COMP2522

Lesson 1: Foundations

INSTRUCTOR

- Jason_Wilder@bcit.ca
- SW2-301
- CST graduate
- C programmer; web and mobile programmer
- Africa
- Option head: Technical Entrepreneurship
- Former Program Head: FSWD
- https://www.bcit.ca/outlines/20261086143/
- Dead Programmers Society: guest speakers, entrepreneurship
- DeadProgrammersSociety.com

ROUTINE

- Lectures: 12 by Jason; one by you
- Labs: work with a partner; marked in class; to get marks, show Jason that day or the following week
- Labs: weekly quiz based on the *previous* week's material
- Quiz is pen and paper; closed book
- In the final week of class, you can redo your worst quiz (not a missed quiz)
- Term project is big (and fun); start early; done outside of class time
- Exams are pen and paper; closed book
- Final exam is not cumulative

Lesson	Topics
1	Classes, modifiers, overloading, static, constructors, strings, comments, conventions
2	Inheritance, exception handling
3	Polymorphism, overriding, equals(), substitution, final and abstract methods/classes
4	Interfaces, default methods, functional interfaces, Comparable
5	Arrays, Collections (lists, sets, maps), iterators
6	Functional interfaces, lambda expressions
7	Method references, generics, bounded types, nested classes,
8	Scanner, Files and Path classes (I/O), directories, Random
9	Streaming and filtering
10	Graphical user interfaces using JavaFX
11	Unit testing and test-driven development
12	Memory, singletons, and more design patterns

LESSON 1 TOPICS

Course overview

Java, Jetbrains IDEA Editor, setting up a project properly

Style guidelines and conventions

Class, Object, Comments, Javadocs, Java Class Library

Final Data, Access Modifiers (private, package, protected, public)

Overloading

Static Data, Static Methods

Primitive Types, Reference Types, and Default Values

Proper Constructor Validation, Constructor Chaining, Instance Initializer Blocks

Strings, String Methods, Regex, StringUtils and importing

IDE

- We will all use Jetbrain's IDEA editor. Use the most-current version of the Ultimate edition.
- How to properly set up a new project:

- File
- New > Project…
- File
- Project Structure…
- Modules
- Sources

- Right-click **src**
- New Directory…
- "code"
- Click the new "code" directory
- Click "Mark as: Sources:"

- Right-click src
- New Directory…
- "tests"
- Click the new "tests" directory
- Click "Mark as: Tests:"
- OK

JAVA TYPES

- Class
- Interface
- Record
- Enum
- Annotation
- Exception

CLASS

- A Java class is a file that describes the data and behaviors of some general category
- e.g: a "BankAccount" class has data such as balanceCad and behaviors such as withdraw()
- The data are known as <u>instance variables</u>; aka fields, attributes, properties
- The behaviors are known as <u>methods</u>; aka functions, procedures
- There is also a function called a constructor, which is called automatically when an instance of a class is being created
- CamelCase, starting with an Uppercase First Letter
- e.g. class BankAccount{}

OBJECT

- An object is one instance of a class
- E.g. my bank account, your bank account, Jeff Bezos's bank account

- The keyword "new" is one way to make objects of a class

- Once you have defined the features of a class ("category"), individual objects can be made of it.

- // below, b1 is a reference to a BankAccount object
- final BankAccount b1;
- b1 = new BankAccount(100.00);

CONVENTION

- Do not do this:
- BankAccount b1 = new BankAccount();
- We will declare all data as final (with a few exceptions)
- We will declare variables together in a group
- We will initialize those variables together later in a group
- The result is code that is extremely organized, fast and simple to read, and easy to debug

- final BankAccount b1;
- final BankAccount b2;
- final BankAccount b3;

- b1 = new BankAccount(100.00);
- b2 = new BankAccount(250.00);
- b3 = new BankAccount(99.99);

- b1.deposit(240.25);
- b2.withdraw(10.00);

VARIABLES AND DATATYPES

- Java variables must be explicitly declared with their datatype
- E.g. **double** balanceCad = 100.00;   // ALWAYS put the units in the name!
- There are two categories of datatypes in Java:
    1. Primitive types: int, double, boolean, char, etc…
    2. Reference types: String, Array, System, BankAccount, etc…
- Primitive types are simple; they hold a value only
- Reference types are more complex; they hold the <u>memory address</u> of an object

DEFAULT TYPES

• int:	When Java encounters a whole number it treats it as an **int**
• double:	When Java encounters a decimal number it treats it as a **double**
• boolean:	When Java encounters **true** or **false**, it treats it as a **boolean**
• char:	When Java encounters a single Unicode character in single quotation marks, it treats it as a **char**
• String:	When Java encounters Unicode characters in double quotation marks, it treats it as a **String**
• Array:	An **array** uses [square brackets]; it stores a fixed number of values of a single datatype

DEFAULT VALUES

Java sets default values (below) for instance variables and static variables

Type	Examples	Default Values
int	-1000, 5, 0	0
double	-1000.234, 5.5, 0.00	0.0
boolean	true, false	False
char	'%', 'A', '5'	'\u0000' (Unicode null character)
String	"hello world", "", "BCIT 2027"	null
array	["hi", "bye", "the end"]	null

NULL

- A String is a reference type
- <u>Any</u> reference type can be null, which means “there is no object” (primitive types cannot be null)
- Strings have a variety of useful methods
- Be careful when using String methods…null does not have *any* methods
- Strings should almost always be initialized to null, not to ""
- null is **not** the same as "":
  - null means no String
  - "" is a String with no characters:
    - but still is located at an address
    - has length() of 0
    - and still has String methods available

NULL

```
final String s1;
final String s2;

s1 = null;
s2 = "";

System.out.println(s1.length()); // NullPointerException!
System.out.println(s2.length()); // 0
```

NULL VS. EMPTY STRING VS. BLANK STRING

- if s == null   true if s is null
- if s.isEmpty()   true if s is not null, but is ""
- if s.isBlank()   true if s is not null, but is empty **or contains only whitespace** e.g. " "

- if(s != null && !s.isEmpty())   // **good** check; won't crash if s is null
  // !s.isEmpty() <u>won't be executed</u> on a null String

versus

- if(!s.isEmpty() && s != null)   // <u>**bad**</u>; crashes if s is null
  // check for null before using *any* String method

- // the next line checks that a String is not null, and contains <u>visible</u> characters
- if(s != null && !s.isBlank())

INSTANCE VARIABLES

- Instance variables <u>must</u> be private, so other classes cannot directly alter their values
- Most instance variables should also be final; if you don't expect or want it to change, don't allow it
- If a variable belongs to a class, make it static

- Use the Principle of Least Privilege: give the smallest possible access to your data (and methods, classes, etc…)

UNITS AND NAMING

- Variables must have very clear names
- Always put the UNITS in the name

- Bad:

  weight   date   name

- Good:

  weightKg   dateBorn   firstName

In 1999, NASA lost a $125M Mars orbiter due to a calculation error where Lockheed Martin engineers used English units (inches, feet) instead of NASA's metric system (centimeters, meters).

FIND SOURCES @ UNBELIEVABLEFACTSBLOG.COM

MAIN() AND PRINTING TO THE CONSOLE

- The entry point of a Java program (where it will begin execution) is the main method, defined as follows:

```
class Main
{
    public static void main(final String[] args)        // args must be final; argv is ok too
    {
            System.out.println("Hello, world!");
            final BankAccount b1;                       // one object
            final BankAccount b2;                       // another object

            b1 = new BankAccount(100.00)
            b2 = new BankAccount(125.50)
    }
}
```

CONVENTIONS: FINAL

- **final** means a variable cannot change its value after it's been set once
- All arguments to all constructors and methods must be **final**
- Most variables should also be **final**
- Declare variables in one line, and initialize them in a separate line

```
class Main
{
      public static void main(final String[] args)
      {
                  final BankAccount b1;
                  final BankAccount b2;

                  b1 = new BankAccount(100.00);
                  b2 = new BankAccount(250.99);
      }
}
```

STATIC

- Variables can be static
- Methods can be static
- Nested classes (more, later) can be static

- Static means "belongs to the <u>class</u> itself", not to objects
- Static means there is one single value per class (NOT one per object)

- No objects are even required in order to access static data or methods

- Static data can (and do) change

STATIC DATA

```
class BankAccount
{
        private static double primeInterestRate = 5.12;  // one single value exists
}
```

STATIC VS INSTANCE DATA

- **Person class**:
- // instance variables: particular to each object
- hair color
- birth date
- first name
- country born in
- weight lb
- eye color
- profession
- salary in Canadian dollars
- // static variables: belong to class, not objects
- population
- weight lb of the heaviest person
- year of first person (e.g. 200000 years ago)
- richest person: elon musk
- breathe air

- **Royal Bank Account class**:
- // instance variables: particular to each object
- account number
- pin
- name
- balance CAD
- account type
- branch
- interest rate
- // static variables: belong to class, not objects
- number of clients
- highest balance CAD
- prime interest rate
- fees
- types of accounts
- headquarters branch

SYMBOLIC CONSTANTS

- Data that is **both static and final** are "symbolic constants"
- Symbolic constants are named using UPPERCASE_WITH_UNDERSCORES
- Use these often
- NEVER put numbers in your code
- Your work will not be marked until all "magic numbers" are gone

```
class BankAccount
{
		private static final double PRIME_INTEREST_RATE = 5.12;
}
```

NO MAGIC NUMBERS EVER AGAIN

- There are 8000000002 people in the world. These 8000000002 people eat a lot of food and cause lots of problems. In 2022, there is still enough food for 8000000001 people. 2023 is improving for lots of people. The hourly minimum wage in bc is $15.20. $15.20 is not easy to buy a house with, especially in 2022. Sometimes it feels like all 8000000001 people are in traffic on my way to work.

NO MAGIC NUMBERS EVER AGAIN

- public static int HUMAN_POPULATION = 8000000009;
- public static final double MIN_WAGE_BC_CAD = 15.99;
- public static final int CURRENT_YEAR = 2026;

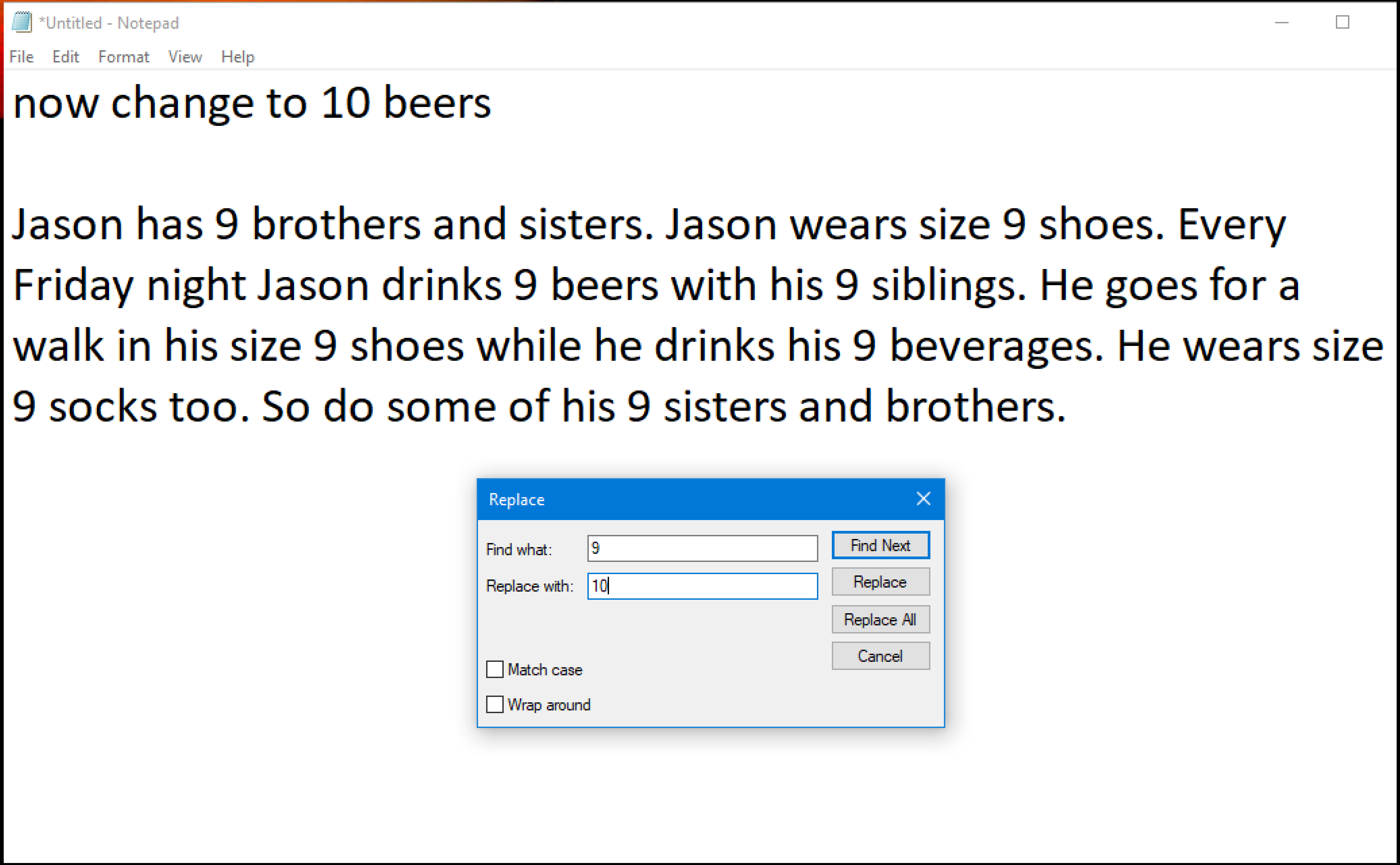
- There are HUMAN_POPULATION people in the world. These HUMAN_POPULATION people eat a lot of food and cause of problems. In CURRENT_YEAR, there is still enough food for HUMAN_POPULATION people. CURRENT_YEAR is improving for lots of people. The hourly minimum wage in bc is $MIN_WAGE_BC_CAD. $MIN_WAGE_BC_CAD is not easy to buy a house with, especially in CURRENT_YEAR. Sometimes it feels like all HUMAN_POPULATION people are in traffic on my way to work.

SPOT THE ERROR

```
if s > 10.0
    sb = false
```

```
if incomingShipSpeedKnots > MAX_SUB_SPD_KNOTS
        incomingShipIsSubmarine = true
```

NO MAGIC NUMBERS



now change to 10 beers

Jason has 9 brothers and sisters. Jason wears size 9 shoes. Every Friday night Jason drinks 9 beers with his 9 siblings. He goes for a walk in his size 9 shoes while he drinks his 9 beverages. He wears size 9 socks too. So do some of his 9 sisters and brothers.

Find what: 9

Replace with: 10

☐ Match case

☐ Wrap around

BETTER

- static final int NUM_SIBLINGS = 10;
- static final int NUM_NIGHTLY_BEERS = 0;
- static final double JASON_SHOE_SIZE = 9.5;

- Jason has NUM_SIBLINGS brothers and sisters. Jason wears size JASON_SHOE_SIZE shoes. Every Friday night Jason drinks NUM_NIGHTLY_BEERS with his NUM_SIBLINGS siblings. He goes for a walk in his size JASON_SHOE_SIZE shoes while he drinks his NUM_NIGHTLY_BEERS beverages. He wears size JASON_SHOE_SIZE socks too. So do some of his NUM_SIBLINGS sisters and brothers.

STATIC METHODS: NO NEED FOR AN OBJECT

```
class BankAccount
{
        private static double primeInterestRate = 5.12;

        static double getPrimeInterestRate()
        {
                return primeInterestRate;  // no object required
        }
}
```

STATIC METHODS

```
public class Circle
{
    public static double getArea(final int radius)
    {
        return 3.14 * radius * radius;
    }
}
```

THE DREADED STATIC ERROR

- Mixing static and non-static
  - static methods cannot directly access non-static (instance) fields
  - throws compilation error

```
private int number;
public static void someMethod(){
    System.out.println(number);
```

Compile Error:
Cannot make static reference to the non-static field number

- One way to fix this is to make number static, BUT this may not be desired as it would then become a class field.

COMMENTS

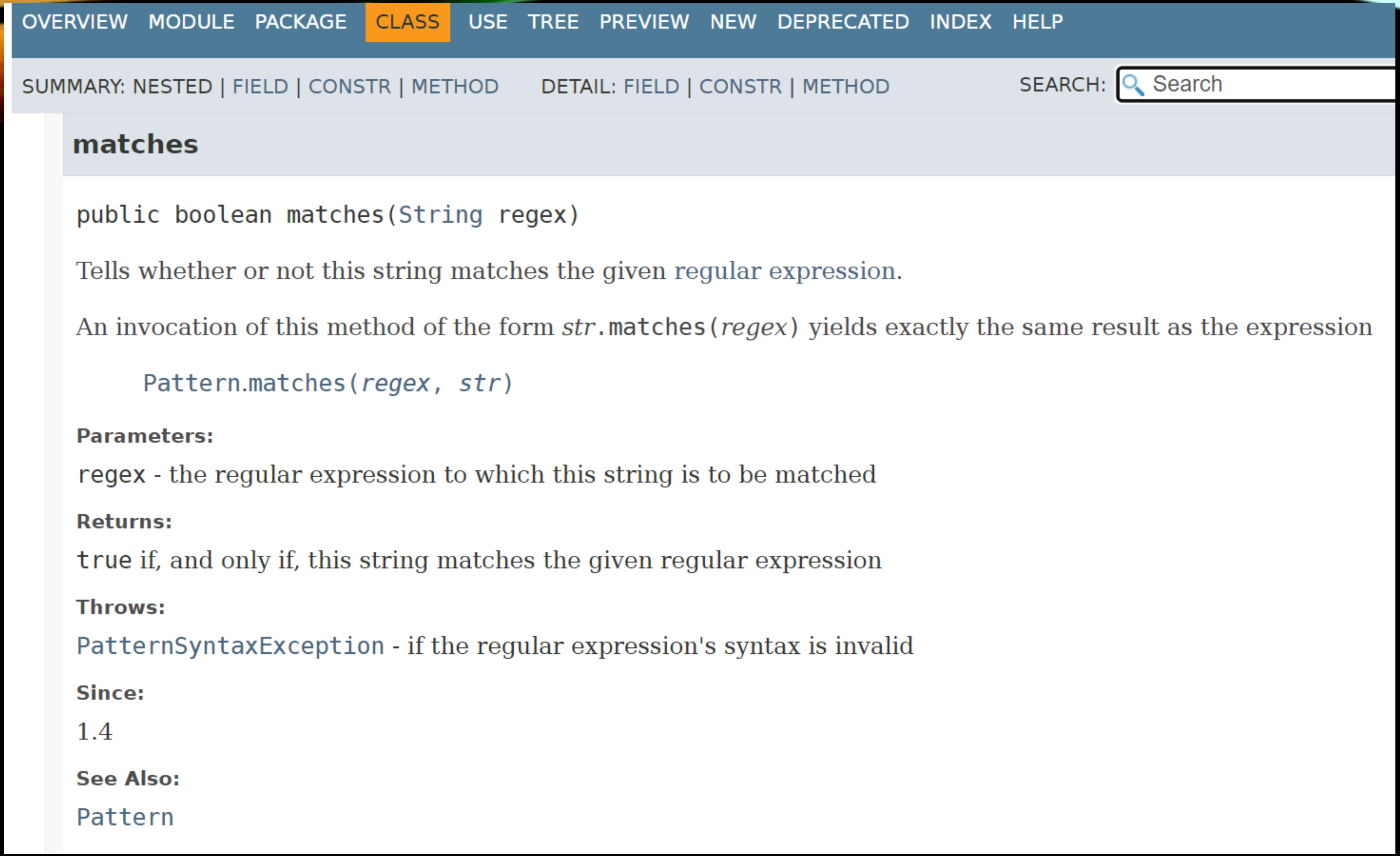
- About 1/3rd of your lines must be comments

- Java comments come in three types:

1. Inline comments // short, in-place comments
2. Multiline comments /* longer comments */
3. JavaDoc comments /** <u>all</u> non-private entities need this */

- JavaDoc comments eventually are published in an HTML class library that acts like a "user manual" for how to use your code.

JAVADOC


OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH: Search

matches

`public boolean matches(String regex)`

Tells whether or not this string matches the given regular expression.

An invocation of this method of the form *str*`.matches(`*regex*`)` yields exactly the same result as the expression

`Pattern.matches(`*regex*`, `*str*`)`

Parameters:

`regex` - the regular expression to which this string is to be matched

Returns:

`true` if, and only if, this string matches the given regular expression

Throws:

`PatternSyntaxException` - if the regular expression's syntax is invalid

Since:

1.4

See Also:

`Pattern`

- Mandatory:
- Classes: description, @author tags, @version tag
- Methods: description, @param tags, @return tag, @throws tags

JAVADOC

```
/**
This class models a Royal Bank account.
@author Jason Wilder
@version 1.0
*/
class RoyalBankAccount
{
        private static double primeInterestRate = 5.12;

        /**
           Returns the prime interest rate.
           @return the prime interest rate
        */
        static double getPrimeInterestRate()
        {
                return primeInterestRate;
        }
}
```

PACKAGE

- A package is a group of related classes
- E.g. The java.math package provides classes for all kinds of math-related code

- <u>Always</u> organize your work into packages

- The first line of a class defines its package
- Package-naming conventions:

1. Lowercase
2. Reverse domain name
3. Meaningful
4. Unique

- E.g. package ca.bcit.comp2522.assignment1.jasonwilder;
- E.g. package ca.amazon;

VISIBILITY/ACCESS MODIFIERS

- 1. Java classes, data, constructors, and methods have a default accessibility of "**package**"
- Classes in the same package have access to this class, data, constructor, or method
- It is often better to explicitly set the access to one of these instead:

2. private:	access is limited to <u>only this very class</u>
3. protected:	access is given to child classes anywhere, and also to any classes in the same package
4. public:	access is given to any code from anywhere

- Note: a public class must be declared in a file of the same name
- e.g. public class Transaction <u>must</u> be declared in Transaction.java

CONSTRUCTOR

- The constructor looks similar to a method, but it is different
- A constructor is called automatically whenever an object is being created with the "new" keyword
- The constructor has the same name as its class
- It is guaranteed to return either an object of that class, or an Exception (more, later)
- The job of the constructor is to <u>initialize</u> an object; for example, to <u>validate</u> the initial instance data values

```
class BankAccount
{
    private static final double PRIME_INTEREST_RATE = 5.12;
    private final String accountNumber;

    BankAccount(final String accountNumber)
    {
        if(accountNumber != null)
        {
            this.accountNumber = accountNumber;
        }
    }
}
```

INSTANCE INITIALIZER BLOCKS

- Even before a constructor is executed, instance initializer blocks are executed for new objects
- These can be used when multiple constructors are re-using the same bunch of code
- In reality, used rarely

```
class BankAccount
{
    private static final double PRIME_INTEREST_RATE = 5.12;
    private final String accountNumber;
    private double balanceCad;

    // instance initializer
    {
        balanceCad = 0.00;  // Java already does this for doubles
    }

    BankAccount(final String accountNumber)
    {
        if(accountNumber != null)
        {
            this.accountNumber = accountNumber;
        }
    }
}
```

STATIC INITIALIZER BLOCKS

- When a class is loaded into memory by the Java Virtual Machine, its static fields and methods are accessible
- A class is loaded into memory (loaded, linked, and initialized):
  - When an instance of the class is created.
  - When a static field or method of the class is accessed.
  - When Class.forName("classname") is called.
  - When a subclass is instantiated or its static members are accessed.
  - When reflection is used to access or modify the class.
- Static initializer blocks can be used to set up complex data or logic

```
class BankAccount
{
    private static final double PRIME_INTEREST_RATE;
    static
    {
        PRIME_INTEREST_RATE = Bank.api.getInterestRate();
    }
}
```

VALIDATION METHODS

- A constructor's main job is to set up objects with valid initial states
- Non-final data also has mutator ("setter") methods that must also validate data
- To guarantee that the proper validation methods are always called, create <u>private</u> <u>static</u> validation methods
- Private static methods are "not overridable" (more, later), which is important
- Call these methods from the constructors and the mutators
- All the validation methods need do is throw Exceptions when bad data values are encountered

VALIDATION METHODS

```
class BankAccount
{
    private double balanceCad;
    private int         pin;

    private final String accountNumber;  // won't change

    private static final int          MIN_PIN_VALUE        = 0;
    private static final int          MAX_PIN_VALUE         = 9999;
    private static final int          ACCT_NUM_LEN          = 6;
    private static final double MIN_BALANCE_CAD = 0.00;

    BankAccount(final String   accountNumber,
                              final double balanceCad,
                              final int          pin)
    {
        validateAccountNumber(accountNumber);
        validateBalanceCad(balanceCad);
        validatePin(pin);

        this.accountNumber = accountNumber;
        this.balanceCad          = balanceCad;
        this.pin                          = pin;
    }

    void setBalanceCad(final double balanceCad)
    {
        validateBalanceCad();
        this.balanceCad = balanceCad;
    }

    void setPin(final int pin)
    {
        validatePin();
        this.pin = pin;
    }

private static void validateBalanceCad(final double balanceCad)
    {
        if(balanceCad < MIN_BALANCE_CAD)
        {
            throw new IllegalArgumentException("bad balance CAD$: " + balanceCad);
        }
    }

    private static void validatePin(final int pin)
    {
        if(pin < MIN_PIN_VALUE | | pin > MAX_PIN_VALUE)
        {
            throw new IllegalArgumentException("bad pin: " + pin);
        }
    }

    private static void validateAccountNumber(final String accountNumber)
    {
        if(accountNumber == null | | accountNumber.isBlank() | | accountNumber.length() != ACCT_NUM_LEN)
        {
            throw new IllegalArgumentException("bad account number: " + accountNumber);
        }
    }
}
```

OVERLOADING

- A class can have multiple constructors, as long as their signatures differ
- i.e., their parameters are different in type, number, and/or order

- Methods can also be overloaded this way (same name but different parameters)

```
BankAccount(final String accountNumber,
            final double balanceCad,
            final int    pin)
{
    validateAccountNumber(accountNumber);
    validateBalanceCad(balanceCad);
    validatePin(pin);

    this.accountNumber = accountNumber;
    this.balanceCad    = balanceCad;
    this.pin           = pin;
}

BankAccount(final String accountNumber, final double balanceCad)
{
    validateAccountNumber(accountNumber);
    validateBalanceCad(balanceCad);

    this.accountNumber = accountNumber;
    this.balanceCad = balanceCad;
}

BankAccount(final String accountNumber, final int pin)
{
    validateAccountNumber(accountNumber);
    validatePin(pin);

    this.accountNumber = accountNumber;
    this.pin = pin;
}
```

OVERLOADING CONSTRUCTORS

```
public Book()
public Book(final String title)
~~public Book(final String authorLastName)~~           // same signature as previous
public Book(final int numPages)
public Book(final String title, final int numPages)
public Book(final int numPages, final String title)
```

In all these cases the compiler can tell them apart:

```
Book b1 = new Book(“harry potter”);          // calls second constructor
Book b2 = new Book(700, “harry potter”);     // calls fifth constructor
```

CONSTRUCTOR CHAINING

- **this()** means the code is calling its own constructor
- If you have two or more constructors, avoid code duplication by having one constructor do most of the work, and the others call that one

```
class BankAccount
{
    private double balanceCad;
    private int    pin;

    private final String accountNumber;

    private static final int        MIN_PIN_VALUE            = 0;
    private static final int        MAX_PIN_VALUE            = 9999;
    private static final int        ACCT_NUM_LEN             = 6;
    private static final double MIN_BALANCE_CAD          = 0.00;
    private static final double DEFAULT_BALANCE_CAD = 0.00;
    private static final int        DEFAULT_PIN              = 0000;

    BankAccount(final String    accountNumber,
                final double balanceCad,
                final int        pin)
    {
        validateAccountNumber(accountNumber);
        validateBalanceCad(balanceCad);
        validatePin(pin);

        this.accountNumber = accountNumber;
        this.balanceCad    = balanceCad;
        this.pin           = pin;
    }

    BankAccount(final String    accountNumber,
                final double balanceCad)
    {
        this(accountNumber, balanceCad, DEFAULT_PIN);
    }

    BankAccount(final String accountNumber,
                final int       pin)
    {
        this(accountNumber, DEFAULT_BALANCE_CAD, pin);
    }
}
```

METHODS IN A CLASS

- A method is a function inside a class
- An instance method belongs to an individual object
- A static method belongs to the class
- A typical class has (<u>in this order</u>, top to bottom):
  - Static variables and constants
  - Instance variables
  - Constructors
  - Setters and getters (mutator and accessor methods)
  - Other methods

METHODS

- A method signature includes <u>only</u> its name and parameters
- A method also has a visibility modifier (private, package, protected, or public)
- Often we provide getter methods for most of the instance variables
- Do not provide setter methods for final instance variables (note: most instance variables <u>are</u> final)

```
void withdraw(final double amountCad)
{
    this.balanceCad -= amountCad;
}

void withdraw(final double amountCad,
                          final int          pin)
{
    if(pin == this.pin)
    {
        this.balanceCad -= amountCad;
    }
}

void withdraw(final double amountCad,
                          final String    accountNumber)
{

if(accountNumber.equalsIgnoreCase(this.accountNumber))
{
        this.balanceCad -= amountCad;
}
```

STRINGS ARE IMMUTABLE

- Once a String has been created, it cannot be modified
- The following code simply points s to a new String (the original String, again, cannot be changed):
  - String s = "hello";
  - s = "world"; // there are now TWO Strings in memory
    // s has the address of "world" in it now
- The following two sections of code run differently in Java too:

**String s1 = "hello";
String s2 = "hello";
// only one String exists; both point to it**
	**String s3 = new String("goodbye");
	String s4 = new String("goodbye");
	// creates two new Strings in memory**

STRINGS ARE IMMUTABLE

- String s1 = "hello";
- System.out.println(Integer.toHexString(s1.hashCode()));

- s1 = "world";
- System.out.println(Integer.toHexString(s1.hashCode())); // new address!

NUMERIC STRINGS ETC

- Sometimes Strings contain String versions of other data types
- E.g.: get input from the user's keyboard (more, later): that input is always a String, but can possibly be "converted" to other types

- We can get the numbers out of numeric Strings in several ways:

```
int a = Integer.parseInt("123");              // 123
int b = Integer.valueOf("123");               // 123
int c = Integer.valueOf("hello");             // NumberFormatException

double c = Double.parseDouble("123.456");     // 123.456
double d = Double.valueOf("123.456");         // 123.456
```

MORE STRING METHODS

- For **non-null** String references, there are many useful String methods we can use:
- https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/String.html

• charAt()	e.g. charAt(0) is the first character of the String
• concat() or +	e.g. String s = “Tiger” + “Woods” which is the same as “Tiger”.concat(“Woods”);
• contains()	e.g. if(s.contains(“the”)) or if(s.toUpperCase().contains(“THE”))
• endsWith()	e.g. if(s.endsWith(“xYz”)) or if(s.toLowerCase().endsWith(“xyz”))
• equals()	e.g. “hello”.equals(“Hello”) is false; “hello”.equals(“hello”) is true
• equalsIgnoreCase()	e.g. “hello”.equalsIgnoreCase(“Hello”) is true
• indexOf()	e.g. “hello”.indexOf(“l”) returns 2 (the index of the first l)
• length()	e.g. “hello”.length() is 5; “”.length() is 0
• replace()	e.g. “hello”.replace(“l”, “T”) creates heTTo
• startsWith()	
• strip()	e.g. “\nhello world \t ” becomes “hello world”
• toLowerCase()	
• toUpperCase()	

MORE STRING METHODS

```
String s1 = "TiGEr wooDs ";
String s2 = "My name is ".concat(s1);
String s3 = "My name is " + s1;

System.out.println(s2);                                  // My name is TiGEr wooDs
System.out.println(s3);                                  // My name is TiGEr wooDs
System.out.println(s1.charAt(0));                        // T
System.out.println(s1.contains("ig"));                   // false
System.out.println(s1.contains("iG"));                   // true
System.out.println(s1.endsWith("oDs"));                  // false; note the space at the end of s1
System.out.println(s1.strip().endsWith("oDs"));          // true; spaces stripped off ends of s1
System.out.println(s1.equals("Tiger Woods"));            // false
System.out.println(s1.equalsIgnoreCase("Tiger Woods ")); // true
System.out.println(s1.indexOf('o'));                     // 7; the first o occurs at the 8th char
System.out.println(s1.length());                         // 12
System.out.println(s1.strip().length());                 // 11
System.out.println(s1.replace('o', 'x'));                // TiGEr wxxDs
System.out.println(s1.startsWith("Ti"));                 // true
System.out.println(s1.toLowerCase());                    // tiger woods
System.out.println(s1.toUpperCase());                    // TIGER WOODS
```

FORMAT STRINGS

- In addition to System.out.print(), use **System.out.printf()** and **String.format()** methods.

- printf():

```
final int yearBorn = 2000;
final double weightKg = 120.129;
final String firstName = "Tiger";

System.out.printf("The golfer %s weighs %.2f kg and was born in %d!\n",
                                        firstName, weightKg, yearBorn);
```

- format():

```
final String s;
s = String.format("The golfer %s weighs %.2f kg and was born in %d!",
                                        firstName, weightKg, yearBorn);
System.out.println(s);
```

- **Both of these print “The golfer Tiger weighs 120.13 kg and was born in 2000!”**

COMPARING STRINGS

- Do not compare Strings using ==
- Strings are compared using .equals()
- There is also .equalsIgnoreCase()
- There is also .compareTo() which tells which String is "bigger"
- A is the smallest letter (smallest Unicode value: 65)
- Z is the largest uppercase letter (90)
- a is even larger than Z (97)
- z is the largest (122)
- https://en.wikipedia.org/wiki/List_of_Unicode_characters

COMPARING OBJECTS (LIKE STRINGS)

- Equality:
  - if(s1 == s2)   // compares addresses, returns **boolean** (you would RARELY do this)
  - if(s1.equals(s2))   // compares the String values, returns boolean
  - if(s1.equalsIgnoreCase(s2)) // “doG” is the same as “DOG” etc…

- Size Comparison:
  - s1.compareTo(s2); // returns an **int**
  - A <u>positive</u> int if s1 > s2;   a <u>negative</u> int if s1 < s2;   <u>zero</u> if s1 equals s2
  - Each character in a String has a numeric Unicode value that can be used to see if Strings are sorted alphabetically

```
if(s1.compareTo(s2) < 0)  // negative means s1 is smaller than s2
{
        System.out.println(“s1 has the lower Unicode value”);
}
```

COMPARING STRINGS

```
public static void main(final String[] args)
{
        final String s1;
        final String s2;
        final String s3;

        s1 = new String("apple");
        s2 = new String("apple");
        s3 = new String("APplE");

        System.out.println(s1 == s2);                       // false
        System.out.println(s1.equals(s2));                  // true
        System.out.println(s1.equals(s3));                  // false
        System.out.println(s1.equalsIgnoreCase(s3));        // true

        System.out.println("hi".compareTo("bye"));          // 6:     h is 104; b is 98
        System.out.println("hi".compareTo("hi"));           // 0:     both are 104
        System.out.println("bye".compareTo("hi"));          // -6:    b is 98, h is 104
}
```

STRINGBUILDER: A BETTER CONCATENATION

- String concatenation can be very expensive when joining a number of String together because Strings are immutable
- StringBuilder offers a more efficient means to join Strings because it works with a single mutable object

```
final StringBuilder builder;
final String generated;
builder = new StringBuilder();          // java.lang.StringBuilder
builder.append("Programming");
builder.append(" ");
builder.append("is");
builder.append(" ");
builder.append("fun");                  // still not a String
generated = builder.toString();         // gives us "Programming is fun"
```

STRINGUTILS

- The Java Development Kit (JDK) contains a lot of Java programming classes: almost 6000 of them
- Third-party libraries extend the capability even more
- Let's use one of the most-popular libraries: Apache Commons' StringUtils class

STRINGUTILS

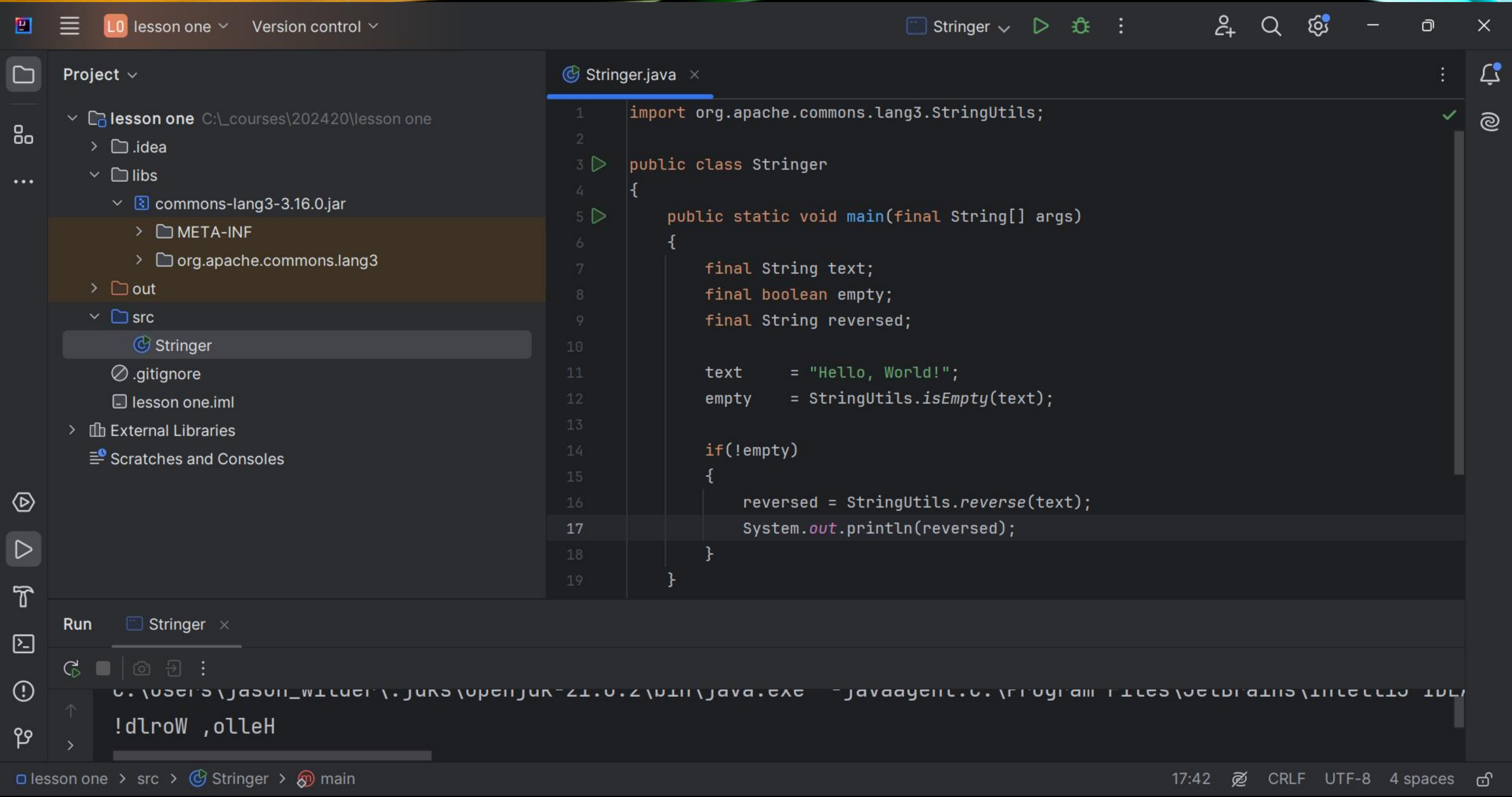
Download the latest library version compressed file from the apache commons lang download page:
https://commons.apache.org/proper/commons-lang/download_lang.cgi

Extract the .jar file

In Jetbrains IDEA:
Create a libs folder in the project navigator
Copy the .jar file there

File
Project Structure…
Modules
Dependencies
+
JARS or Directories…
<Navigate to your .jar file in your libs folder, selecting the **commons-lang3-3.xx.x.jar** file, set Scope to **Compile** > OK

Create a class which can now use the StringUtils methods
import org.apache.commons.lang3.StringUtils;



```
lesson one C:\_courses\202420\lesson one
  .idea
  libs
    commons-lang3-3.16.0.jar
      META-INF
      org.apache.commons.lang3
  out
  src
    Stringer
  .gitignore
  lesson one.iml
External Libraries
Scratches and Consoles
```

```java
import org.apache.commons.lang3.StringUtils;

public class Stringer
{
    public static void main(final String[] args)
    {
        final String text;
        final boolean empty;
        final String reversed;

        text     = "Hello, World!";
        empty    = StringUtils.isEmpty(text);

        if(!empty)
        {
            reversed = StringUtils.reverse(text);
            System.out.println(reversed);
        }
    }
```

```
C:\Users\jason_wilder\.jdks\openjdk-21.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
!dlroW ,olleH
```

TRY ANOTHER THIRD-PARTY LIBRARY

```
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

public class Dater
{
    public static void main(final String[] args)
    {
        final DateTime          dateTime;
        final DateTimeFormatter fmt;
        final String            formattedDate;

        dateTime = new DateTime();
        System.out.println("Current date and time: " + dateTime);

        fmt = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss");
        formattedDate = fmt.print(dateTime);

        System.out.println("Formatted date and time: " + formattedDate);
    }
}
```

CONVENTIONS

- Instance variables are private
- Most instance variables are final
- Always use clear names for data and methods
- Always include units in variable names (e.g. lengthCm)
- JavaDoc the following:
  - the class
  - non-private constants
  - non-private constructors
  - non-private methods
- Use private static validation methods (e.g. in constructors)
- Use constructor chaining to avoid code duplication: this()

CONVENTIONS

- Local variables (i.e. those declared in a method) should be final
- Declare and initialize variables in separate lines
- Use initializer blocks for complex logic
- Compare objects (including Strings) using .equals(), not ==
- Never use magic numbers; use SYMBOLIC_CONSTANTS instead
- Use the principle of least privilege: make everything private and final if you can; if not, make it as restricted as you can while still allowing the application to run