Working with Selected Classes from the Java API

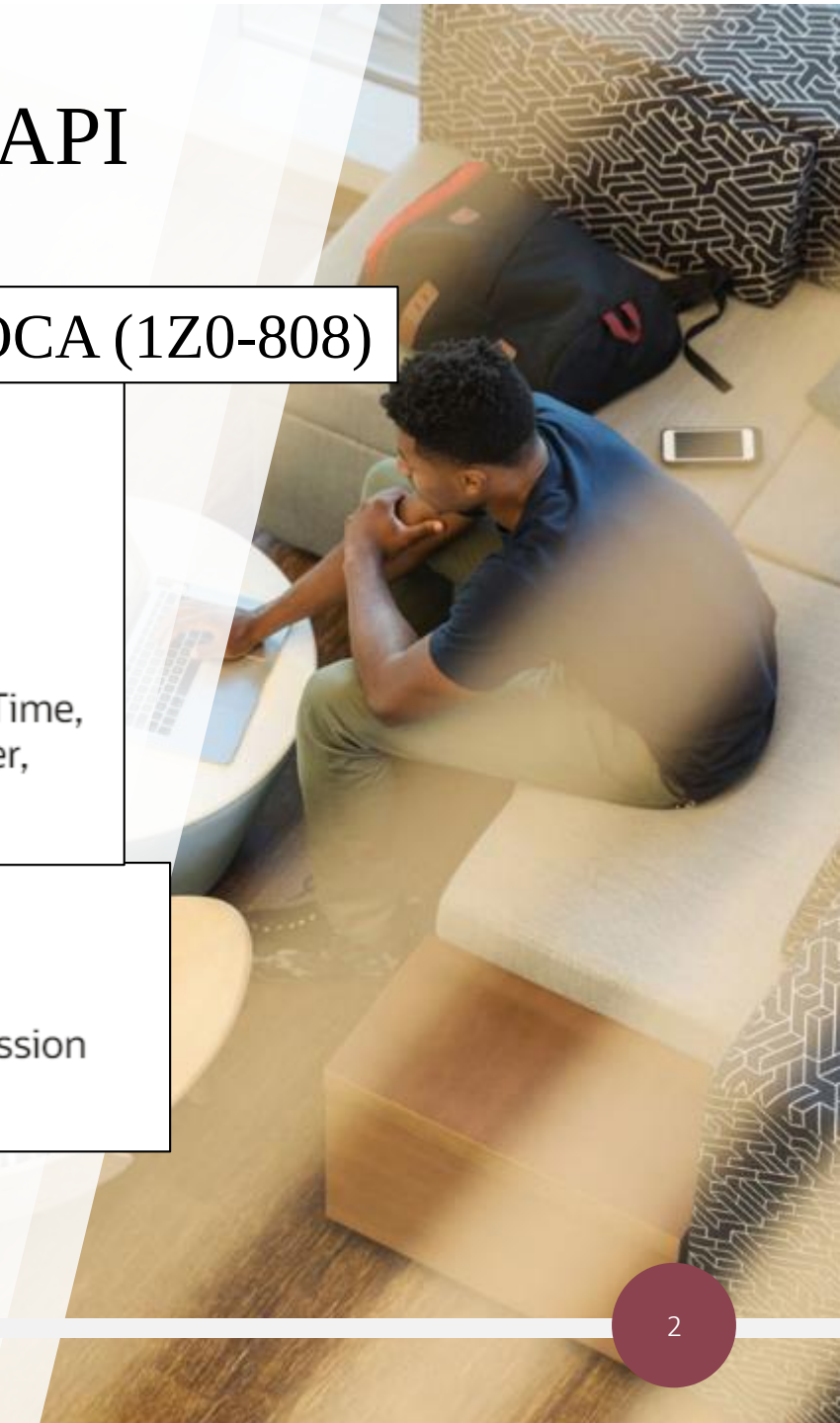
Working with Selected Classes from the Java API

Java 8 OCA (1Z0-808)

Working with Selected classes from the Java API

- ✓ Manipulate data using the StringBuilder class and its methods
- ✓ Create and manipulate Strings
- ✓ Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

- → Declare and use an ArrayList of a given type
- ✓ Write a simple Lambda expression that consumes a Lambda Predicate expression



ArrayList

- An array is a data structure for storing primitives or objects of the same type.

- Arrays have a major limitation:
  - ➢ fixed in size; therefore, you must know the size at compile time

- ArrayLists have no such fixed-in-size limitation and are often referred to as “expandable arrays”.

- However, ArrayLists can not store primitives (they are wrapped in their respective wrapper type).

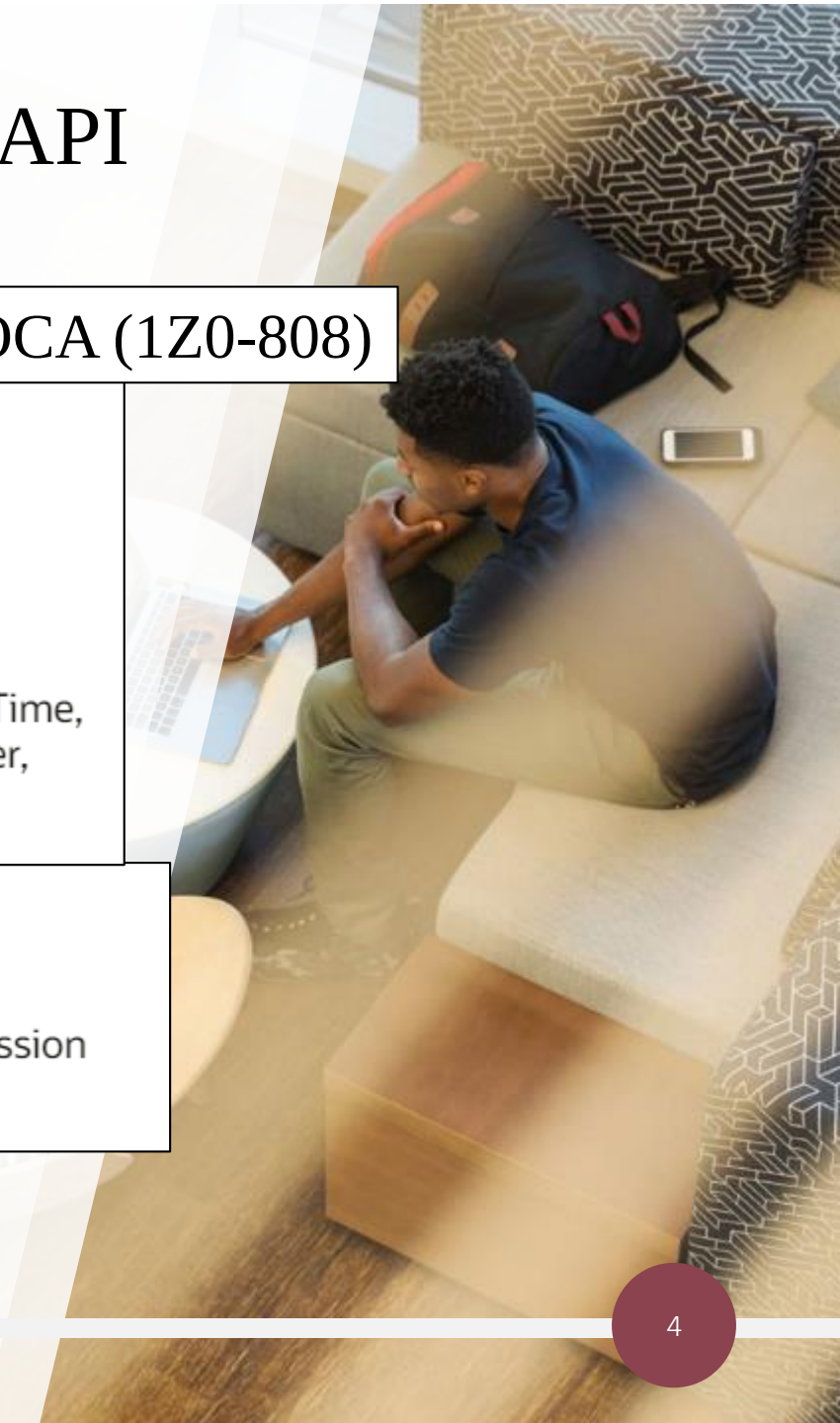
Working with Selected Classes from the Java API

Java 8 OCA (1Z0-808)

Working with Selected classes from the Java API

- ✓ Manipulate data using the StringBuilder class and its methods
- ✓ Create and manipulate Strings
- ✓ Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

- ✓ Declare and use an ArrayList of a given type
- → Write a simple Lambda expression that consumes a Lambda Predicate expression



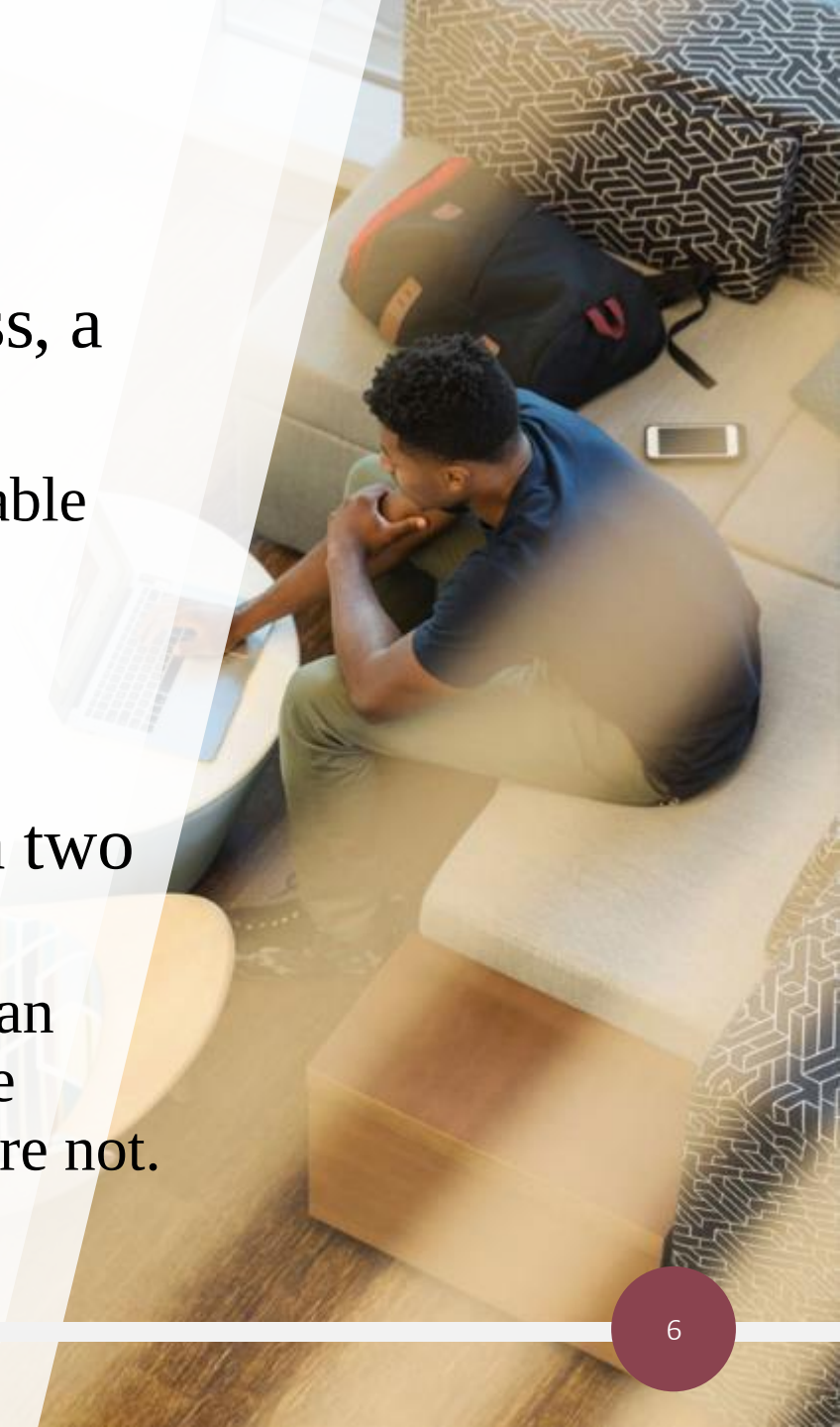
Interfaces

- In general, when you create an interface, you are defining a contract for *what* a class can do; without saying anything about *how* the class will do it.
- A class “signs” the contract with the keyword *implements*.
- When implementing an interface, you are agreeing to adhere (obey) to the contract defined in the interface.
- If a concrete (non-abstract) class is implementing an interface, the compiler will ensure that the class has implementation code for each abstract method in the interface.

Interfaces

- Whereas a class can extend from only one other class, a class can implement many interfaces.
  - class Dog extends Animal implements Moveable, Loveable
  - a Dog “is-a” : Animal, Moveable and Loveable

- As of Java 8, it is now possible to inherit **concrete** methods from interfaces. Interfaces can now contain two types of concrete methods: *static* and *default*.
  - Implementation classes are NOT required to implement an interface’s *static* or *default* methods. The *default* interface methods are inheritable but the *static* interface methods are not.



Interfaces

- Interfaces themselves are implicitly *abstract*
  - public abstract interface I{} ⇔ public interface I
  - top-level interfaces can have *public* or package-private access

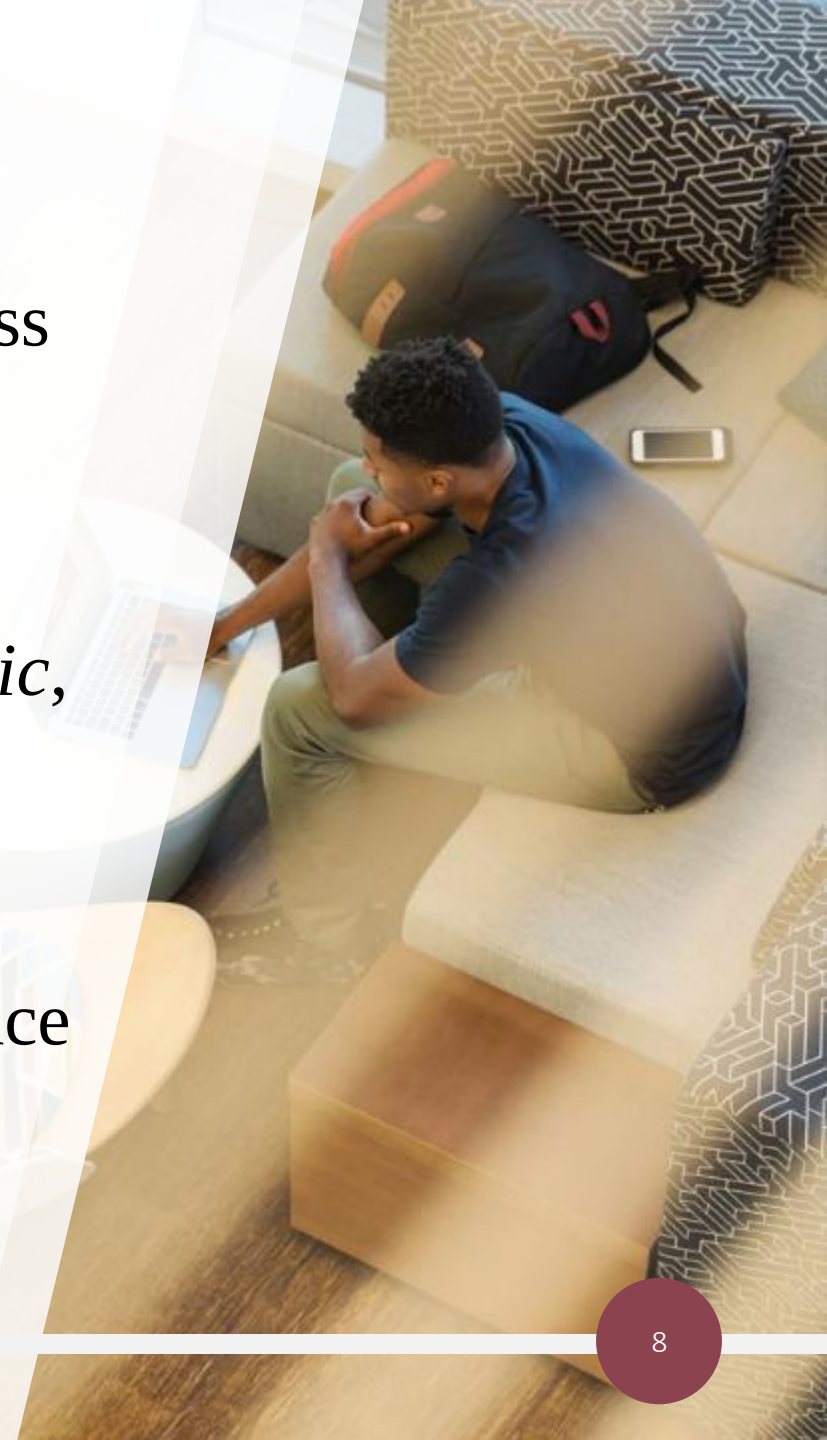
- All interface methods are implicitly *public*

Interfaces

- All interface methods are implicitly *abstract* (unless declared as *default* or *static*, the new features introduced in Java 8).

- All variables declared in an interface must be *public*, *static* and *final* i.e. interfaces can only declare constants (not instance variables).

- As with *abstract* classes you cannot *new* an interface type but they can be used as references:
  - *Printable p = new Printer(); // Printable is an interface*



Functional Interfaces

- A functional interface is an interface that has **only <u>one</u> abstract** method. This is known as the SAM (<u>S</u>ingle <u>A</u>bstract <u>M</u>ethod) rule.
  - *default* methods do not count
  - *static* methods do not count
  - methods inherited from *Object* do not count*

```
@FunctionalInterface
interface SampleFI{
    void m();
}
```

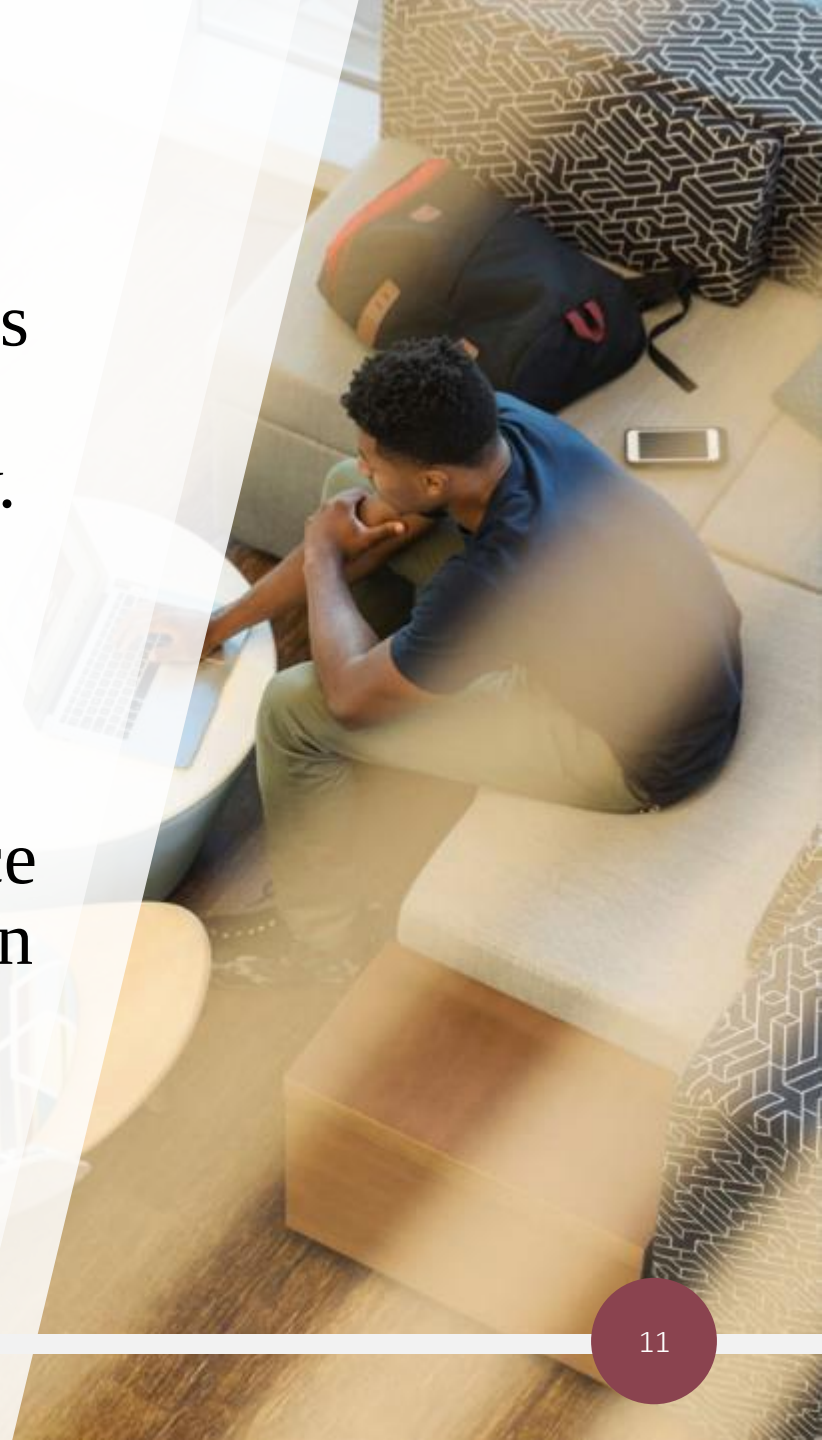
Lambdas

- A lambda expression is just a block of code that helps in making your code more concise.
- A lambda expression only works with functional interfaces.
- **A lambda expression is an instance of a class that implements a functional interface**.

Lambdas

- Lambdas look a lot like methods and in some quarters are called “anonymous methods”. However, it is an instance with everything but the method stripped away.

- A lot can be inferred (by the compiler) from the interface definition (which remember, has only one abstract method). The lambda expression is the instance that implements the interface that has been boiled down to the bare essentials.



```java
interface I{
    void m();// a functional interface i.e. it has only one
             // abstract method
}

public class BasicLambdas {
    public static void main(String[] args) {
        // pre-Java 8
        I i = new I(){
            @Override
            public void m(){
                System.out.println("I::m()");
            }
        };
        i.m();  //  I::m()

        // Java 8 - Lambda expression
        I lambdaI = () ->  {
            System.out.println("Lambda version");
        };
        I lambdaI2 = () ->  System.out.println("Lambda version");
        lambdaI.m();   //  Lambda version
        lambdaI2.m();  //  Lambda version
```

Method code.

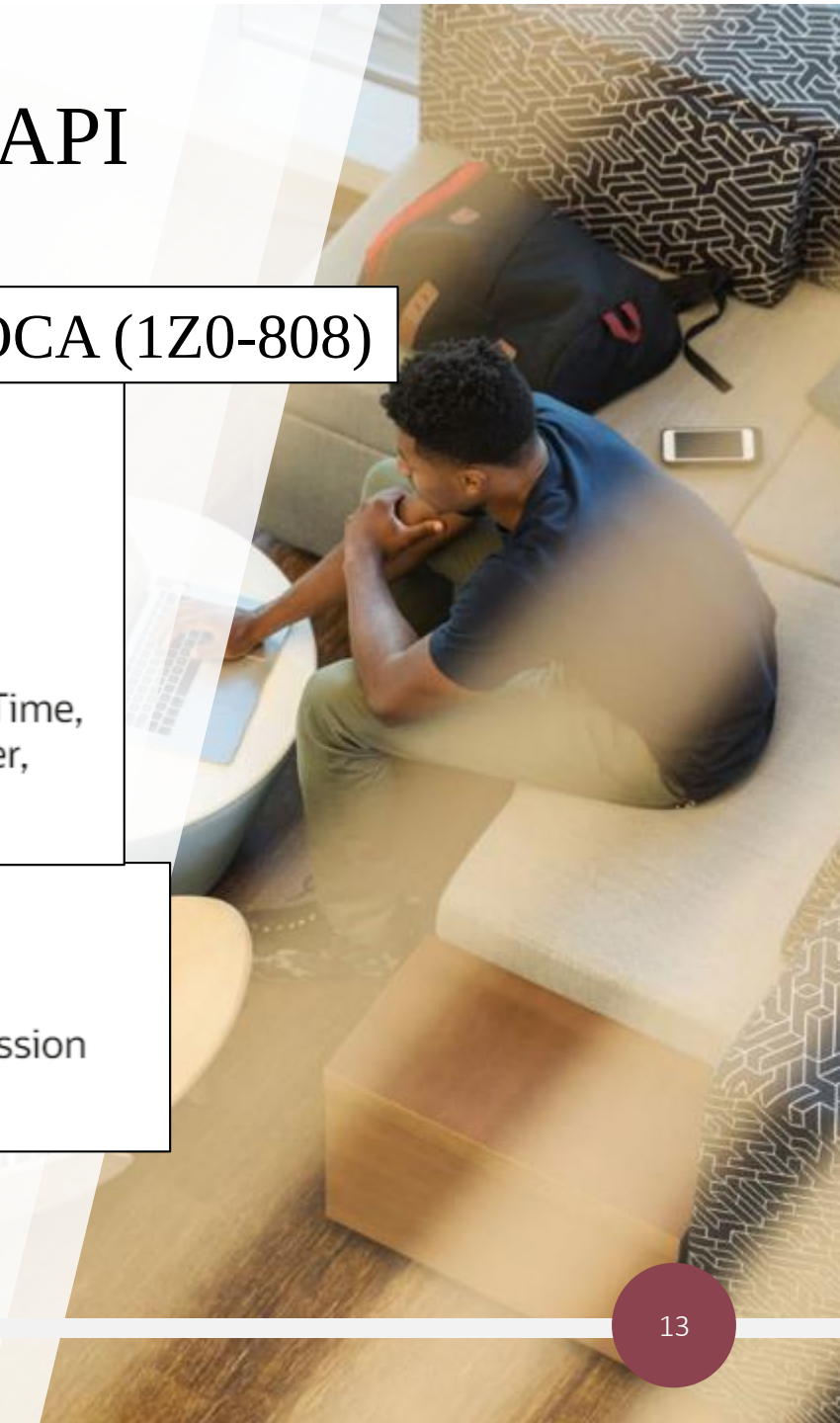
Working with Selected Classes from the Java API

Java 8 OCA (1Z0-808)

Working with Selected classes from the Java API

- ✓ Manipulate data using the StringBuilder class and its methods
- ✓ Create and manipulate Strings
- → Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

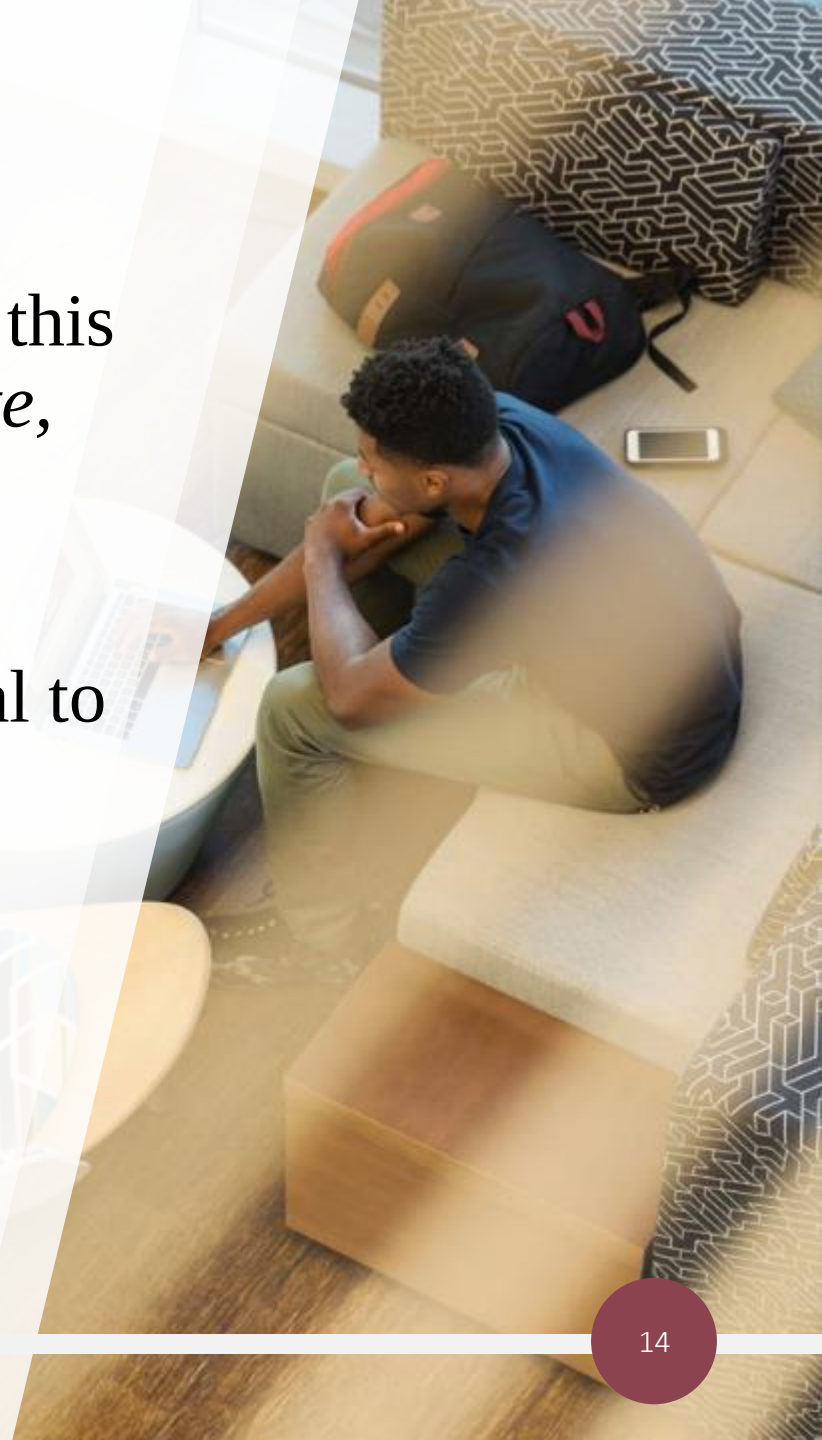
- ✓ Declare and use an ArrayList of a given type
- ✓ Write a simple Lambda expression that consumes a Lambda Predicate expression



Calendar Data

- *java.time* package is new in Java 8 and the classes in this package are designed to replace the “old” *java.util.Date*, *java.util.Calendar* and *java.text.DateFormat*

- Local dates and times - these dates and times are local to your time zone (*LocalDate*, *LocalTime* and *LocalDateTime*). All are immutable.



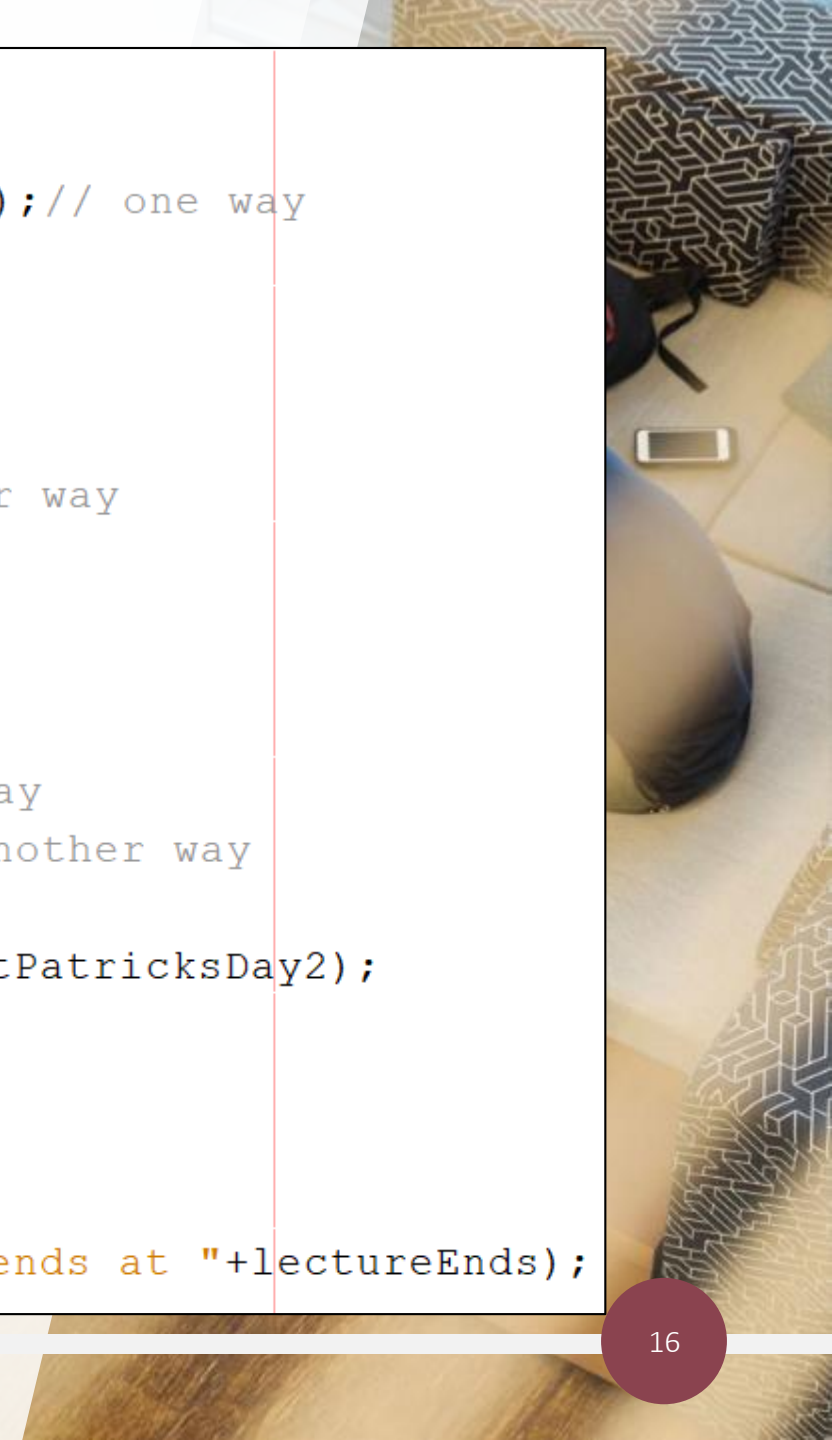
Calendar Data

- Formatters for dates and times - parse/print dates and times according to patterns and styles (*DateTimeFormatter*). Immutable.

- *Period* - represents days or longer. Immutable.

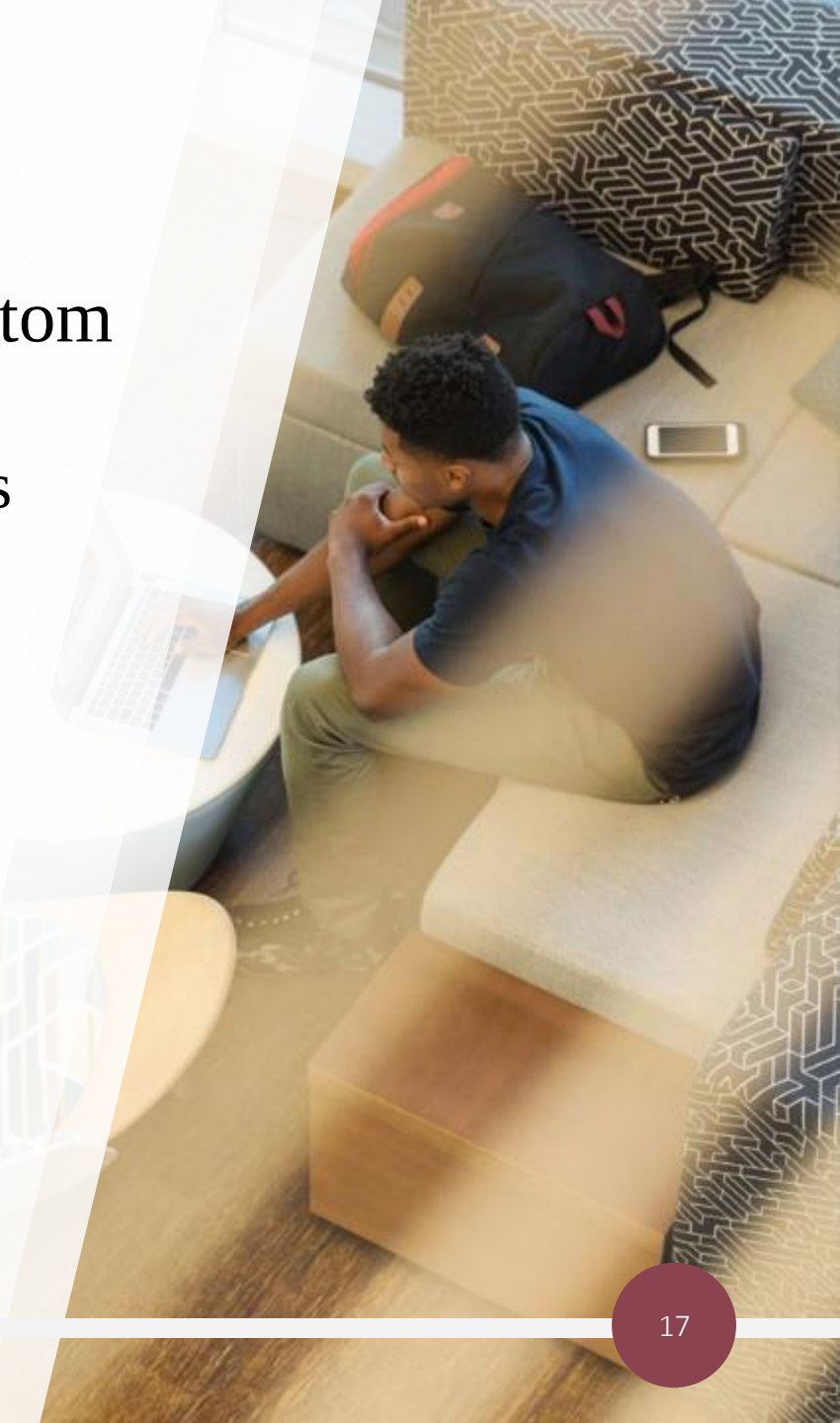
- Do not "*new*" a class in the date/time API. Use factory creation methods e.g. *now()*, *of()* and *parse()*.

```
LocalDate nowDate = LocalDate.now();
LocalTime nowTime = LocalTime.now();
LocalDateTime nowDateTime = LocalDateTime.of(nowDate, nowTime);// one way

// Output:  It is now 2021-06-23T16:59:12.464384100 here.
System.out.println("It is now "+nowDateTime+ " here.");

LocalDateTime nowDateTime2 = LocalDateTime.now();   // another way
// Output:  It is now 2021-06-23T16:59:12.482183600 here.
System.out.println("It is now "+nowDateTime2+ " here.");

// Setting St. Patricks Day, 2022
LocalDate stPatricksDay1 = LocalDate.of(2022, 3, 17);// one way
LocalDate stPatricksDay2 = LocalDate.parse("2022-03-17");// another way
// Output:  St. Patricks Day 2022-03-17; 2022-03-17
System.out.println("St. Patricks Day "+stPatricksDay1+ "; "+stPatricksDay2);

LocalTime lectureBegins = LocalTime.of(9, 0);
LocalTime lectureEnds   = LocalTime.parse("11:00");
// Output:  Lecture begins at 09:00 and ends at 11:00
System.out.println("Lecture begins at "+lectureBegins+ " and ends at "+lectureEnds);
```



Formatting Dates and Times

- In addition to standard formats, it supports a custom format string for customisation:
  - case matters i.e. *M* means month whereas *m* means minutes (of the hour)
  - the number of symbols is important:
    - M would output 1 for January
    - MM would output 01 for January
    - MMM would output Jan for January
    - MMMM outputs the full month, January



Formatting Dates and Times

- The date/time classes have a *format(formatter)* method and *DateTimeFormatter* has a *format(TemporalAccessor)*. Both ways work.
  - *TemporalAccessor* is an interface that *LocalDate*, *LocalTime* and *LocalDateTime* all implement.

Formatting Dates and Times

- Common symbols:
  - y year e.g. 2021, 21
  - M month
  - d day-of-month e.g. 15
  - D day-of-year e.g. 101
  - E day-of-week e.g. Tue
  - h Hour (12 hour clock 1-12)
  - H Hour (24 hour clock 0-23)
  - m Minute
  - s Second e.g. 55
  - S means fraction of second e.g. 328
  - a am/pm

Formatting Dates and Times

```
public static void preDefinedFormats(){
    LocalDate date = LocalDate.of(2021, Month.JANUARY, 1);
    LocalTime time = LocalTime.of(9, 00);
    LocalDateTime dateTime = LocalDateTime.of(date, time);
    System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));// 2021-01-01
    System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));// 09:00:00
    System.out.println(dateTime.format(DateTimeFormatter.
                                       ISO_LOCAL_DATE_TIME));// 2021-01-01T09:00:00


}
```

Formatting Dates

y	year	e.g. 2021, 21
M	month	
d	day-of-month	e.g. 15
D	day-of-year	e.g. 101
E	day-of-week	e.g. Tue

```
    // Date
    LocalDate date = LocalDate.of(2022, Month.AUGUST, 10);
    System.out.println(date);// 2022-08-10
//  DateTimeFormatter format =
//      DateTimeFormatter.ofPattern("yyyy-MMM-dd");// 2022-Aug-10
//  DateTimeFormatter format =
//      DateTimeFormatter.ofPattern("yyyy-MM-dd E D");// 2022-08-10 Wed 222
//    DateTimeFormatter format = // needed 4 E's to get full day i.e. Wednesday
//        DateTimeFormatter.ofPattern("yy-MMM-dd EEEE d");// 22-Aug-10 Wednesday 10
//    System.out.println(date.format(format));
```

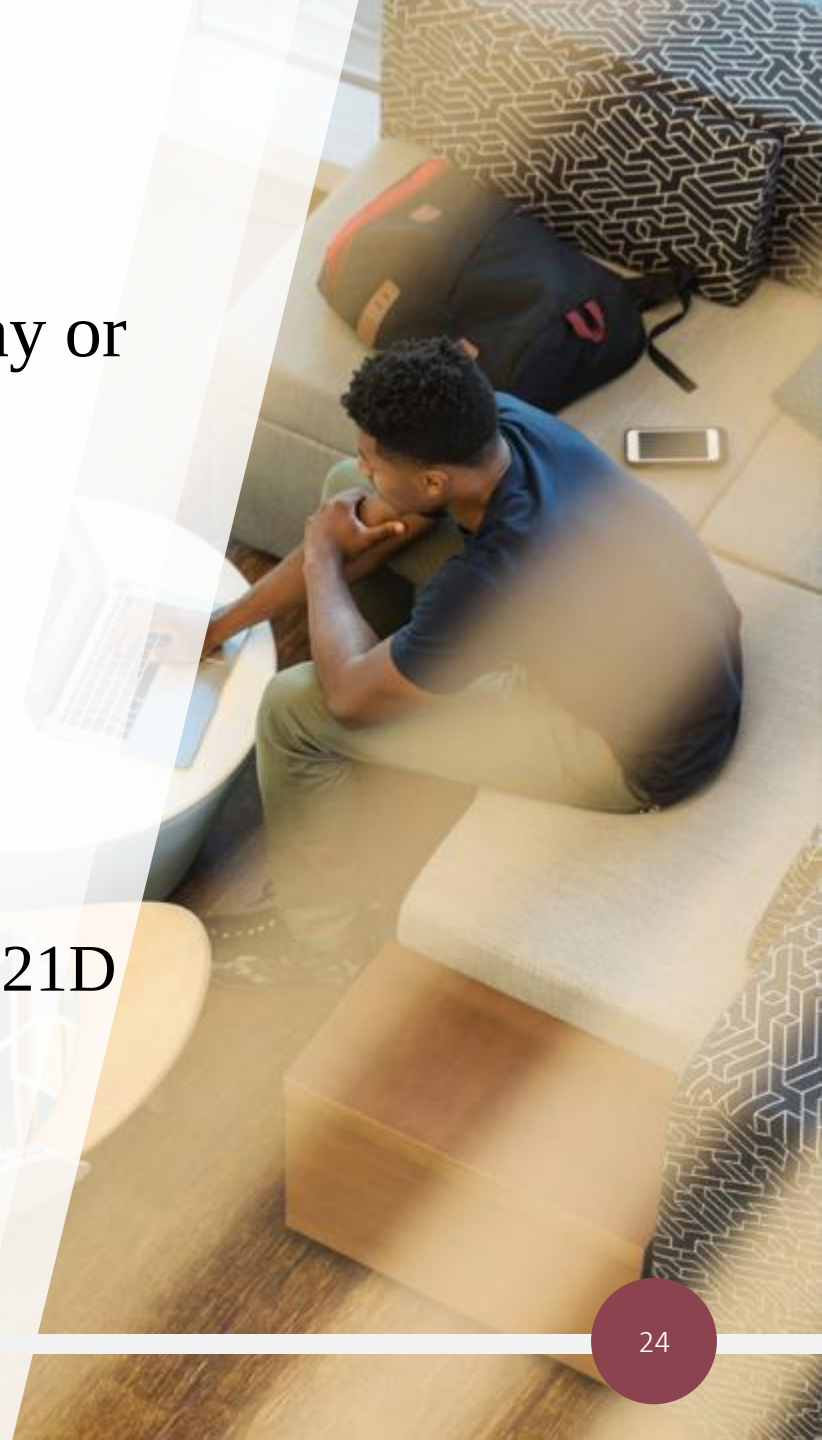
Formatting Times

h	Hour (12 hour clock 1-12)
H	Hour (24 hour clock 0-23)
m	Minute
s	Second e.g. 55
S	means fraction of second e.g. 328
a	am/pm

```
// Time
LocalTime time = LocalTime.of(14, 40);
System.out.println(time);// 14:40
//         DateTimeFormatter format =
//             DateTimeFormatter.ofPattern("hh:mm:ss");// 02:40:00
//         DateTimeFormatter format =
//             DateTimeFormatter.ofPattern("HH:mm");// 14:40

// DateTimeFormatter format = // IllegalArgumentException: Unknown pattern letter: P
//     DateTimeFormatter.ofPattern("Phone call at h:mm a");
DateTimeFormatter format = // Phone call at 2:40 pm
    DateTimeFormatter.ofPattern("'Phone call at' h:mm a");


System.out.println(time.format(format));
```

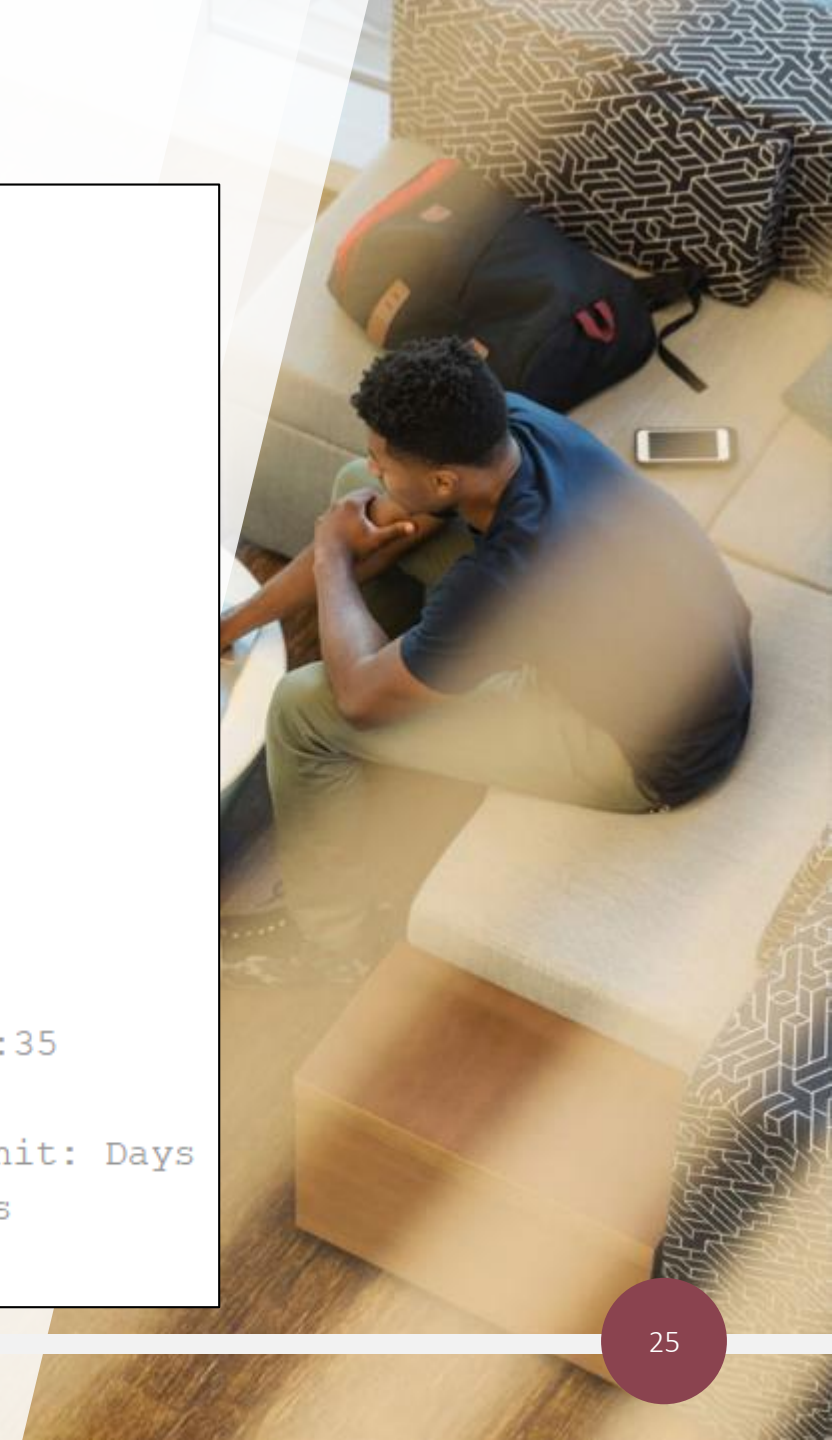
Period

- A *Period* represents years, months and days i.e. a day or more of time.
- *Period*s are output with a P.
- They are created using the static factory methods ofXXX()
  - Period.ofYears(1); // every year, P1Y
  - Period.ofMonths(1); // every month, P1M
  - Period.ofWeeks(3); // weeks are not stored, stored as P21D
  - Period.of(1,2,3); // every year, 2 months and 3 days, P1Y2M3D



Period

```
public static void howPeriodIsOutput(){
    System.out.println(Period.of(1, 2, 3));// P1Y2M3D

    // 1. Java omits measures that are 0.
    // 2. It is ok to have more days in a month.
    // 3. It is ok to have more months in a year.
    System.out.println(Period.of(0, 33, 65));// P33M65D

    LocalDate date = LocalDate.of(2021, Month.MARCH, 1);
    LocalTime time = LocalTime.of(11, 35);
    LocalDateTime dateTime = LocalDateTime.of(date, time);
    Period sevenDays = Period.ofDays(7);

    System.out.println(date.plus(sevenDays));     // 2021-03-08
    System.out.println(date);     // Immutable types => 2021-03-01
    System.out.println(dateTime.plus(sevenDays));// 2021-03-08T11:35

    // next line: UnsupportedTemporalTypeException: Unsupported unit: Days
    System.out.println(time.plus(sevenDays));// 'time' has no Days
}
```



Period

```
LocalDate christmasDay  = LocalDate.of(2021, 12, 25);
Period prepTime         = Period.ofWeeks(4);
LocalDate reminder      = christmasDay.minus(prepTime);
// P = Period, D = days
System.out.println("Period = "+prepTime);  // Period = P28D
System.out.println("Reminder = "+reminder);// Reminder = 2021-11-27
```