SWEN-601
Software Construction

Variables, Types, & Functions

Java Packages

Activity: Accept the GitHub Classroom Assignment

You will be asked to accept a new assignment at the start of nearly every class.

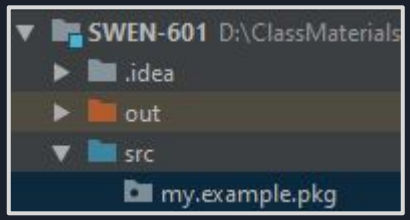
You should get used to accepting the assignment and starting your new project right after you finish your quiz each day.

Your instructor has provided a GitHub classroom invitation. You should be able to find it under “Homework” on MyCourses.

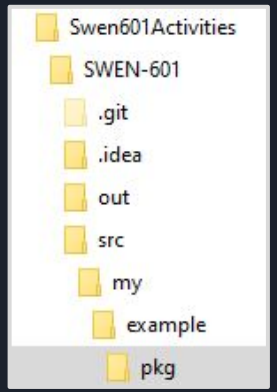
1. Click the GitHub classroom invitation.
2. Assuming that you have already linked your GitHub account with your name in the class roster, you should be prompted to accept the assignment. Do so.
3. Once the repository is created, copy the URL.
4. Clone the repository to your local file system. As before, the repository will be empty.
5. Create a new IntelliJ Project inside the repository, and push it to GitHub.
6. You are now ready to begin today’s activities!

Java Packages

- Java programs very often include many more than one class.
- Simply dumping all of the classes together in one big folder would quickly get very disorganized and difficult to manage.
- Thankfully, Java provides a mechanism for organizing your classes: the ***package***.
- A ***package*** creates a namespace into which you can place Java classes that are closely related to each other.
- A package is named using words separated by dots, e.g. `my.example.pkg`.
- While packages may appear to be hierarchical, the namespaces are ***flat***, i.e. There is no special relationship between "`my.example`" and "`my.example.pkg`".



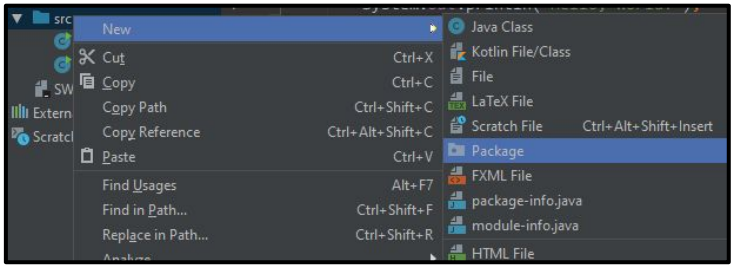
IntelliJ IDEA will flatten empty packages. Once classes have been added, the packages will expand.



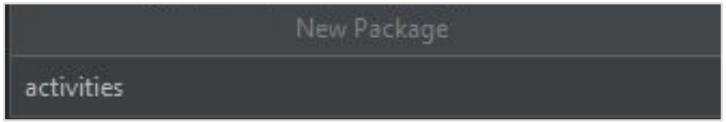
Packages are represented in the file system as folders. Again, the folders may be hierarchical, but there is no relationship.

Activity: Creating a Package

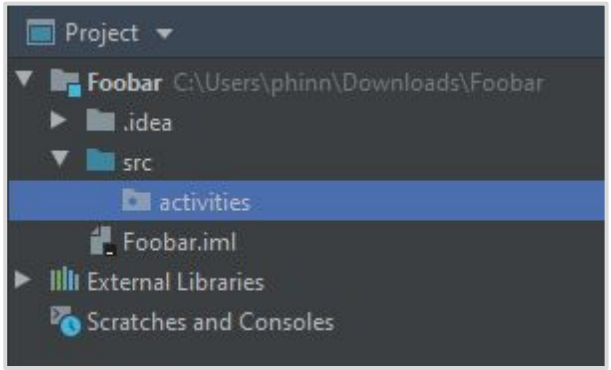
1. Right click on the src folder in your project and select *New → Package* from the popup.



2. In the dialog, name your new package `activities`, and click *OK*.



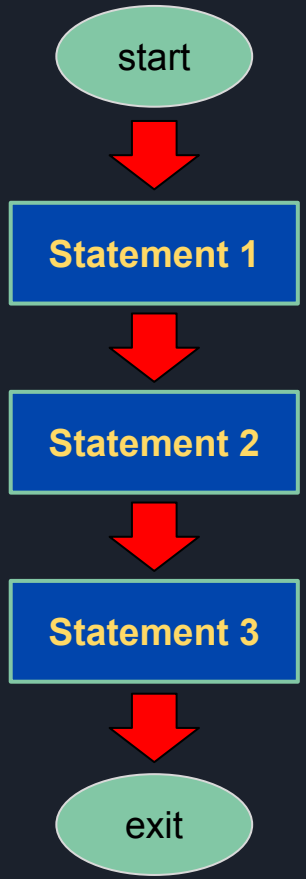
3. This will create a new folder under your src folder. You should implement your solution to all of today's activities in this folder.



Statements,
&
The `Head { Body }` Pattern

Statements & Sequential Execution

- A computer program comprises *statements*.
  - Also typically referred to as “lines of code.”
  - Statements contain instructions for what the computer should do when the statement is executed.
  - In Java, a typical statement begins at the start of a line and ends with a semicolon (`;`) on the same or a later line.
- Statements are executed in the order in which they are written into the program.
  - From top to bottom, beginning with the first line.
  - The program exits after the last statement is executed.
- The sequential execution of statements in a program is referred to as *sequential flow of control*.



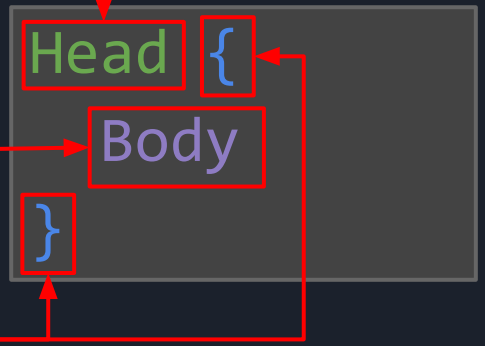
The `Head { Body }` Pattern

- In Java the `Head { Body }` *Pattern* is used to structure computer programs.

There is always some kind of `Head`, like a class or `main` method declaration.

The `Body` follows the `Head` and defines a **block** of 0 or more related statements.

The `Body` is almost always enclosed in a pair of curly braces (`{ }`) that indicate the start and end of the block of statements.



As you write Java, you will see this pattern over and over again.

The Head { Body } Pattern

```
/*
 * Filename: HelloCSC171.java
 */
public class HelloSwen601 {

    public static void main( String[] args ) {

        System.out.println("Hello, SWEN-601!");

    }

}
```

This Head starts with a class declaration...

The Body comprises the statements between the curly braces.

The Head { Body } Pattern

```
/*
 * Filename: HelloCSC171.java
 */
public class HelloSwen601 {

    public static void main( String[] args ) {

        System.out.println("Hello, SWEN-601!");

    }

}
```

You will see that most Java programs repeat the pattern over and over.

The `main` method declaration is another `Head`...

...and the statements inside of the `main` method are another `Body`.

The Head { Body } Pattern

- Unlike some other languages, in Java, whitespace is *insignificant*.
- This means that the following blocks are equivalent to *the compiler*.

```
Head {
  Body Stmt 1;
  Body Stmt 2;
  Body
    Stmt
    3;
}
```

```
Head { Body Stmt 1; Body Stmt 2; Body Stmt 3; }
```

```
Head {
  Body Stmt 1; Body Stmt 2;
       Body Stmt 3;
}
```

- However, stylistic convention says that whitespace *is* used to improve readability of the code for *humans*.
  - This typically means that your code should look like the first example in most cases.

Types, Literals, Variables, Scope, & Expressions

Types

- Very often a statement in a program needs to use some *values* computed by or provided by the computer.
- Every value has a specific ***type***. In Java, there are two broad categories of types: *primitives* and *references*.
- ***Primitive Types*** are the basic building blocks of every other type. They include:
  - Numbers - integers and floating point numbers.
  - Characters - an individual letter, number, or symbol.
  - Booleans - values that are either true or false.
- ***Reference Types*** combine primitives and other references to create more complex compound structures.
  - Strings - So far these are the only reference type we have used.
  - We will talk about more soon!

Java Primitive Types		
Name	Description	Example(s)
`byte`	8-bit signed integer.	`-128, 127`
`short`	16-bit signed integer.	`12876`
`int`	32-bit signed integer.	`-21474836`
`long`	64-bit signed integer.	`4284866286`
`float`	32-bit signed floating point.	`3.14159`
`double`	64-bit signed floating point.	`-14657.241`
`char`	16-bit unsigned integer type; a single Unicode character.	`'a', 'b', '!', '&'`
`boolean`		`true, false`

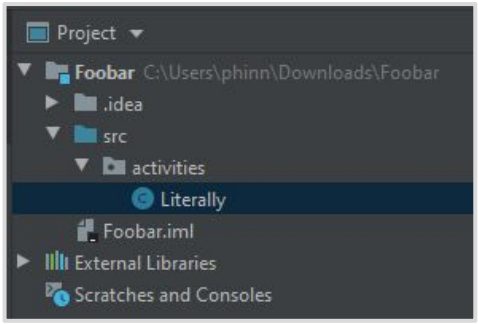
Literals

- A literal is a value that is typed directly into code.
  - Tells the computer to *"literally use this value."*
- No computation is required.
- For example:
  - Numbers: `1`, `5`, `-127436`, `3.14159`, `98.6`, etc.
  - Characters: `'a'`, `'Z'`, `'3'`, `'&'`, etc.
  - Strings: `"Buttercup"`, `"My Dear Aunt Sally"`, `"August 27th, 2019"`, etc.
  - Booleans: either `true` or `false`.
- Remember, a single character (`char`) is a *primitive type* and is enclosed in single quotes (`'`).
- A `String` is a reference type and is enclosed in double quotes (`"`).

While `String`s are reference types, they are very special in Java.

They are afforded many of the same features as primitive types, including the ability to use a `String` literal in your code.

Activity: Literally Printing



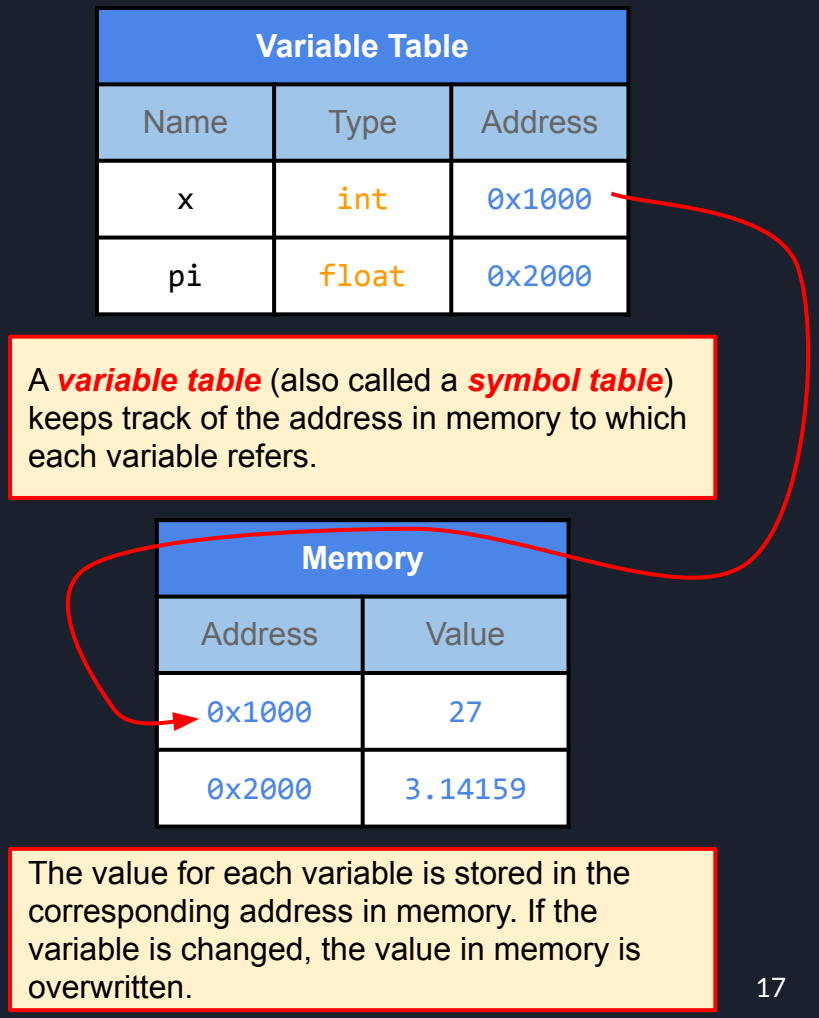
Make sure to create your new class inside the `activities` package!

If you accidentally create it in the wrong place, just drag it into the package.

- Create a new Java class named `Literally`.
- Add a main method.
- Use `System.out` to print a few literals including at least the following:
  - One natural number.
  - One floating point number.
  - A character.
  - A String.
  - A boolean.
- Run your program.
- Push your code to GitHub.

Variables

- A computer is capable of storing, retrieving, and manipulating information.
- The information is stored in and retrieved from *memory*.
- A ***variable*** is a name for a location in memory.
  - This is an alternative to requiring that the programmer keep track of the specific address of a needed value.
- The value stored in the variable's location in memory can, and frequently will, change.
  - When the variable name is used in code, the most recently stored value is substituted in its place.
- There are a few rules related to variables.



(Some) Variable Rules

```
// a variable cannot be used before
// it is declared.
System.out.println(x); // error

// a variable is declared with a
// type and a name.
int x;

// a variable cannot be used before
// a value has been assigned
System.out.println(x); // error

// a literal value of the correct
// type may be assigned
x = 17;
```

```
// the value of the variable may
// be changed at any time
x = 22;


// trying to assign a value of the
// wrong type is a syntax error
x = false;
```

There are *lots* of other rules related to variables, but we aren't ready to talk about them yet. We will add more as we continue to learn about Java.

Scope

- ***Scope*** refers to the parts of a program in which a variable may be used.
- Recall the `Head { Body }` pattern discussed earlier.
  - A variable must be declared within the `Body` of something, e.g. the `Body` of the `main` method.
  - A variable's scope *begins* when it is declared and *ends* at the curly brace (`}`) at the end of the `Body`.
- Trying to use a variable that is out of scope will cause a ***syntax error***.
  - Your program will not compile.
- Trying to declare a second variable with the same name as another variable that is in scope will also cause a ***syntax error***.

```
public class Variables {

  public static void main(String[] args) {
    // x scope begins
    int x = 5;

    System.out.println(x + 7);

    // y scope begins
    char y = 'O';

    // z scope begins
    char z = 'k';

    System.out.print(y);
    System.out.println(z);
  } // x, y, z scope ends
}
```

Expressions

- An ***expression*** tells the computer how to compute a value.
- An expression combines one or more *values* together using ***operators*** to compute some new resulting value.
- There are different operators for different data types.
  - Numbers - arithmetic - add (**+**), subtract (**-**), multiply (*****), divide (**/**), modulo (**%**)
  - Strings - concatenation - `"foo" + "bar"`
  - Booleans - logical operators - and (**&&**), or (**||**), not (**!**)
- Whenever a program requires a value, either an expression or a literal may be used.
  - Provided that it is the right kind of value!

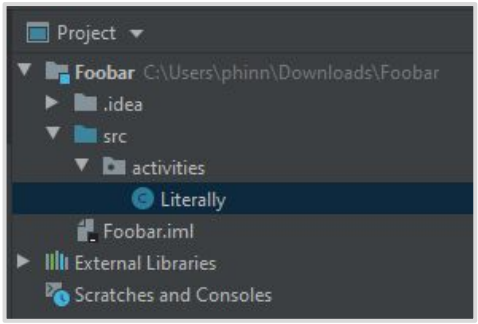
```
// an expression can be used as the
// value assigned to a variable
float x = 22.5 * 13.6 / 12;

// the standard order of operations is
// used (PMDAS)
int y = 12 + 10 / 2 * 6;


// variables may be used in expressions
float z = x + y / 3.1;


// in fact, a variable may be used to
// change its own value
x = x + 4;
```

Activity: Express Yourself



- Create a new Java class named `Expressions`.
- Add a main method.
- Declare three variables of any numeric type.
  - Use an expression to assign a value to each variable.
  - At least one expression must use one variable to assign a value to another, e.g. `int z = x + y;`
- Print the name and value of all three variables.
- Push your code to GitHub.

Make sure to create your new class inside the `activities` package!

If you accidentally create it in the wrong place, just drag it into the package.

Functions, Parameters, & Return Values

Functions

- It is often the case that you would like to execute the same code more than once.
- You ***could*** copy and paste the code, but that has lots of drawbacks.
  - Adds lots of extra code.
  - Duplicates bugs.
  - Changes need to be made in multiple places.
- Virtually all programming languages provide an alternative to cut-and-paste coding: *functions*.
- A function encapsulates a ***block*** of statements.
  - This block is contained within the `Body` of the function.
- Each time you want to execute the statements in the function, you write a ***call*** the function.
  - The statements in the `Body` of the function are executed.

Functions are the primary mechanism for ***reuse*** in programming languages.

Rather than duplicating code (by copying and pasting or retying), a function is called wherever and whenever the code in its body needs to be executed.

You've already written several variations of the main function, which is automatically called whenever your program is executed.

Let's take a look at writing other functions, and then calling them from main (or even each other).

Anatomy of a Function

```
public class Functions {

  public static void sayMyName() {
      System.out.println("Bobby");
  }

  public static void main(String[] args) {
    sayMyName();
    sayMyName();
    sayMyName();
  }
}
```

You have already seen the `main` function, so most of this should be familiar to you by now.

Like many other Java constructs, functions follow the `Head { Body }` pattern.

The `Head` is the declaration which must include the name of the new function, e.g. `sayMyName`.

The `Body` of the function is enclosed in curly braces (`{}`) and should include any statements that you would like to execute when the function is called.

The function is called using its name. Each time it is called, the statements in its `Body` will be executed.

Activity: Printing Pets

- Create a new class in the `activities` package called "`PetNames`."
- Write a function that prints your favorite pet's name to standard output with *one letter on each line*.
  - If you don't have any pets, you may use your own name or the name of a friend.
- Write a `main` function, and call your new function at least twice.
- When you are finished, push your code to GitHub.

The first several lines of your output should look something like this.

Parameters

- It is often the case that a function needs one or more input values in order to do its work.
  - A function that adds two numbers and prints the result.
  - A function that converts temperatures from Celsius to Fahrenheit.
  - etc.
- A function may declare zero or more ***parameters*** as part of its declarations.
  - Parameters are declared between the parentheses after the function's name.
  - Each parameter must be declared with a type and a name.
- When the function is called, a value of the correct type must be specified for each parameter.
  - These values are ***arguments*** to the function.

A parameter is a variable that is declared as part of the method's ***signature***.

Like any other variable, a parameter is declared with a ***type*** and a ***name*** and must be assigned a value before it can be used.

The arguments to the function are used to assign a value to each parameter.

The ***scope*** of the parameter is the entire Body of the function.

Function Parameters

Parameters are declared as part of the method ***signature***, between the parentheses.

Each parameter must be declared with a type and a name.

```
public static void sayHello(String name, int age) {

    System.out.println("Hello, " + name +
      "you are " + age + " years old!");
}
```

The ***scope*** of the parameters is the `Body` of the function, and so they can be used anywhere in the function.

```
sayHello("Bobby", 44);
sayHello("Buttercup", 8);
sayHello("President Obama", 58);
```

When the function is called, an ***argument*** of the correct type *must* be provided for each parameters.

Activity: Arithmetic

```
x=4.0, y=2.0
x + y = 6.0
x * y = 12.0
x - y = 2.0
x / y = 2.0
```

Your output each time that you call the function should look similar to this.

- Create a new class in the `activities` package called "`Calculator`."
- Write a function that takes two floating point parameters x and y. It should print the following:
  - The values for both x and y
  - `x + y = ?`
  - `x * y = ?`
  - `x - y = ?`
  - `x / y = ?`
- Write a `main` function, and call three times with different values.
- When you are finished, push your code to GitHub.

Return Values

- All functions in Java *must* declare a ***return type***.
- The return type indicates the type of value that will be returned when the function is called.
- A return type of `void` indicates that the function does not return a value.
  - So far, all of the functions that we have written have declared a `void` return type.
- If the function declares a return type other than `void`, it ***must*** return a value of that type.
  - This is done using a `return` statement.
  - The `return` statement *must* return a value that matches the declared return type.
  - The return must be the last statement in the method.

If a method declares a return type other than `void` and does not return a value, this will cause a compiler error.

Any code that follows a `return` statement is unreachable. Any such code will also cause a compiler error.

Finally, if a method returns a type that is not compatible with its declared return type, this will also cause a compiler error.

Returning from Functions

```
public static float cubed(float base) {
    System.out.println("Cubing: " + base);
    return base * base * base;
}
```

The return type is declared as part of the method declaration.

All of our previous examples declared a `void` return type, and so a return statement was not necessary.

This function declares a float return type, and so *must* include a `return` statement that returns a floating point value.

```
cubed(5.7);

float result = cubed(3.14159);
System.out.println(cubed(2.5));
```

When the function is called, the returned value may be ignored.

Or it may be used, e.g. assigned to a variable or printed to standard output.

Activity: Pounds to Kilograms

```
pounds: 186.2
kilos: 84.63636
pounds: 207.0
kilos: 94.090904

total: 178.72726
```

Your output should look something like this.

- Create a new class in the `activities` package called "`WeightConverter`."
- Write a function that declares a floating point parameter for a weight in pounds:
  - Prints the weight in pounds.
  - Calculates the weight in kilograms (there are 2.2 pounds to a kilogram) and prints it.
  - Returns the weight in kilograms.
- Write a `main` function that calls your function twice and prints the total weight returned.
- When you are finished, push your code to GitHub.

JavaDoc for Functions

- As mentioned last time, JavaDoc is a special kind of comment that is meant to document your code.
  - Recall that JavaDoc begins with `/**` and ends with `*/`
- JavaDoc can be automatically converted into HTML format to create a web-viewable document of your code.
- JavaDoc is used to document Java functions, including:
  - A description of the function.
  - The name and purpose of each parameter.
  - The return type (if any) and a description of the expected values.
- In IntelliJ IDEA, typing `/**` just before a function and pressing enter will generate a stubbed JavaDoc that you can fill in.

```java
/**
 * This function adds two numbers
 * together and returns the result.
 *
 * @param x The first number to add.
 * @param y The second number to add.
 *
 * @return The sum of the two numbers.
 */
public int adder(int x, int y) {
  return x + y;
}
```

From this point forward, you will be expected to write JavaDoc for all of your functions (including `main`).

Standard Input

Reading Standard Input With Scanner

- Just like `System.out` is used to send output to the terminal, `System.in` can be used to read input typed by the user into the terminal.
  - This is referred to as ***standard input***.
- However, `System.in` is fairly complicated and difficult to use. A much easier alternative is the `java.util.Scanner` class.
- It provides several useful methods, including:
  - `next()` - returns the first word that the user typed (up to the first space).
  - `nextLine()` - reads up to the end of the next line of input (up until the user pressed the enter key).
  - `nextInt()` - reads the next work and returns it as an `int`.
  - `nextFloat()` - reads the next word and returns it as a `float`.
  - etc.
- Refer to the online documentation for many more.

```java
// you will want to import the class
import java.util.Scanner;

public class Input {

  public static void main(String[] args) {
    // create a scanner to read from standard
    // input (System.in)
    Scanner scanner = new Scanner(System.in);

    // read the next line
    String line = scanner.nextLine();

    // read the next word
    String word = scanner.next();

    // read the next word as an int
    int number = scanner.nextInt();
  }
}
```

A call to one of `Scanner`'s methods will ***block*** (your program will pause) until input is available.

Activity: More Arithmetic

```
Enter two numbers: 4.0 2.0

x=4.0, y=2.0
x + y = 6.0
x * y = 12.0
x - y = 2.0
x / y = 2.0
```

Your output should look something like this.

- Modify your `Calculator` class so that the main method uses a `Scanner` to prompt the user to enter the values for x and y.
  - Hint: Use `System.out.print` for the prompt.
- Call your function to print the result of the 4 arithmetic operations (+, *, -, /).
- When you are finished, push your code to GitHub.