Why do we need methods?

Algorithms

- Recall that an algorithm is a precise, unambiguous, step-by-step description for how to produce an answer

- Example algorithm: “How to build a Lego house"
  - Pick foundation piece
  - Lay two rows of bricks for walls
  - Add windows and surrounding bricks
  - Add two rows of bricks above windows
  - Build roof
  - ...

Java version

```
// This code prints instructions to build a Lego house
out.println("Pick foundation piece.");
out.println("Lay two rows of bricks for walls.");
out.println("Add windows and surrounding bricks.");
out.println("Add two rows of bricks above windows.");
out.println("Build roof.");
```

Big problem

- Our house building plans don’t have reusable parts
- Consider making a two-story house...
  - Pick foundation piece
  - Lay two rows of bricks for walls
  - Add windows and surrounding bricks
  - Add two rows of bricks above windows
  - Lay two rows of bricks for walls
  - Add windows and surrounding bricks
  - Add two rows of bricks above windows
  - Build roof

Java version

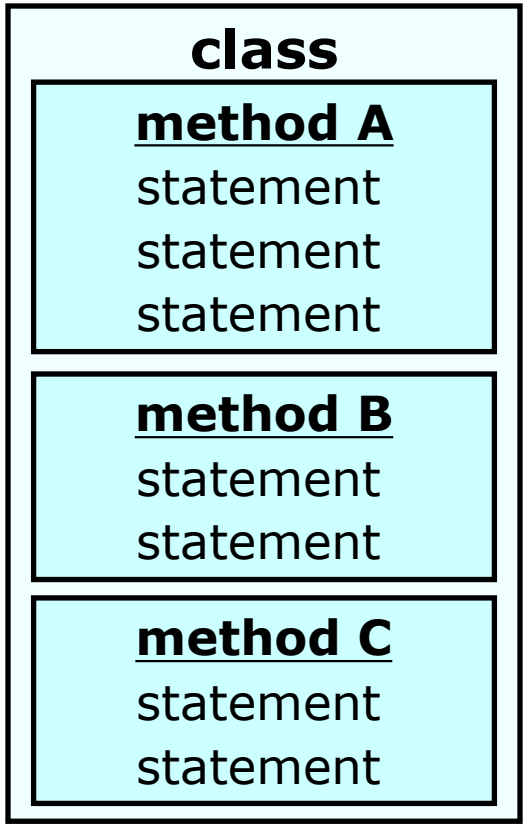
```
// This code prints instructions to build a 2-story house
out.println("Pick foundation piece.");
// build first floor
out.println("Lay two rows of bricks for walls.");
out.println("Add windows and surrounding bricks.");
out.println("Add two rows of bricks above windows.");
// build second floor
out.println("Lay two rows of bricks for walls.");
out.println("Add windows and surrounding bricks.");
out.println("Add two rows of bricks above windows.");
// add roof
out.println("Build roof.");
```

- This redundancy is undesirable since if we need to make a fix or want to make a change, we need to do it in multiple places.

Java Basics: Defining Methods

Methods

- A *method* is a named group of statements
  - captures the *structure* of a program
  - eliminates *redundancy* by code reuse
  - This is *procedural decomposition*: dividing a problem into methods
- Writing a helper method is like adding a new command to Java



Two types of Methods

- In Java, there are two types of methods:
  - Static methods
  - Member methods
- Static methods are helper functions that can do useful things for us
  - All the methods of the **`Math`** class and the **`Character`** class are static methods
- Member methods are associated with an object type and work on the data in an object
  - All the methods of the **`String`** class are member methods

Declaring a static method

- Syntax:

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

Declaring a static method

`public` says anyone can use the method

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

Declaring a static method

`static` says this is not a member method

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

Removing the `static` keyword will make the method a member method

Declaring a static method

the return type specifies the type of the value returned to the caller

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

If the method does not return a value, specify the type `void`

Declaring a static method

This is the name you want to give the method

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

It must obey the rules for a Java identifier

Declaring a static method

This is the name you want to give the method

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

You should use a descriptive name

Declaring a static method

This is the list of parameters that the method will receive from the caller

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

If there are no parameters, the parentheses are left empty

Declaring a static method

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

These are the statements that will be executed when the method is called

Terminology

This is called the method *header*

```
public static return_type name(parameters) {
    statement;
    statement;
    ...
    statement;
}
```

This is called the method *body*

Declaring a static method

- Example:
  - This is a simple example of a void method that takes no parameters and does not return a value

```
public static void welcomeMsg() {
    out.println("Welcome to the Coursera MOOC");
    out.println("on Android programming.");
}
```

Calling a static method

- Syntax:

  name(**parameters**);

  – You can call the same method many times

- Example:

```
welcomeMsg();
```

  – Output:

```
Welcome to the Coursera MOOC
on Android programming.
```

Control flow

- When a method is called, the program's execution...
  - "jumps" into that method, executing its statements, then
  - "jumps" back to the point where the method was called

- Each method continues its execution from the point of the call when the call returns

- Thus one method can call another method, which can itself call a different method, etc.

Control flow

```
public static void twelveDays() {
    day1();
    day2();
}
public static void day1() {
    out.println("A partridge in a pear tree.");
}
public static void day2() {
    out.println("Two turtle doves, and");
    day1();
}
```

- Output:

```
A partridge in a pear tree.
Two turtle doves, and
A partridge in a pear tree.
```

Control flow

```
public static void twelveDays() {
    day1();

    day2();
}
```

```
public static void day1() {
    out.println("A partridge in a pear tree.");
}
```

```
public static void day2() {
    out.println("Two turtle doves, and");
    day1();
}
```

```
public static void day1() {
    out.println("A partridge in a pear tree.");
}
```

When to use methods

- Place statements into a static method if:
  - The statements are related structurally, and/or
  - The statements are repeated

- The order of methods in a class does *not* matter to Java
  - Pick a sensible order for humans
  - Example: important methods at top

Java Basics:
Method Parameters

Parameterization

- A *parameter* is something passed to a method by its caller
  - We already saw parameters when we called the method of the Math class
  - Now we want to add parameters to our own methods

- A parameter is a variable with a slight twist:
  - Declared by a method
  - Initialized by each call to the method

Declaring a parameter

- Stating that a method requires a parameter in order to run
- Every parameter has a type and a name

```
public static void name ( type  name ) {
    statement(s);
}
```

- Example:

```
public static void sayHello(String name) {
    out.println("Hello, my name is " + name);
}
```

  – When `sayHello` is called, the caller must specify a string value to print (i.e., initialize the parameter variable).

Passing parameters

- Calling a method and specifying values for its parameters

  name (**expression**) ;

  This does the initialization; there is no assignment operator involved

- Example:

```
public static void test() {
    sayHello("Doug");
}
```

Output:

```
Hello, my name is Doug
```

Multiple parameters

- A method can accept multiple parameters
  - When calling it, you must pass values for each parameter

- Declaration:

```
public static void name (type name, ..., type name) {
    statement(s);
}
```

- Call:

```
name (value, value, ..., value);
```

- Values and parameters are matched <u>by position</u>

Overloading

- Two or more methods can be defined with the same name *if* the parameter list can be used to determine which method is being invoked
- This useful ability is called *overloading*
- The number of arguments *and* the types of the arguments determines which method is invoked.
  - If there is no exact match, Java attempts the automatic type conversions, of the kinds discussed earlier, to create a match
  - If there is still no match, an error message is produced

Overloading

- Example:

```
public static void printAve(int num1, int num2) {
    double ave = (num1+num2)/2.0;
    out.println("The average was: " + ave);
}

public static void printAve(int num1, int num2, int num3){
    double ave = (num1+num2+num3)/3.0;
    out.println("The average was: " + ave);
}

public static void printAve(double num1, double num2) {
    double ave = (num1+num2)/2.0;
    out.println("The average was: " + ave);
}
```

Scope

- **scope**: The part of a program where a variable exists
  - From its declaration to the end of the enclosing `{ }` braces
    - A variable declared in a method exists only in that method.

```
public static void example() {
    int x = 3;
    {
        int i = 5;
        out.println(x + " " + i);
    }
    // i no longer exists here
}
// x ceases to exist here
```

i's scope

x's scope

Scope implications

- A parameter has the scope of the method it is declared in
- Variables without overlapping scope can have same name

```
public static void printAve(int num1, int num2) {
    ...
}
public static void printAve(double num1, double num2) {  // OK
    ...
}
```

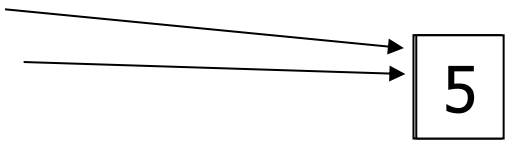
- A variable can't be declared twice in the same scope or used out of its scope

```
public static void printAve(int num1, int num2) {
    int num1 = 2;            // ERROR: overlapping scope
    ...
}
```

How parameters are passed

- When the method is called:
  - The value is copied into the parameter variable
  - The method executes using that value
  - This is known as the *call-by-value* mechanism
    - Also known as *pass-by-value*

```
public static void process() {
     int x = 5;
     double(3);
     double(x);
}
public static void double(int num) {
     out.println("Your value doubled is: " + 2*num);
}
```



Value semantics

- Modifying the parameter will <u>not</u> affect the caller's variables, even those used to initialize the parameter
  - This is because the parameter receives a <u>copy</u> of the value

```
public static void process() {
    int x = 16;
    funny(x);
    out.println("2: x = " + x);
    ...
}
public static void funny(int x) {
    x = 2 * x;
    out.println("1: x = " + x);
}
```

Output:

```
1: x = 32
2: x = 16
```

Java Basics:
Method Return Values

Defining Methods That Return a Value

- As before, the method definition consists of the method header and the method body
  - But the type of the value being returned replaces `void`

- Example:

```
public static double tripleIt(double number)
{
    double result = 3.0 * number;
    return result;
}
```

Defining Methods That Return a Value

- The body of the method definition must contain

  `return` *`Expression`*`;`

  - This is called a *return statement*
  - The *`Expression`* must produce a value of the type specified in the heading
- The body can contain multiple `return` statements, but a single `return` statement often makes for better code
  - The *first* `return` statement executed completes the execution of the method and control returns to the caller *immediately*

Return examples

```
// Returns the volume of a sphere with radius r
public static double sphereVolume(double r)
{
   double volume = (4.0/3.0) * Math.PI * Math.pow(r,3);
   return volume;
}


// Return a double value rounded to 2 decimal places
public static double round2(double num)
{
   int scaleUp = (int)(num * 100.0 + 0.5);
   double result = scaleUp / 100.0;
   return result;
}
```

Return examples

- You can shorten the examples by returning an expression:

```
// Returns the volume of a sphere with radius r
public static double sphereVolume(double r)
{
   return (4.0/3.0) * Math.PI * Math.pow(r,3);
}


// Return a double value rounded to 2 decimal places
public static double round2(double num)
{
   return (int)(num * 100.0 + 0.5) / 100.0;
}
```

static methods in this MOOC

- We've focused on static methods in this lesson
- All the methods that print data to the screen work correctly if the "**`out`**" object is globally available
  - The **`System.out`** object used in desktop computing is global
- Unfortunately, the "**`out`**" object is not global but is rather a member of our **`Logic`** class in which we are operating
- For our methods to access the "**`out`**" object, the methods cannot be static
- So simply remove the **`static`** keyword from the method headers if you need to access the "**`out`**" object