of days of this period.
`int getMonths()`: Gets the amount of months of this period.
`int getYears()`: Gets the amount of years of this period.
`long get(TemporalUnit unit)` : Gets the value of the requested unit.

Note that I have merely reproduced the descriptions from the JavaDoc because there is not much to discuss about them. They work as indicated by their names. The exam does not have trick questions on this aspect either.

12.3.6 Chaining method calls

Most of the methods that you have seen thus, far for creating new date/time objects (i.e. now, of, ofXXX, from, plusXXX, minusXXX and withXXX) return a reference of the same type. That makes it easy for you to customize the object you want by chaining multiple method calls in one statement. For example:

```
LocalDate ld = LocalDate.now().withYear(2010).plusMonths(2).minusDays(10);
```

First, you should get used to evaluating the result of such chained method calls because you will see such code on the exam. Second, evaluating such method calls is usually straight forward but you need to watch out for the following tricks that you will encounter in the exam -

1. **Watch out for lost objects** - I have said it multiple times already but since it is very easy to forget, I will say it again - all date/time objects are immutable. Therefore, none of the methods change the object they are invoked on. They return a new object. If you don't assign this new object to a variable, you won't see the result of your method calls. If you can tell what the following code will print, you know what I am talking about:

   ```
   LocalDate ld = LocalDate.of(2018, 1, 1);
   ld.plusMonths(2).plusDays(10);
   System.out.println(ld);
   ```

   You should change the second line to `ld = ld.plusMonths(2).plusDays(10);` if you want to see `2018-03-11` in the output.

2. **Watch out for return types** - Unlike the other creation methods, the **atXXX** methods return a reference of a different type. For example, `LocalDate`'s `atTime` method returns a `LocalDateTime`. If a call to this method is embedded in the call chain, pay close attention to the type of the reference variable to which the result is being assigned. For example, the following code will not compile:

   ```
   LocalDate ld = LocalDate.of(2018, 1, 1);
   ld = ld.withYear(2010).plusDays(10).atTime(10, 20).plusMonths(2);
   ```

   Observe that the `of` method of `LocalDate` is being invoked above and therefore, it returns a `LocalDate`. For the same reason, the call to `plusDays` also returns a `LocalDate`. But the call to `atTime` on a `LocalDate` returns a `LocalDateTime` and therefore, the next call to `plusMonths` in the chain returns a `LocalDateTime`. A `LocalDateTime` object cannot be assigned to a variable of class `LocalDate`.

3. **Watch out for unsupported properties** - You can't add days to a `LocalTime` or hours to a `LocalDate`. Thus, while you may be led to believe that the following code will print `"2018-01-01"` (if you manage to avoid the first trap) but the fact is that it will not compile because `LocalDate` doesn't even have methods that alter time related fields.

   ```
   LocalDate ld = LocalDate.of(2018, 1, 1);
   ld.plusHours(10); // will not compile
   System.out.println(ld);
   ```

4. **Watch out for chaining of static methods of Period** - This one is better explained through code. What do you think the following code will print?

```
LocalDate ld = LocalDate.of(2018, 1, 31).of(2019, 1, 1);
System.out.print(ld+" ");
Period p1 = Period.ZERO;
Period p2 = p1.ofDays(1).ofMonths(1);
System.out.println(p2);
```

It prints - `2019-01-01 P1M`. From the output, it looks like the call to `ofDays(1)` on `p1` had no effect!
Actually, the call to `of(2018, 1, 31)` on `LocalDate` had no effect either but the result produced in case of a `Period` is more alarming.
In both the cases, the **ofXXX** methods are **static** methods and produce a new object from scratch. Unlike the instance methods (i.e. with, plus, minus,), the static methods (of and parse) don't build "on top of" a previous object. As you know, it is technically valid to invoke a static method on a reference variable but invoking a static method on a reference is the same as invoking a static method on the class name. Thus, `LocalDate.of(2018, 1, 31)` produces a new `LocalDate` instance and chaining another `of` method to this `LocalDate` instance is actually the same as invoking `LocalDate.of(2019, 1, 1)` independently. However, since both the calls provide values for all the fields of a `LocalDate`, there is no difference in the end result.

Similarly, in case of `Period`, invoking `ofMonths(1)` on an existing `Period` instance is the same as invoking `Period.ofMonths(1)`. However, since `ofMonths` method creates a Period with 0 years and 0 days, it seems as if this call lost the value of the days property that was set earlier through `Period.ofDays(1)`.

In fact, none of the static methods make use of the values of an existing date/time instance even if you invoke the static method using a reference instead of the class name. If you want to create a Period with 1 month and 1 day, you should use `Period.of(0, 1, 1);`.

12.4 Declare and use an ArrayList of a given type

12.4.1 ArrayList and collections

Processing multiple objects of the same kind in the same way is often a requirement in applications. Printing the names of all the Employees in a list of Employees, computing interest for a list of Accounts, or something as simple as computing the average of a list of numbers, require you do deal with a list of objects instead of one single object. You have already seen a data structure that is capable of holding multiple objects of the same kind - array. An array lets you hold references to multiple objects of a type and pass them around as a bunch. However, arrays have a couple of limitations. First, an array cannot grow or shrink in length after it is created. If you create an array of 10 Employees and later find out you have 11 Employees to hold, you will need to create a new array with 11 slots. You can't just add one more slot in an existing array. Second, inserting a value in the middle of an array is a bit difficult. If you want to insert an element just before the last

element, you will have to first shift the last element to the next position and then put the value. Imagine if you want to insert an element in the middle of the list!

Although both of the limitations can be overcome by writing some extra code, these requirements are so common that writing the same code over and over again is just not a good idea. `java.util.ArrayList` is a class that incorporates these, and several other, features out of the box. The following is an example that shows how easy it is to manage a list of objects using an ArrayList:

```
import java.util.ArrayList;
public class TestClass{
  public static void main(String[] args){
      ArrayList al = new ArrayList();

      al.add("alice"); //[alice]
      al.add("bob"); //[alice, bob]
      al.add("charlie"); //[alice, bob, charlie]

      al.add(2, "david"); //[alice, bob, david, charlie]

      al.remove(0); //[bod, david, charlie]

      for(Object o : al){  //process objects in the list
         String name = (String) o;
         System.out.println(name+" "+name.length());
      }

      //dump contents of the list
      System.out.println("All names: "+al);
  }
}
```

You will get a few warning messages when you compile the above program. Ignore them for now. It prints the following output when run:

```
bob 3
david 5
charlie 7
All names: [bob, david, charlie]
```

The above program illustrates the basics of using an `ArrayList`, i.e., adding objects, removing, iterating over them, and printing the contents of the `ArrayList`. I suggest you write a program that does the same thing using an array of strings instead of an `ArrayList`. This will give you an appreciation for the value that `ArrayList` adds over a raw array.

Before we move forward, I need to talk about a few points that are not on the OCA exam but are related to `ArrayList` and are important anyway.

Collections and Collections API

`ArrayList` belongs to a category of classes called **"collections"**. A **"collection"** is nothing but a group of objects held up together in some form. You can visualize a collection as a bag in which you put several things. Just like you use a bag to take various grocery items from a store to your home, you put the objects in a collection so that you can pass them around to other objects and methods. A bag knows nothing about the items themselves. It merely holds the items. You can add items to it and take items out of it. It doesn't care about the order in which you put the items or whether there are any duplicates or how they are stored inside the bag. The behavior of a bag is captured by an interface called `Collection`, which exists in `java.util` package.
Now, what if you want a collection that ensures objects can be retrieved from it in the same order in which they were added to it? Or what if you want a collection that does not allow duplicate objects? You can think of several features that can be added on top of a basic collection. As you have already learnt in "Working with Inheritance- I and II", subclassing/subinterfacing is the way you extend the behavior of an existing class or an interface. the Java standard library takes the same route here and extends `java.util.Collection` interface to provide several interfaces with various features. In fact, the Java standard library provides a huge tree of interfaces along with classes that implement those interfaces. These classes and interfaces are collectively known as the **"Collections API"**.
So what has `ArrayList` got to do with this, you ask? Well, `ArrayList` is a class that implements one of the specialized collection interfaces called `List`. `java.util.List` **extends** `java.util.Collection` to add the behavior of ordering on top of a simple collection. A `List` is supposed to remember the order in which objects are added to the collection. Okay, so that takes care of the "List" part of ArrayList, what about the "Array" part? Again, as you have learnt in previous chapters, an interface merely describes a behavior. It doesn't really implement that behavior. You need a class to implement the behavior. In this case, `ArrayList` is the class that implements the behavior described by `java.util.List` and to implement this behavior, it uses an array. Hence the name **ArrayList**. Remember I talked about writing code to overcome the limitation of an array that it is not flexible enough to increase in size automatically? `ArrayList` contains that code. When you add an object to an `ArrayList`, it checks whether there is space left in the array. If not, it allocates a new array with a bigger length, copies the elements from the old array to this new array, and then adds the passed object to the new array. Similarly, it contains code for inserting an object in the middle of the array. All this is transparent to the user of an `ArrayList`. A user simply invokes methods such as add and remove on an `ArrayList` without having any knowledge of the array that is being used inside to store the object.

Note

In case you are wondering whether another way to implement a List that does not use an array exists, then yes. There is. `java.util.LinkedList`. But again, this is way beyond the scope of the exam, which is why I will not cover it here. I suggest you explore the various classes and interfaces of the Collections API if you are going to face technical interviews.

Generics

Now, regarding the warning messages the I asked you to ignore. Observe the `for` loop in the above code. The type of the loop variable `o` is `Object` (and not `String`). To invoke the `length` method of `String` on `o`, we need to cast `o` to `String`. This is because, in the same way that a bag is unaware of what is inside of it, so too does the `ArrayList` object have no knowledge about the type of the objects that have been added to it. It is the responsibility of the programmer to cast the objects that they retrieve from the list to appropriate types. In this program, we know that we have added `String`s to this list and so we know that we can cast the object that we retrieve from this list to `String`. But what if we were simply given a pre-populated list as an argument? Since the list would have been populated by another class written by someone else, what guarantee would we have about the type of objects we find in the list? None, actually. And as you are aware, if we try to cast an object to an inappropriate type, we get a `ClassCastException` at run time. Getting a `ClassCastException` at run time would be a coding mistake that will be discovered only at run time. Discovering coding mistakes at run time is not a good thing and the compiler is only trying to warn us about this potential issue by generating warning messages while compiling our code.

The first warning message that it prints out is, "`Warning: unchecked call to add(E) as a member of the raw type java.util.ArrayList`" for the code `al.add("alice");` at line 8. It prints the same warning every time an object is added to the list. The compiler is trying to tell us that it does not know what kind of objects this `ArrayList` is supposed to hold and that it has no way of checking what we are adding to this list. By printing out the warning, the compiler is telling us that it will not be held responsible for the objects that are being put in this list. Whoever wants to retrieve objects from this list must already know what type of objects are present inside the list and must cast those objects appropriately upon retrieval at their own risk. In other words, this list is basically **"type unsafe"**. It is unsafe because it depends on assumptions made by humans about the contents of the list. This assumption is not checked or validated by the compiler for its truthfulness. One can put any kind of object in the list and others will not realize until they are hit with a `ClassCastException` at run time when they try to cast the retrieved object to its expected type.

Java solves this problem by allowing us to specify the type of objects that we are going to store in a list while declaring the type of the list. If, instead of writing `ArrayList al = new ArrayList();`, we write `ArrayList<String> al = new ArrayList<String>();` all the warning messages go away. The compiler now knows that the `ArrayList` pointed to by `al` is supposed to hold only `String`s. Therefore, it will be able to keep a watch on the type of objects the code tries to add this list. If it sees any code that attempts to add anything that is not a `String` to the list, it will refuse to compile the code. The error message generated by the following code illustrates this point:

```
ArrayList<String> al = new ArrayList<String>();
al.add(new Object());
```

The error message is:

```
Error: no suitable method found for add(java.lang.Object)
    method java.util.Collection.add(java.lang.String) is not applicable
```

The compiler will not let you corrupt the list by adding objects that this list is not supposed to keep. Similarly, the compiler will not let you make incorrect assignments either. Here is an example:

```
List<String> al = new ArrayList<String>(); //al points to a List of Strings
al.add("hello"); //valid
String s = al.get(0); //valid, no cast required
Object obj = al.get(0); //valid because a String is-a Object
Integer in = al.get(0); //Invalid. Will not compile
```

The error message is:

```
Error: incompatible types: java.lang.String cannot be converted to java.lang.Integer
```

Observe that we supplied information about the type of object the list is supposed to keep by appending <String> to the variable declaration and the ArrayList creation. This way of specifying the type information is a part of a feature introduced in Java 5 known as "generics". Although Generics is a huge topic that deserves a chapter or two of its own, the OCAJP exam covers a very small aspect of this topic.
For the purpose of the exam, you need to know just the following points:

1. A type safe ArrayList or List variable can be declared by specifying the type within < and >. For example,

   ```
   ArrayList<String> stringArray;
   List<Integer> iList;
   ArrayList<com.acme.Foo> fooList;
   ```

2. A type safe ArrayList object can be instantiated similarly by specifying the type within < and >. For example,

   ```
   new ArrayList<String>();
   new ArrayList<Integer>();
   new ArrayList<com.acme.Foo>();
   ```

3. If you are instantiating a type safe ArrayList while assigning it to a type safe variable in the same statement, then you may omit the type name from the instantiation as shown below:

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

   Upon encountering <>, the compiler infers the type of the ArrayList object from the type of the variable. The <> operator is known as the "diamond operator". It can be used only in a new statement. Thus, the statement ArrayList<>al; will not compile but new ArrayList<>(); will. This operator was introduced in Java 7 with the sole purpose of saving you from typing a few extra keystrokes.

12.4.2 ArrayList API

ArrayList has three constructors.

1. `ArrayList()`: Constructs an empty list with an initial capacity of 10. Just like you saw with the `StringBuilder` class, capacity is simply the size of the initial array that is used to store the objects. As soon as you add the 11th object, a new array with bigger capacity will be allocated and all the existing objects will be transferred to the new array.

2. `ArrayList(Collection c)`: Constructs a list containing the elements of the specified collection.

3. `ArrayList(int initialCapacity)`: Constructs an empty list with the specified initial capacity. This constructor is helpful when you know the approximate number of objects that you want to add to the list. Specifying an initial capacity that is greater than the number of objects that the list will hold improves performance by avoiding the need to allocate a new array every time it uses up its existing capacity.

   It is possible to increase the capacity of an `ArrayList` even after it has been created by invoking `ensureCapacity(int n)` on that `ArrayList` instance. Calling this method with an appropriate number before inserting a large number of elements in the `ArrayList` improves performance of the add operation by reducing the need for incremental reallocation of the internal array. The opposite of `ensureCapacity` is the `trimToSize()` method, which gets rid of all the unused space by reducing its capacity to the match the number of elements in the list.

Here are a few declarations that you may see on the exam:

```
List<String> list = new ArrayList<>(); //ok because ArrayList implements List

List<Integer> al = new ArrayList<Integer>(50); //initial capacity is 50

ArrayList<String> al2 = new ArrayList<>(list); //copying an existing list
List list2 = new List(list); //will not compile because List is an interface, it cannot
    be instantiated
```

Besides using a constructor to create an `ArrayList`, there is a utility class named `Arrays` in `java.util` package that has a very convenient `asList` method. This method converts an array into a `List` and is used very often. For example:

```
List<String> strList = Arrays.asList( new String[]{ "a", "b", "c" } );

List<Integer> al = Arrays.asList( new Integer[]{1, 2, 3 } );
```

Observe that the type of the variables in the above code is `List` and not `ArrayList`. This is because the return type of `asList` is `List`. Although you will not be tested on this method in the exam, you may see code that uses this method to build an `ArrayList` with a few elements in a single statement.

Important methods of ArrayList

`ArrayList` has quite a lot of methods. Since `ArrayList` implements `List` (which, in turn, extends `Collection`), several of `ArrayList`'s methods are declared in `List`/`Collection`. The exam does

not expect you to make a distinction between the methods inherited from `List`/`Collection` and the methods declared in `ArrayList`.

The following are the ones that you need to know for the exam:

1. `String toString()`: Well, `toString` is not really the most important method of `ArrayList` but since we will be depending on its output in our examples, it is better to get it out of the way early. `ArrayList`'s `toString` first gets a string representation for each of its elements (by invoking `toString` on them) and then combines into a single string that starts with `[` and ends with `]`. For example, the following code prints `[a, b, c]`:

```
ArrayList al = new ArrayList();
al.add("a");
al.add("b");
al.add("c");
System.out.println(al);
```

   Observe the order of the elements in the output. It is the same as the order in which they were added in the list. Calling `toString` on an empty `ArrayList` gets you `[ ]`. I will use the same format to show the contents of a list in code samples.

Methods that add elements to an `ArrayList`:

1. `boolean add(E e)`: Appends the specified element to the end of this list. Don't worry about the type of the method parameter `E`. It is just a place holder for whichever type you specify while creating the `ArrayList`. For example, `E` stands for `String`, if you create an `ArrayList` of Strings, i.e., `ArrayList<String>`.
   This method is actually declared in `Collection` interface and the return value is used to convey whether the collection was changed as a result of calling this method. In case of an `ArrayList`, the `add` method always adds the given element to the list, which means it changes the collection every time it is invoked. Therefore, it always returns `true`.

2. `void add(int index, E element)`: Inserts the specified element at the specified position in this list. The indexing starts from `0`. Therefore, if you call `add(0, "hello")` on an list of `Strings`, `"hello"` will be inserted at the first position.

3. `boolean addAll(Collection<? extends E>c)`: Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator. Again, for the purpose of the OCAJP exam, you don't need to worry about the `"? extends E"` or the Iterator part. You just need to know that you can add all the elements of one list to another list using this method. For example:

```
ArrayList<String> sList1 = new ArrayList<>();
sList1.add("a"); //[a]
ArrayList<String> sList2 = new ArrayList<>();
sList2.add("b"); //[b]
sList2.addAll(sList1); //sList2 now contains [b, a]
```

4. `boolean addAll(int index, Collection<? extends E>c)`: This method is similar to the one above except that it inserts the elements of the passed list in the specified collection into this list, starting at the specified position. For example:

```
ArrayList<String> sList1 = new ArrayList<>();
sList1.add("a"); //[a]
ArrayList<String> sList3 = new ArrayList<>();
sList3.add("b"); //[b]
sList3.addAll(0, sList1); //sList3 now contains [a, b]
```

Methods that remove elements from an `ArrayList`:

1. `E remove(int index)`: Removes the element at the specified position in this list. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, c]
String s = list.remove(1); //list now has [a, c]
```

   It returns the element that has been removed from the list. Therefore, `s` will be assigned the element that was removed, i.e., `"b"`. If you pass an invalid int value as an argument (such as a negative value or a value that is beyond the range of the list), an `IndexOutOfBoundsException` will be thrown.

2. `boolean remove(Object o)`: Removes the first occurrence of the specified element from this list, if it is present. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, a]
list.remove("a"); //[b, a]
```

   Observe that only the first `a` is removed.
   You have to pay attention while using this method on an `ArrayList` of `Integers`. Can you guess what the following code will print?

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 }
    ));
list.remove(1);
System.out.println(list);
list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 } ));
list.remove(new Integer(1));
System.out.println(list);
```

   The output is:

```
[1, 3]
[2, 3]
```

   Recall the rules of method selection In case of overloaded methods. When you call `remove(1)`, the argument is an `int` and since a remove method with `int` parameter is available, this

method will be preferred over the other remove method with `Object` parameter because invoking the other method requires boxing `1` into an `Integer`.

This method returns `true` if an element was actually removed from the list as a result of this call. In other words, if there is no element in the list that matches the argument, the method will return `false`.

3. `boolean removeAll(Collection<?>c)`: Removes from this list all of its elements that are contained in the specified collection. For example the following code prints `[ c ]`:

```
ArrayList<String> al1 = new ArrayList<>(Arrays.asList( new String[]{"a", "b",
    "c", "a" } ));
ArrayList<String> al2 = new ArrayList<>(Arrays.asList( new String[]{"a", "b" } ));
al1.removeAll(al2);
System.out.println(al1); //prints [ c ]
```

Observe that unlike the `remove(Object obj)` method, which removes only the first occurrence of an element, `removeAll` removes all occurrences of an element.

This method returns `true` if an element was actually removed from the list as a result of this call.

4. `void clear()`: Removes all of the elements from this list.

Methods that replace an element in an `ArrayList`:

1. `E set(int index, E element)`: Replaces the element at the specified position in this list with the specified element. It returns the element that was replaced.

Example:

```
ArrayList<String> al = ... // create a list containing [a, b, c]
String oldVal = al.set(1, "x");
System.out.println(al); //prints [a, x, c]
System.out.println(oldVal); //prints b
```

Methods that read an `ArrayList` without modifying it:

1. `boolean contains(Object o)`: The object passed in the argument is compared with each element in the list using the equals method. A `true` is returned as soon as a matches is found, a `false` is returned otherwise. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c" } ));
System.out.println(al.contains("c")); //prints true
System.out.println(al.contains("z")); //prints false
System.out.println(al.contains(null)); //prints true
```

Observe that it does not throw a `NullPointerException` even if you pass it a `null`. In fact, a `null` argument matches a `null` element.

2. `E get(int index)`: Returns the element at the specified position in this list. It throws an `IndexOutOfBoundsException` if an invalid value is passed as an argument.

3. `int indexOf(Object o)`: The object passed in the argument is compared with each element in the list using the equals method. The index of the first element that matches is returned. If this list does not contain a matching element, -1 is returned. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c", null } ));
System.out.println(al.indexOf("c")); //prints 3
System.out.println(al.indexOf("z")); //prints -1
System.out.println(al.indexOf(null)); //prints 1
```

Observe that just like `contains`, `indexOf` does not throw a `NullPointerException` either even if you pass it a `null`. A `null` argument matches a `null` element.

4. `boolean isEmpty()`: Returns `true` if this list contains no elements.

5. `int size()`: Returns the number of elements in this list. Recall that to get the number of elements in a simple array, you use the variable named `length` of that array.

6. `List<E>subList(int fromIndex, int toIndex)` Note: It is not clear whether this method is included in the exam or not. I haven't heard people getting a question on this method recently. It is good to have a basic idea about it though. This method returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive. If `fromIndex` and `toIndex` are equal, the returned list is empty.

The returned list is backed by this list, which means any change that you make in the returned list is visible in the this list as well. Here is an example:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", "b", "c", "d", "e" } ));
List<String> al2 = al.subList(2, 4);
System.out.println(al2); //prints [c, d]
al2.add("x");
System.out.println(al2); //prints [c, d, x]
System.out.println(al); //prints [a, b, c, d, x, e]
```

Observe the output of the last two lines. `x` was added to the end of the list pointed to by `al2`. However, since `al2` is just a view of a portion of the original list pointed to by `al`, `x` is visible in the middle of this list.

The examples that I have given above are meant to illustrate only a single method. In the exam, however, you will see code that uses multiple methods. Here are a few points that you should remember for the exam:

1. **Adding nulls**: `ArrayList` supports `null` elements.
2. **Adding duplicates**: `ArrayList` supports duplicate elements.
3. **Exceptions**: None of the `ArrayList` methods throw `NullPointerException`. They throw `IndexOutOfBoundsException` if you try to access an element beyond the range of the list.
4. **Method chaining**: Unlike `StringBuilder`, none of the `ArrayList` methods return a reference to the same `ArrayList` object. Therefore, it is not possible to chain method calls.

Here are a few examples of the kind of code you will see in the exam. Try to determine the output of the following code snippets when they are compiled and executed:

1. —

```
ArrayList<String> al = new ArrayList<>();
al.add("a").add("b");
System.out.println(al);
```

2. —

```
ArrayList<String> al = new ArrayList<>();
if( al.add("a") ){
  if( al.contains("a") ){
    al.add(al.indexOf("a"), "b");
  }
}
System.out.println(al);
```

3. —

```
ArrayList<String> al = new ArrayList<>();
al.add("a"); al.add("b");
al.add(al.size(), "x");
System.out.println(al);
```

4. —

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.remove("b");
System.out.println(list1);
```

5. —

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
System.out.println(list1);
list1.remove("b");
System.out.println(list1);
```

6. —

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.removeAll("b");
System.out.println(list1);
```

12.4.3 ArrayList vs array

You may get a few theoretical questions in the exam about the advantages and disadvantages of an ArrayList over an array. You have already seen all that we can do with ArrayLists and arrays, so, I am just going to summarize their advantages and disadvantages here.

Advantages of ArrayList

1. **Dynamic sizing** - An ArrayList can grow in size as required. The programmer doesn't have to worry about the length of the ArrayList while adding elements to it.
2. **Type safety** - An ArrayList can be made type safe using generics.
3. **Readymade features** - ArrayList provides methods for searching and for inserting elements anywhere in the list.

Disadvantages of ArrayList

1. **Higher memory usage** - An ArrayList generally requires more space than is necessary to hold the same number of elements in an array.
2. **No type safety** - Without generics, an ArrayList is not type safe at all.
3. **No support for primitive values** - ArrayLists cannot store primitive values while arrays can. This disadvantage has been mitigated somewhat with the introduction of autoboxing in Java 5, which makes it is possible to pass primitive values to various methods of an ArrayList. However, autoboxing does impact performance.

Similarities between ArrayLists and arrays

1. **Ordering** - Both maintain the order of their elements.
2. **Duplicates** - Both allow duplicate elements to be stored.
3. **nulls** - Both allows nulls to be stored.
4. **Performance** - Since an ArrayList is backed by an array internally, there is no difference in performance of various operations such as searching on an ArrayList and on an array.
5. **Thread safety** - Neither of them are thread safe. Don't worry, thread safety is not on the exam, but you should be aware of the fact that accessing either of them from multiple threads, may cause produce incorrect results in certain situations.

12.5 Write a simple Lambda expression

12.5.1 Lambda Expressions

While taking training sessions, I have observed that many Java beginners feel intimidated by lambda expressions. Most of the time this fear is because of the name "lambda". The word lambda itself does not create any useful mental picture and that makes it a bit difficult to relate to. On top of that, most books and tutorials start explaining lambda expression as a way to do "functional" programming, which is another buzz word that is hard to relate to when a person is just learning "regular" programming!

Well, I can tell you that there is no need to be afraid. Lambda Expressions are actually very simple. While writing test programs, have you ever been frustrated while typing the words "public static void main(String[] args)" over and over again? I am sure you have been. If you are using an IDE such as NetBeans or Eclipse, then you are probably aware of shortcuts such as `psvm`. You type `psvm`, hit the tab key and the IDE replaces psvm with the text "public static void main(String[] args)" immediately. A lambda expression is just like that but for the compiler. Think of a lambda expression as a shortcut for the compiler that does two things - defines a class with a method and instantiates that class. As soon as the compiler sees a lambda expression, it expands the expression into a class definition and a statement that instantiates that class. If you think of it as a tool that saves you from typing a lot of keystrokes, you will start loving it. You will love it so much that you will look for opportunities to use it as much as possible. Let me show you how cool it is.

Imagine that you are working on an application for a car shop. You have a list of cars and you need to filter that list by various criteria. You may need to filter it by company, you may need to filter it by price, or by any other property that the users may want. The following code shows how one might do it. You should actually copy all of the following code in a single file named `TestClass.java` so that you can try it out. Just add `import java.util.*;` at the top.

```
class Car{
    String company; int year; double price; String type;
    Car(String c, int y, double p, String t){
        this.company = c; this.year = y;
        this.price = p; this.type = t;
```

```
    }
    public String toString(){  return "("+company+" "+year+")"; }
}
```

The `Car` class represents a car with a few properties. Agreed, `Car` is not well encapsulated. Ideally, `Car` should have had private fields and public accessors. But since encapsulation is not relevant here, I haven't shown these methods to conserve space.

```
class CarMall {
    List<Car> cars =  new ArrayList<>();

    CarMall(){
        cars.add(new Car("Honda", 2012, 9000.0, "HATCH"));
        cars.add(new Car("Honda", 2018, 17000.0, "SEDAN"));
        cars.add(new Car("Toyota", 2014, 19000.0, "SUV"));
        cars.add(new Car("Ford", 2014, 13000.0, "SPORTS"));
        cars.add(new Car("Nissan", 2017, 8000.0, "SUV"));
    }


    List<Car> showCars(CarFilter cf){
        ArrayList<Car> carsToShow = new ArrayList<>();
        for(Car c : cars){
            if(cf.showCar(c)) carsToShow.add(c);
        }
        return carsToShow;
    }
}

interface CarFilter{
    boolean showCar(Car c);
}
```

`CarMall` represents the shop. It creates a list of a few `Car` objects. This list contains details of all the cars that the shop has. It has a `showCars` method that returns a list of cars based on any given criteria. Instead of specifying the actual criteria for filtering the cars inside the showCars method, it uses a `CarFilter` instance to determine whether a car needs to be listed or not. The `CarFilter` interface declares just the basic structure of a filter. Note that it is not really possible to code the actual filtering criteria inside the showCars method because the criterion is determined by the user of the `CarMall` class. By accepting an interface as an argument, the showCars method lets the caller decide the criterion.

```
public class TestClass{
    public static void main(String[] args) {
        CarMall cm = new CarMall();
        CarFilter cf = new CompanyFilter("Honda");
        List<Car> carsByCompany = cm.showCars(cf);
        System.out.println(carsByCompany);
```

```
    }
}
class CompanyFilter implements CarFilter{
    private String company;
    public CompanyFilter(String c){
        this.company = c;
    }
    public boolean showCar(Car c){
        return company.equals(c.company);
    }
}
```

`TestClass` represents a third party class that uses `CarMall`. It wants to get the details of all cars from a particular company, say, `Honda`. To do that, it defines a `CompanyFilter` class that contains the actual logic for filtering cars based on company name. At run time, it creates a `CompanyFilter` object and passes it to `CarMall`'s `showCars` method, which returns a filtered list of cars.

Now look at the following code for TestClass that uses a lambda expression:

```
public class TestClass{
    public static void main(String[] args) {
        CarMall cm = new CarMall();
        List<Car> carsByCompany = cm.showCars(c -> c.company.equals("Honda"));
        System.out.println(carsByCompany);
    }
}
```

Observe that there is no separate class that implements `CarFilter` and there is no explicit instantiation of a `CarFilter` object either. Both of these tasks have been replaced by a very short statement - `c ->c.company.equals("Honda")`. That's it. We have actually eliminated 10 lines of code with that change! Go ahead, count them :)
As I said before, the lambda expression used above is just a shortcut for the compiler. The compiler actually expands this expression into a fully-fledged class definition plus code for instantiating an object of that class. Once you understand how the compiler is able to do this expansion, lambda expressions will seem like a piece of cake to you.

In fact, all the information that is required to do the expansion is available in the context of `cm.showCars(...)` method call already. The compiler knows that it must pass an object of a class that implements `CarFilter` to the `showCars(CarFilter cf)` method. It knows that this class must implement the `boolean showCar(Car c)` method because that is the only abstract method declared in the `CarFilter` interface. The compiler gathers from the signature of this method that this method receives an argument of type `Car` and returns a `boolean`. From this information, it can easily generate the following code on its own:

```
//This class is created by the compiler. It can give any random name to the class here!
class XYZ implements CarFilter{
   public boolean showCar(Car <<parameterName>>){
      return <<an expression that returns a boolean must appear here>>;
```

```
    }
}
```

The only thing that the compiler cannot generate on its own is the boolean expression that goes inside the `showCar` method. It is not a coincidence that that is exactly what our lambda expression `c ->c.company.equals("Honda")` contains. The compiler simply takes the variable name from the left-hand side of `->` of the lambda expression, i.e., `c`, plugs in the expression given on the right-hand side of `->` i.e. `c.company.equals("Honda")`, into the body of the above method and it has the complete code for a class that implements `CarFilter`! Finally, it throws in an instantiation expression `new XYZ();` as a bonus! These are the same two things that are needed to invoke `cm.showCars(CarFilter cf)` method, i.e., code for a class that implements `CarFilter` and an expression that instantiates an object of that class!

I suggest you go through the above discussion a couple of times to absorb the steps that the compiler takes for expanding a lambda expression into a fully-fledged class before moving forward. As an exercise, try to expand the lambda expression `x ->x.price>10000` into a class that implements `CarFilter`.
Observe that the lambda expression does not specify the method name, the parameter types, and the return type. The compiler infers these three pieces of information from the context in which you put the lambda expression. Thus, a lambda expression must exist within a context that can supply all this information. Furthermore, you know that in Java, a method cannot exist on its own. It can exist only as a member of a class (or an interface, for that matter). This implies that the method generated by the compiler for a lambda expression cannot exist on it own either. The compiler must create a class as well to contain the method. Since a lambda expression has no name, the class must be such that it requires implementation of exactly one abstract method whose name the compiler can use for the generated the method. The only way this can happen is if the class generated by the compiler implements an interface with exactly one abstract method or extends an abstract class with exactly one abstract method. If the interface or the class has more than one abstract method or no abstract method, the compiler wouldn't know which method the code in the lambda expression belongs to.

Java language designers decided not to allow lambda expressions for abstract classes to reduce complexity. Thus, the only question remaining is which interface should the generated class implement. That depends on the context. The type that the context expects is described as the **"target type"** and is the interface that the class generated by the compiler must implement. In the above code, the `showCars` method expects a `CarFilter` object (i.e. object of a class that implements `CarFilter`). Therefore, the generated class must implement `CarFilter`. In technical terms, `CarFilter` is the target type of the lambda expression that is passed as an argument to `showCars` method.

You may wonder at this point why the compiler can't provide a made up name to the generated method just like it did for the generated class. Well, it could provide a made up name to the method but what would that achieve? Since the programmer wouldn't know that made up name, how would they call this method? Remember that the programmer doesn't need to know the name of the class because the compiler passes an object of this class and the programmer

knows the name of the parent class or the interface. You have learnt in the Polymorphism section that a reference of type parent class/interface can be used to invoke a method on a subclass. That is exactly what is happening here. The receiver of the lambda expression doesn't care about the type of the actual object that it receives because the receiving code invokes a method on that object using an interface reference. Take a look at the `showCars` method of `CarMall` again. It uses a reference of type `CarFilter` to invoke the `showCar` method. It doesn't matter to this code what name the compiler gives to the generated class. This code only cares about the name of the method.

Functional Interfaces

From the above discussion, it should be clear that a lambda expression can be written only where the target type is an interface with exactly one abstract method. Java has a special name for such an interface: **Functional Interface**. Java standard library includes several functional interfaces in `java.util.function` package. These functional interfaces capture functionality that is required by Java applications very often and are also used heavily by other Java library classes. Understanding these functional interfaces is required for the OCPJP exam but for the OCAJP exam, you need to worry about only `java.util.function.Predicate` interface. I will get to it but first, you need to know the various ways in which you can write a lambda expression.

12.5.2 Parts of a Lambda expression

You have seen that a lambda expression is basically code for a method in compact form. It has two parts separated by the "arrow" operator, i.e., ->. The left side is for variable declarations and the right side is for the code that you want executed. Just like a method, a lambda expression also can have any number of parameters and can return (or not return) a value. Unfortunately, Java allows lambda expressions to be written in several different ways. The reason for having so many ways is to cut out as much redundant code as possible. The OCAJP exam expects you to know all these ways. You will be asked to identify valid and invalid lambda expressions in the exam. From this perspective, I have categorized the variations into two categories - variations on the parameter section and variations on the body section.

There are three possibilities for the parameters section:

1. **No parameter** - If a lambda expression takes no parameters, the parameter part of the expression must have an empty set of brackets, i.e., ( ). For example,
   ```
   () ->true //valid
   ->1 //invalid, missing variable declaration part
   ```

2. **One parameter** - If a lambda expression takes exactly one parameter, the parameter name may be specified within brackets, i.e., (pName) or without the brackets, i.e., pName. For example,

   ```
   a -> a*a //valid
   (a) -> a*a //valid
   ```

You may also include the parameter type for the parameter name if you want but then you will need to use brackets. For example:

```
(int a) -> a*a //valid
int a -> a*a //invalid
```

3. **More than one parameters** - If a lambda expression takes more than one parameter, all the parameter names must be specified within the brackets, i.e., (pName1, pName2, pName3). For example,

```
(a, b, c) -> a + b + c //valid
a, b -> a+b //invalid, parameters must be within ( )
```

Again, parameter types are optional. For example. `(int a, int b, int c) ->a + b + c //valid`
If you are specifying parameter types, you must specify them for all the parameters. Thus, `(int a, int b, c) ->a + b + c` would be invalid because it does not specify the type of `c`

The syntax of the code part of a lambda expression is simple. It can either be an expression or a block of code contained within curly braces. Given that the body may or may not return a value, there are four possibilities:

1. **Expression with or without a return value** - This is the most common use case and is therefore, the smallest. You can simply put an expression on the right side of `->`. If the expression has a return value and if the lambda expression is supposed to return a value, the compiler will figure out that the value generated by the expression is to be returned and will insert a return statement on it own. You must not write the return keyword. For example,

```
a ->a + 2 //valid
a - >return a + 2 //invalid, must not have return keyword
```

Similarly, an expression that doesn't return any value can also be used directly as the body of the lambda expression. For example,

```
(a, b) -> System.out.println(a+b)
 //method call is a valid expression
```

2. **Block of code with or without a return value** - If you have multiple lines of code, you must write them within curly braces, i.e., { }. If the expression is supposed to return a value, you must use a return statement to return the desired value.

This is pretty much the same as writing a method body with or without a return value. You can use this syntax even if you have just one statement in the body. For example, here is Lambda expression that returns a value:

```
(a) -> {
          int x = 2;
          int y = x+a;
          return y;
       }
```

and here is one that doesn't:

```
() -> {
          int x = 2;
          int y = 3;
          System.out.println(x+y);
       }
```

Observe that unlike lambdas with just an expression as their bodies, the statements within the block end with a semi-colon. This is just like a regular code block. All the rules that apply to code within a method body apply to code within a lambda expression's code block as well. After all, the compiler uses this code block to generate a method body.

The OCAJP exam does not try to trick you with complicated lambda expressions. If you learn the basic rules shown above, you will not have any trouble identifying valid and invalid lambda expressions.

12.5.3 Using Predicate interface

Let's take a look at the `CarFilter` interface that we defined in our `CarMall` example again:

```
interface CarFilter{
    boolean showCar(Car c);
}
```

The whole purpose of this interface is to let you check whether a `Car` satisfies a given criteria so that you could filter a list of Cars. Filtering through a list of objects is such a common requirement in Java application that the Java standard library includes a generic interface for this purpose - `java.util.function.Predicate`. It looks like this:

```
interface Predicate<T>{
  boolean test(T t);
}
```

The `<T>` part means that this interface can be "typed" for any class. You don't have to worry too much about what "typed" means because generics is not on the OCAJP exam. All you need to know is that you can use this interface to work with any class and if the name of that class is `T`, then the method `test` will accept an object of type `T` and return a `boolean`.

So let's change the code for `CarMall`'s `showCars` method to use `Predicate` interface:

```
List<Car> showCars(Predicate<Car> cp){
        ArrayList<Car> carsToShow = new ArrayList<>();
```

```
        for(Car c : cars){
            if(cp.test(c)) carsToShow.add(c);
        }
        return carsToShow;
    }
```

Observe that we have typed `Predicate` to `Car` in the above code. Apart from that, the above code is the same as previous one. But by using the `Predicate` interface instead of writing a custom interface, we have eliminated another three lines of code.

There is no change in the code that calls `showCars` method. The lambda expression that we used earlier, i.e., `cm.showCars(c ->c.company.equals("Honda"))` works for this new method as well. It works because the lambda expression never required us to use the name of any interface or method. Therefore, the lambda expression was not tied to a particular interface or method. It was only tied to a particular behavior -, i.e., to a method that takes `Car` as an argument and returns a `boolean`. We relied on the compiler to produce an appropriate class with an appropriate method. We supplied only the raw code for the method body. Earlier the compiler generated a class that implemented the `CarFilter` interface and it now generates a class that implements the `Predicate<Car>` interface using the same code that we wrote in the lambda expression! In fact, the change is so subtle that if you have both the versions of `showCars` method in `CarMall`, the compiler will reject the line `cm.showCars(c ->c.company.equals("Honda"))` with the error message, "`reference to showCars is ambiguous. Both method showCars(java.util.function.Predicate<Car>) in CarMall and method showCars(CarFilter) in CarMall match`".
I showed only one method in `Predicate` interface but it actually has three `default` methods and one `static` method in addition to the abstract `test` method. I didn't mention them before because they have nothing to do with lambda expressions. You will notice that these methods are basically just helpful utility methods. I haven't ever seen anyone getting any question on these but since `Predicate` is mentioned explicitly in the exam objectives, it is better to be aware of these methods.

1. `default Predicate<T>and(Predicate<? super T>other)` : Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. This is helpful when you have more than one checks to be performed. For example, the following code checks whether a `Car` satisfies two predicates using two separate invocations of `test()`:

```
Predicate<Car> p1 = c -> c.company.equals("Honda");
Predicate<Car> p2 = c -> c.price>(20000.0);

Car c = ...
if(p1.test(c) && p2.test(c)) System.out.println("yes");
```

   Instead of making two calls to `test()`, you could combine the two predicates into one and use only one call to test, like this:

```
Predicate<Car> p3 = p1.and(p2);
Car c = ...
if( p3.test(c) ) System.out.println("yes");
```

As I said before, you do not need to worry about `? super T` part. It relates to generics, which is not on the OCAJP exam.

2. `default Predicate<T>negate()` : Returns a predicate that represents the logical negation of this predicate. For example, if you have `Predicate<Car>p = c ->c.price<20000;` then `Predicate<Car>notP = p.negate();` represents a predicate where `price is >= 20000`.

3. `default Predicate<T>or(Predicate<? super T>other)` : Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. For example, if you have `Predicate<Car>isHonda = c ->c.company.equals("Honda");` and `Predicate<Car>isToyota = c ->c.company.equals("Toyota");` then `Predicate<Car>isHondaOrToyota = isHonda().or(isToyota);` represents a predicate where the company name is `"Honda"` or `"Toyota"`.

4. `static <T>Predicate<T>isEqual(Object targetRef)` : Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`. This method is a way to convert a class's equals method into a `Predicate`. For example, normally, you would compare two Car objects using `c1.equals(c2)`. You could create a `Predicate` out of the equals method like this - `Predicate equals = Predicate.isEqual(c1);` and then compare `c1` with other `Car` objects using this Predicate, i.e., `equals.test(c2)`.

12.5.4 Using Predicate with ArrayList

Generics made Java collections type safe, while Lambda expressions together with functional interfaces gave them super powers. Most of the classes and interfaces of the collections API were updated in Java 8 to include methods that accept functional interfaces so that programmers can invoke those methods with lambda expressions. Things such as iterating, filtering, and replacing elements, that took several lines of code can now be done by half a line of code.

Fortunately or unfortunately, the OCAJP exam only covers `ArrayList` class from the Collection API and only covers the `Predicate` interface from the `java.util.function` package. There is only one method in `ArrayList` that uses `Predicate` interface - `boolean removeIf(Predicate<? super E>filter)`. `ArrayList` inherits this method from the `List` interface, which inherits it from the `Collection` interface. This method removes all of the elements of this list that satisfy the given predicate. You should expect questions on it in the exam. Let's start with the following simple code:

```
ArrayList<Integer> iList = new ArrayList<>();
iList.addAll(Arrays.asList(1, 2, 3, 4, 5, 6));
Predicate<Integer> p = x->x%2==0;
iList.removeIf(p);
System.out.println(iList);
```

The above code prints `[1, 3, 5]`. As you have probably guessed, the lambda expression returns `true` if an element is even. The `removeIf` method executes this lambda expression for each of the elements in the list and removes the element if the expression returns `true` for that element.

Note that we didn't really have to use the variable `p` in the above code. We could have

passed the lambda expression directly to the `removeIf` method, i.e., `iList.removeIf( x->x%2==0 );`

Scope of variables in a lambda expression

There is just one more thing that you need to know about lambda expressions. The variables that you define in the variable section of a lambda expression exist in the same scope as which the lambda expression itself exists. This means, you cannot redefine the variables that already exist in that scope. This is illustrated in the following code:

```
List<String> names = Arrays.asList(new String[]{ "alex", "bob", "casy", "abel"});
for(String n :  names){
     Predicate p = n->n.startsWith("a"); //will not compile
     if(p.test(n))  {
         System.out.println(n);
     }
}
```

The above code will fail to compile with an error message that says, `"variable n is already defined"`. This is because `n` is already in scope within the for loop as well as in the lambda expression. So when the lambda expression tries to define `n`, the compiler complains.

It is possible to access a variable that is in scope of the lambda expression from within the lambda expression's body but only if that variable is declared as final or is "effectively final". Don't be alarmed with the term "effectively final". It just means that even though the variable is not explicitly declared as final, its value is not changed though out the scope in which it exists, and so the compiler assumes it as final. Here is an example:

```
List<String> names = Arrays.asList(new String[]{ "alex", "bob", "casy", "abel"});
int x = 0;
for(String n :  names){
     Predicate<String> p = k->{
         System.out.println(n); //valid
         //System.out.println(x); //will not compile
         return k.startsWith("a");
     };
     if(p.test(n))  {
         System.out.println(n);
     }
}
x = 1; //x is being changed here
```

In the above code, the variable `n` is never changed after it is assigned a value. Thus, `n` is effectively final. But `x` is not because it is assigned a new value later in the code.

12.6 Exercises

1. Create a class that implements the following interface with appropriate implementation for the methods of the interface:

```
interface StringComparator{
   boolean compare(String s, StringBuilder sb);
   boolean compare(int ivalue, String svalue);
   boolean compareSpecial(String s1, String s2); //this method must compare the
    two string arguments while ignoring white spaces at the beginning or at the
    end of the strings. For example, it must true if the arguments are " hello"
    and "hello "
   boolean checkString(String s); //this method must return true if the argument
    contains no character. Show at least three ways to implement this method
}
```

2. Write code to determine whether the `toString` and `substring` methods of `StringBuilder` and `String` classes return an interned string or not. Confirm your results by checking the JavaDoc API descriptions of these methods.

3. Write a method that takes a `String` and returns a `String` of the same length containing the `'X'` character in all positions except the last 4 positions. The characters in the last 4 positions must be the same as in the original string. For example, if the argument is `"12345678"`, the return value should be `"XXXX5678"`.

4. Implement exception handling in the above method such that the method will not end up with an exception if the input string is null or of size less than 4.

5. Implement the same method as above but with a `StringBuilder` as the input parameter.

6. Write a method that takes a `String[]` as an argument and returns a `String` containing the concatenation of all the strings in the input array. Invoke your method with different arguments. Make sure that your code handles the cases where the argument is null, contains a few nulls, or contains only nulls. Is this a good place to make use of a `StringBuilder`?

7. Assuming that system date is `1st July 2018`, create `LocalDate` objects containing the same date using the `of`, `now`, and `parse` methods of `LocalDate`. Print the `LocalDate` objects so, created and observe the printed values.

8. Create a method that takes a `LocalDateTime` as argument and returns a `String` containing just the date (without the time) in ISO format.

9. Pass the `String` returned by the method that you created above to another method that returns a `LocalDate` object representing the same date.

10. Create a method that takes a `List` of `LocalDateTime` objects and returns a `List` of `LocalDate` objects containing only the dates having the same day and month as today.

11. Create a method that takes an array of strings and returns an `ArrayList` containing the same strings.

12. Update the above method to remove the duplicate elements from the `ArrayList` before returning.

13. Create a method with the signature `switch(ArrayList al, int a, int b)`. This method should return the same list but after switching the elements at positions `a` and `b`.

14. Given the following lambda expressions, define appropriate interfaces that can be implemented using these lambda expressions.
    `() ->true`
    `k ->k>5`

15. Given the following interfaces, create lambda expressions that can be used to capture these interfaces.

```
interface Shape{
   double computeArea();
}

interface Operation{
   void operate(String name, double[] params);
}
```

16. Write a method that takes a list of `Image` objects and a `Predicate` as arguments, and returns another list containing only those Images that satisfy the predicate.

17. Assuming that the `Image` class has `width` and `height` properties, invoke the above method that filters out images that are smaller than 100 x 100.

Index

Symbols

A

C

D

E

W

Z