303.4.2 - 303.5.1
Java Classes, Methods, Constructors, Objects, Access Modifiers, and Overloading

Reusable Units of Work

Lesson 1

Object-Oriented Programming

Learning Objectives:

By the end of this lesson, learners will be able to:

- ❑ Define Object-Oriented Programming concepts.
- ❑ Utilize classes and class members in Java.
- ❑ Explain objects and instances.
- ❑ Describe Constructor, Constructor Overloading, and Access Modifier.

Agenda

- Topic 1: Overview of Object-Oriented Programming.
  - Pillars of Object-Oriented programming.
- Topic 1a: Introduction to Java Classes.
  - What Are Classes?
  - Classes Composing Functionality.
  - What are Class Members?
  - Class Definition in Java.
  - Where do Objects Come From?
  - Class Members - Fields (Class Variables).
  - Local variables.
- Topic 2: Access Specifiers.
- Topic 3: Where do Objects Come From?
- Topic 4a: Class Members - Constructors.
- Topic 4b: Constructor Overloading.

1. Overview of Object-Oriented Programming

- Object-Oriented programming (OOP) is a programming paradigm, which uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs.
- The emphasis is on data rather than on the procedure.
- Here, we observe two of the tenets of OOP:
  - **Reusability**. Classes, once defined, are templates for highly functional objects that we can use and re-use.
  - **Composition**. We can define classes that bring disparate types of objects together to achieve advanced functionality and solve specialized problems.

Overview of Object-Oriented Programming (continued)

- **Classes** and **Objects** are basic concepts of **Object-Oriented Programming (OOP)**, which revolves around the real-life entities.
- OOP relies on a few things that you have already used:
  - Classes – Contains *skeletons* for your code.
  - Objects – Contains runtime entities.
  - Methods – Contains the logic of your code.
  - Variables – Contains values your code can use.

Pillars of Object-Oriented Programming

The four pillars of (OOP) are:

1. **Encapsulation**: The process of wrapping code and data together into a single unit.
2. **Abstraction**: The process of hiding implementation details and exposing only the functionality to the user.
3. **Inheritance**: The process of one class inheriting properties and methods from another class.
4. **Polymorphism**: The ability to perform many things in many ways.

1a: Introduction to Java Classes

- Classes are the foundation of any Java application. All Java code runs within the context of a class. You may recall from our first look at a console application that the ***main()*** method — the entry point for the JVM — was written within a class definition.

- Well-designed classes represent reusable units of work. For instance:
  - The String class gives you all of the capabilities that you need to work with string data.
  - The Scanner class gives you the ability to read and parse values from a console, file, or other sources of character data.

- Classes model the real world, or for games and other exercises in imagination, the unreal world.
  - **Medical** records software will use classes to represent patients, pathologies, and therapies.

  - A **flight simulator** will use classes to represent wind, weather, and a jet. The jet class will certainly be composed of other classes that represent its many components – wings, landing gear, hydraulics, cockpit controls, etc.

What Are Classes?

- In Java, a ***class*** is a definition of ***objects*** of the same kind. In other words, a *class* is a **blueprint, template,** or **prototype** that defines and describes the *static attributes* and *dynamic behaviors* common to all objects of the same kind.
- A **class** in Java is a **logical template** to create **Objects** that share common properties and methods.
- When we define a class, we provide **fields** to hold the data, **getters/setters** to provide access to the data, and **methods** to work with the data. We might also provide mechanisms to notify consumers of the class of changes made to the data. We may also provide **constructors** to initialize instances of the class when they are created.

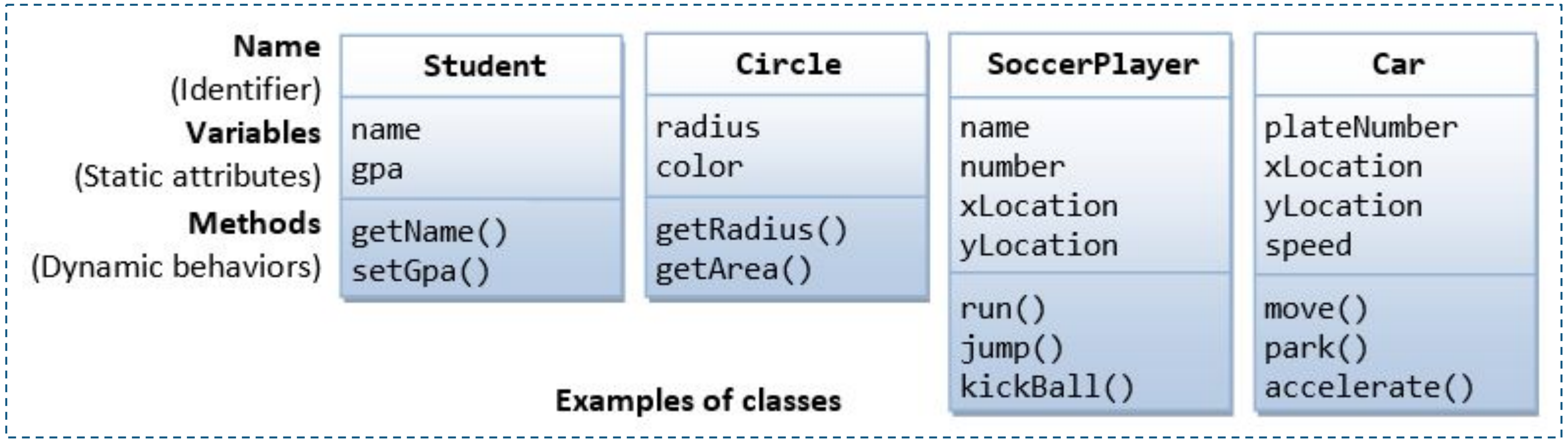
Classes Composing Functionality

- ❑ When creating an application, bring multiple classes together to accomplish the functional objectives of that application. You will write some of the classes yourself.
- ❑ Many others will come from packages — either the JDK packages, or the many packages that the developer community has made available.
- ❑ You will discover that some of the classes have utility in several applications, so you will create your own shareable packages.
- ❑ **Composition and code-reuse** are two of the core concepts of Object-Oriented Programming (OOP).

What are Class Members?

Classes are composed of distinct parts:

- **Fields** *(also called class variables and attributes,)* - local to the class; they give an object *state*.
- **Methods** *(or behaviors and operations)* - specialized to work with the fields and manage the state of objects.
- **Constructors** - give the ability to initialize objects once they are created.
- **Finalizer** - special methods that are called by the Garbage Collector.



Source: ntu.edu.sg

Class Definition in Java

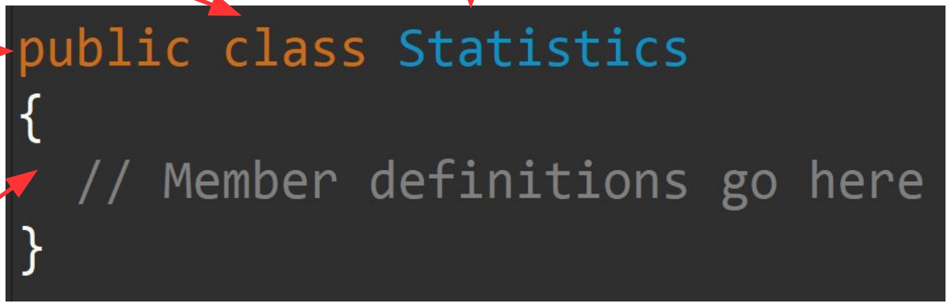
How to create a Class in Java

In Java, we use the keyword ***class*** to define a class, but cannot start with a number. The best practice is to use Pascal casing; although, it is not enforced by the compiler.

Classes are declared using the class keyword.

Classes can optionally have an access specifier. Classes declared without an access modifier have package access.

The body of a class is contained within a pair of curly braces.

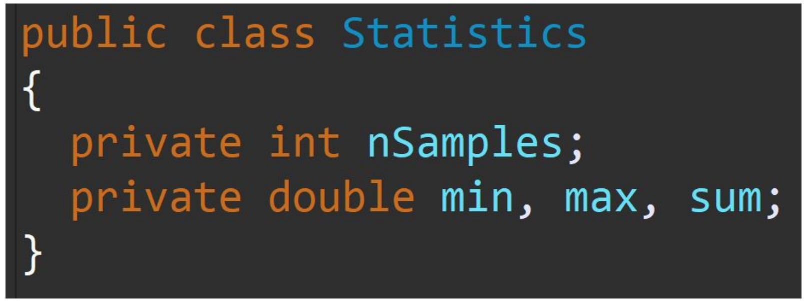


```
public class Statistics
{
  // Member definitions go here
}
```

Class Members - Fields (Class Variables)

- Let's design our class named ***Statistics*** to calculate the minimum, maximum, and sum of a collection of ***double values***. We need **variables** in which to store these values.
- **Variables** defined within a class are called ***Fields*** or ***Class variables*.**
- Fields or Class variables can be declared anywhere within the class definition; but by convention, they are declared near the **top of the class definition.**

Here, we have declared four fields.

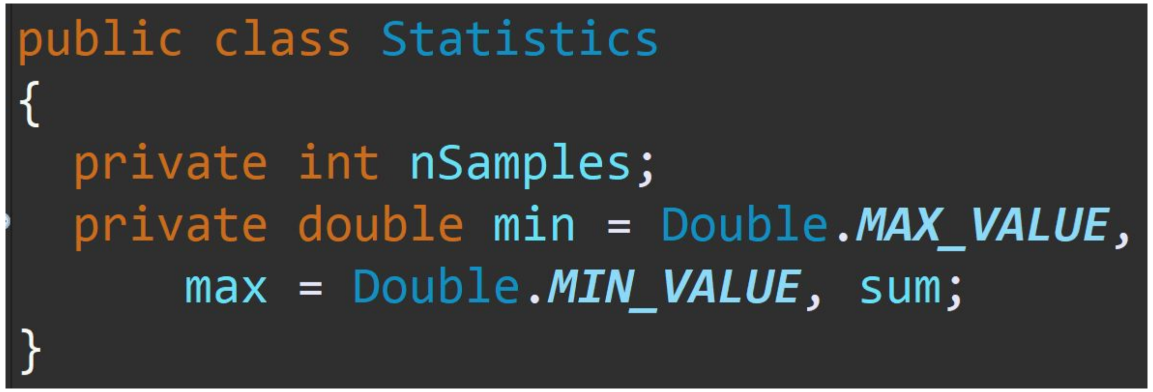


```
public class Statistics
{
  private int nSamples;
  private double min, max, sum;
}
```

continue…

Initializing Fields (Class Variables)

- Fields are automatically initialized to default values; they can be explicitly initialized.
  - Numeric types are initialized to **0**.
  - Reference types are initialized to **null**.
- For our Statistics class, it makes sense to initialize ***min*** and ***max*** so that they are ready to be used in the loop where we will process a collection of double values.



```
public class Statistics
{
  private int nSamples;
  private double min = Double.MAX_VALUE,
      max = Double.MIN_VALUE, sum;
}
```

Local Variables

Local variables are defined **inside** the ***method (not outside the method)***. They are declared and initialized within the method body.

When declaring a ***local variable***, we must initialize it with a ***default value*** within the method block because the JVM DOES NOT provide any default values for local variables.

Remember: *Class variables or* Fields are similar to local variables except:

- Fields can be accessed anywhere within the class definition.
- Fields can be given **access specifiers.**
- Class variables are members of Class and local variables are members of method.
- Fields must declare a default value for local variables.

```
public class MyClass {
    private int max; // Class variable or field

    public static void main(String[] args) {

        int number = 10;   // local variable
        number = number + 20;
    }

 }
```

2. Access Specifiers

Access specifiers (also called access modifiers), as the name suggests, access modifiers in Java to help restrict the scope of a class, constructor, variable, method, and data members.

- **public** – visible/access to **all other classes or everywhere.**
  - `public class BankAccount`
- **private** – visible/access **only to the current class**, its methods, and every instance (object) of its class:
  - a child class cannot refer to its **parent's** private members - `private String myID;`
- **default** (no specifier) – the class or member is accessible to the class itself and to other classes defined within the same package.
- **protected** – visible to the current class, and **all of its child classes:**
  - `protected int myWidth;`
  - **child classes** are classes that inherit from ("extend") this class.

3. Where do *Objects* Come From?

An ***Object*** is the **instance of the class**, which helps programmers use variables and methods from inside the class. However, we can create multiple instances of the *Object* for the same class.

- The class has to be declared only once, but Objects of that class can be declared several times, depending on the requirement.
- Objects act as a **variable** of the class.
- A **class** does not take any memory spaces when a programmer creates one. An **object** takes memory when a programmer **creates an Object** of that class.

Instantiation: Instantiate == create an instance == create an object of a class.

- In Java , the **new** keyword is used to create new ***Objects***.
  - **Declaration** − A variable declaration with a variable name with an object type.
  - **Instantiation** − The **'new'** keyword is used to create the object.
  - **Initialization** − The **'new'** keyword is followed by a call to a constructor.

Creating an *Object* for Statistics Class

Creating a **object** of ***Statistics*** class.

Caller class (main class)

Instantiate of Statistics class

```
double[] values = new double[100];
for(int i=0;i<100;++i) values[i] = Math.random() * 100;
Statistics stats = new Statistics(values);
```

```
public class Statistics
{
  private int nSamples;
  private double min = Double.MAX_VALUE,
      max = Double.MIN_VALUE, sum;

  public Statistics(double[] values) {
    for(int i=0; i<values.length; ++i) {
      double v = values[i];
      if (v < min) min = v;
      if (v > max) max = v;
      sum += v;
      nSamples++;
    }
  }
}
```

- The **new** operator does three things:
    1. allocates memory on the **Heap memory** to accommodate all of the Statistics object's fields;
    2. invokes a constructor so that the object can initialize itself; and
    3. returns a reference to the new object that we can store in a local variable.

Click here for more information about Objects mechanism and Heap memory.

4a. Class Members - Constructors

- Constructors are special methods that are invoked <u>only</u> when an object is created using the **new** operator.
- Constructors are similar to methods, except:
  - They have **NO** return type (not even *void*).
  - They always have the **same name as the class**.
  - They **CANNOT** be directly invoked, except from another constructor.
- If a class does not declare any constructors, the compiler will **automatically generate a default constructor** – *a public constructor that accepts no arguments.*
- A class can declare multiple constructors as long as each constructor has a unique parameter list. This is called ***constructor overloading*.**
- Best practice: Classes should have at least two constructors; one is without argument, and the other is with argument.

Example - Constructors

- **Constructor for our Statistics class:**
  - This constructor is public – it is accessible everywhere.
  - The constructor has no **return** type.
  - The constructor has the same name as the class.
  - This constructor takes a single argument of type double**[]**.
- Within the constructor, we initialize all of our fields.
- When the constructor finishes, our object has meaningful *state*.

```java
public class Statistics
{
  private int nSamples;
  private double min = Double.MAX_VALUE,
      max = Double.MIN_VALUE, sum;
// Constructor for our Statistics class is shown here.
  public Statistics(double[] values) {
    for(int i=0; i<values.length; ++i) {
      double v = values[i];
      if (v < min) min = v;
      if (v > max) max = v;
      sum += v;
      nSamples++;
    }
  }
}
```

4b. Constructor Overloading

The technique of having two (or more) constructors in a class is known as ***constructor overloading***. A class can have multiple constructors that differ in the **parameters number** and/or **type of parameters**. It is not, however, possible to have two constructors with the exact same parameters.

```
public class Person{
private String name;
private int age;

public Person() {

}

public Person(String name) {
     this.name = name;
    / *here the code of the second constructor is run,
and the age is set to 0 */
}

public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

First Constructor

Second Constructor

Third Constructor

```
public static void main(String[] args) {
   Person paul = new Person("Paul", 24);
   Person eve = new Person("Eve");

    System.out.println(paul);
    System.out.println(eve);
  }
```

Lesson 2

Overview of Methods in Java

Learning Objectives:

By the end of lesson, learners should be able to:

- ❑ Describe Methods and Method Attributes.
- ❑ Explain Methods Overloading and the Return Types of methods.

Agenda/Outline

- Topic 5: Overview of Java Method.
  - Method Reusability.
  - Semantic Code Organization.
  - Avoiding Code Repetition.
  - Method Declarations.
  - Method Access Specifier.
  - Method Return Types.
  - Void Return Type.
  - Method Parameter Lists.
  - Method Overloading.
  - Identify Correct Overloads.
  - Use Informative Parameter Names.
  - What Makes Parameter Lists Unique?
  - Why Return Type Does Not Matter.
  - The Entry Point and the Call Chain.

5. Overview of Java Method

- A method is a named ***block of code.***
- A method can be called (invoked) by name from other methods.
- A method can define zero or more parameters, allowing it to be called with different options.
- A method can optionally return a value.
- Methods are also called ***subroutines, subprograms, or functions***.
  - Function is sometimes reserved for those methods that return a value.
- Methods can help organize code to make it more ***readable*** and ***reusable***.
- Each method has a unique ***signature*** based on its name and parameters.
  - A method's return type **is not** part of its signature.
- **Remember**: Methods are always a ***member of a class***.

Method Reusability

Suppose you have an application that defines a **spreadsheet** document, and you want to autosize a column so that it is wide enough to visually accommodate all of its content over every row. This involves examining each row of the column, assessing the width needed for that cell's content, and then keeping the maximum value found to use as the final column width.

That is a lot of code! And once you have written the code, it is not something that you want copied/pasted all over your application. It is ideal for placing in a method named ***autoFitColumn***. The method could take just one parameter – the integer index of the column to auto-fit. Now, you can re-use it whenever you need it:

```
document.autoFitColumn(1);
```

Semantic Code Organization

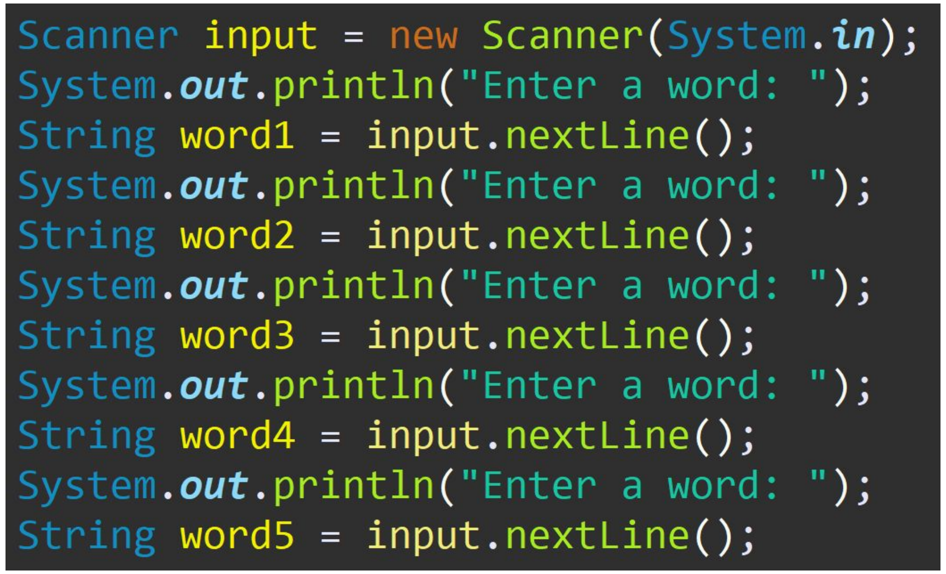
- Methods allow us to organize code into semantically meaningful units.
- This example is gaming code implemented *without* methods:

```java
System.out.println("You won!");
long gainExp = (long)Math.floor(Math.pow(200, (1 + enemy.getLevel()/10)));
this.experience += gainExp;
long threshold = (long)Math.floor(Math.pow(400, this.level/10));
long delta = threshold - this.experience;
if (delta <= 0) {
  this.level++;
  this.experience = -delta;
  System.out.println(this.getName() + " has levelled up.");
}
```

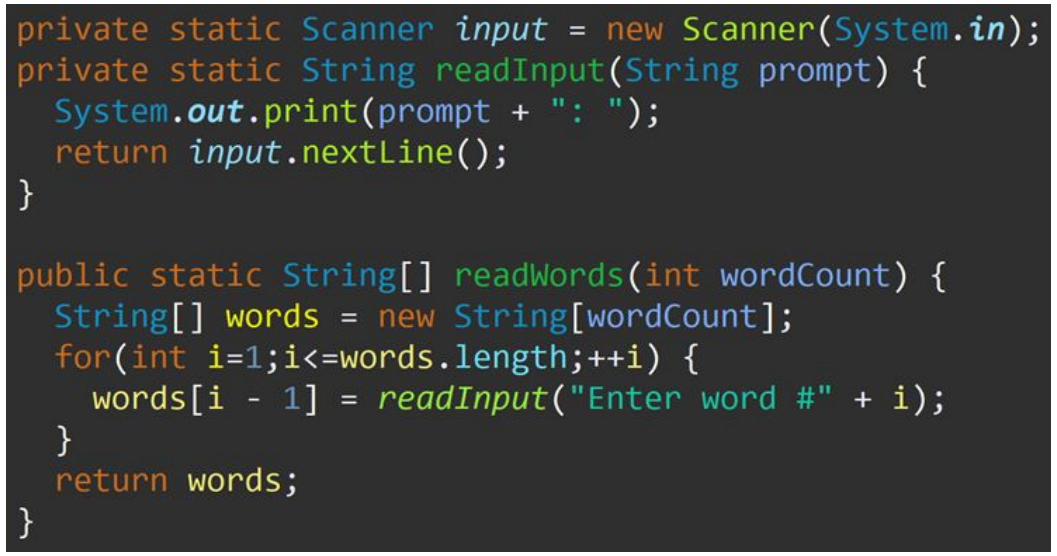
- This example is gaming code implemented *with* methods:

```java
System.out.println("You won!");
this.increaseExperience(enemy.calculateExperienceGiven(this.level));
```

Avoiding Code Repetition



```java
Scanner input = new Scanner(System.in);
System.out.println("Enter a word: ");
String word1 = input.nextLine();
System.out.println("Enter a word: ");
String word2 = input.nextLine();
System.out.println("Enter a word: ");
String word3 = input.nextLine();
System.out.println("Enter a word: ");
String word4 = input.nextLine();
System.out.println("Enter a word: ");
String word5 = input.nextLine();
```



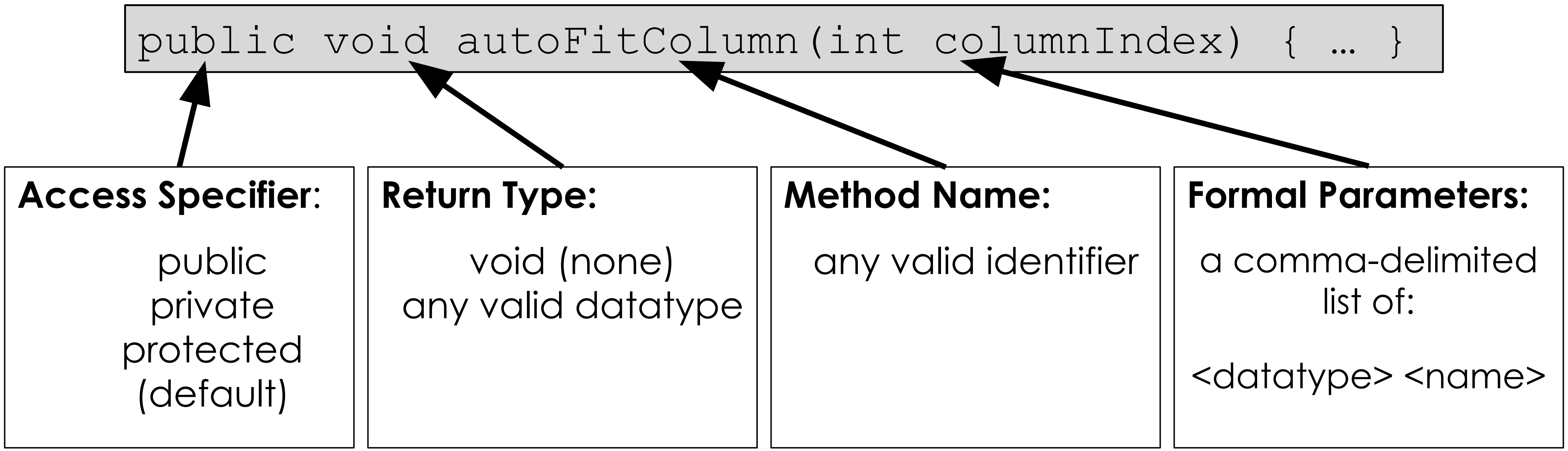
```java
private static Scanner input = new Scanner(System.in);
private static String readInput(String prompt) {
  System.out.print(prompt + ": ");
  return input.nextLine();
}

public static String[] readWords(int wordCount) {
  String[] words = new String[wordCount];
  for(int i=1;i<=words.length;++i) {
    words[i - 1] = readInput("Enter word #" + i);
  }
  return words;
}
```

- The repetitive code of the snippet on the left can be replaced by the method-based code on the right.
- In these examples, it is not necessarily less code, but it is definitely more versatile and reusable.

Method Declarations

- Here is the declaration of the method we created earlier:



```
public void autoFitColumn(int columnIndex) { … }
```

Access Specifier:

public
private
protected
(default)

Return Type:

void (none)
any valid datatype

Method Name:

any valid identifier

Formal Parameters:

a comma-delimited list of:

<datatype> <name>

- The **method body** is 0 or more statements contained within the curly braces following the declaration.

Method Declaration Examples

Examples of spreadsheet class' method declarations:

- **public boolean addWorksheet(String worksheetName)**;
- **public void autoFitRow(int startRowIndex, int endRowIndex)**;
- **public LocalDateTime getCellValueAsDateTime(int rowIndex, int columnIndex)**;
- **public double getCellValueAsDouble(String cellReference)**;
- **public double getColumnWidth(int columnIndex)**;
- **public List<String> getSheetNames()**;
- **public boolean isColumnHidden(String columnName)**;

🟥	Access Modifier
🟧	Return Type
⬛	Method Name
🟦	Parameter list / Arguments

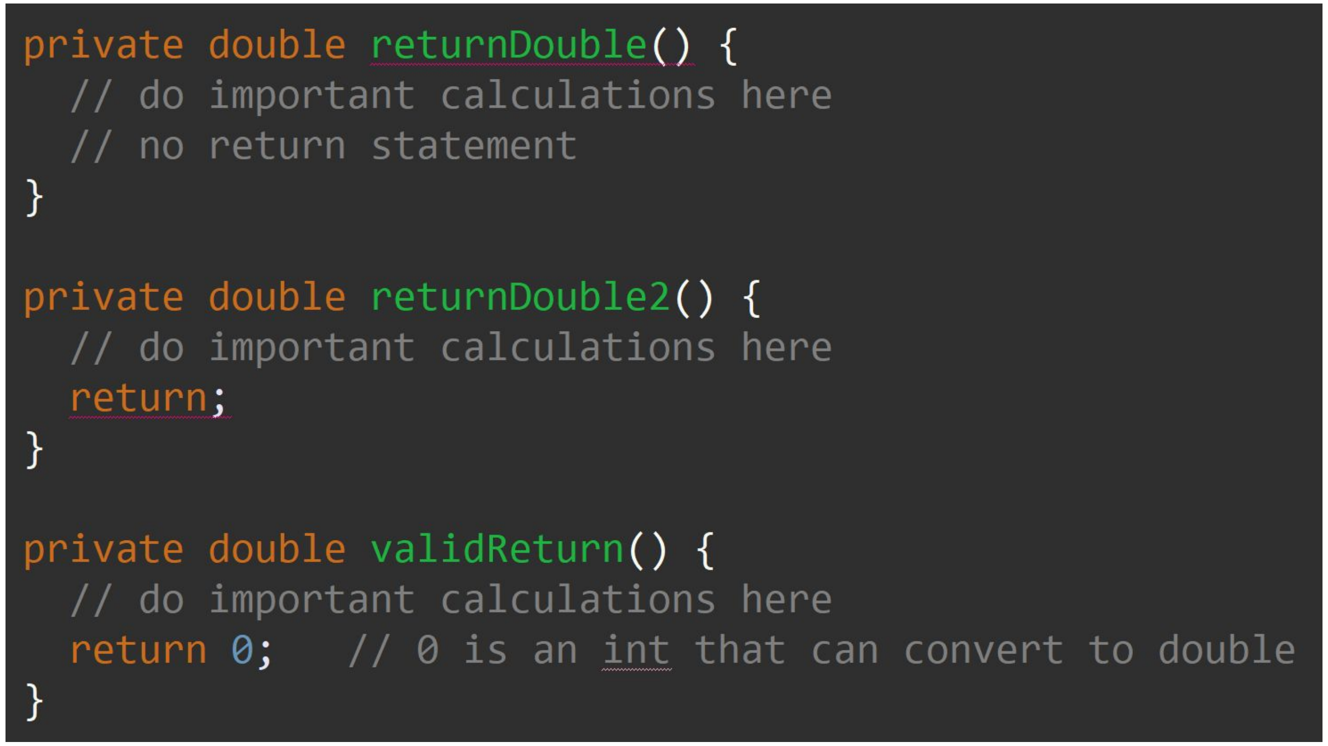
Method Access Specifiers

- A method's access specifier (also called access modifier) determines its visibility to consumers of a class.
- The access rules are the same as the rules for other class members:
  - **public** means that the method can be invoked by any consumer of the class.
  - **private** means that the method can be invoked only from within the class definition.
  - **protected** means that the method can be invoked from within the class or from within any subclass of the class.
  - **(default)** means that the method can be invoked from within the class or by any other class within the same package that contains this class.

Method Return Types

- All methods declare a return type.
- If a method declares a **non-void** return type, it ***must*** return a value of that data type.
  - A method can also return a value that can be automatically converted to the declared return type.

- **Challenge**:
  - Which of these method declarations is valid?



```
private double returnDouble() {
  // do important calculations here
  // no return statement
}

private double returnDouble2() {
  // do important calculations here
  return;
}

private double validReturn() {
  // do important calculations here
  return 0;    // 0 is an int that can convert to double
}
```

Void Return Type

- If a method does not need to return a value, it declares a return type **void**.
- If method performs a task but it does not give anything back. It is said to be **void** of a return value.
- Returning anything from a method with a **void** return type leads to a compile error.

```
public void printName()
{
  System.out.println("My name is James");
}


public void printNameTwo()
{
     return "My name is James";
}


public void printNameThree()
{
    return 0;
}
```

- This is an error:

Method Parameter Lists

- A method's parameters are enclosed in parentheses.
- Methods can declare 0 or more parameters:
  - 0 parameters: **`public void doSomethingSpecial() { … }`**
  - 1 parameter: **`public void doSomethingSpecial(int count) { … }`**
  - 2 parameters: **`public void doSomethingSpecial(int count, double value) { … }`**
  - 3 parameters: **`public void doSomethingSpecial(int count, double value, int options) { … }`**
- Each of these methods has the same name and return type (void).
- Each of these methods has a unique parameter list.
- When methods have the same name, they <u>must</u> have unique parameters.
  - This technique is called ***overloading***.
  - The return-type is irrelevant.

Method Overloading

- A class can define multiple methods with the same name.
- Each method with the ***same name*** must have a ***unique parameter list***.
  - This is a valid method overload:

```
public void talkToMe(String statement) {
      // TO DO
}
public void talkToMe(String statement, double confidence) {
      // TO DO
}
```

- Based on the methods' unique signatures, the compiler can always figure out exactly which method to call.

Method Overloading (continued)

- This is an example of invalid method overloading:

```
public void talkToMe(String statement, int count) {
      // TO DO
}
public void talkToMe(String testimony, int witnessCount) {
      // TO DO
}
```

- To the compiler, these two methods are identical:
  - The <u>names of the parameters do not matter</u>.
  - The compiler only sees the <u>parameter types</u> and their <u>order</u>.

Identify Correct Overloads

Which of these methods is unique from the perspective of the compiler? Assume the first-declared method is correct.

```
public void gotoTheMoon(int dayCount) { ... }

public int gotoTheMoon(int daycount) { ... }

public void gotoTheMoon(int dayCount, boolean countWeekends) { ... }

public int gotoTheMoon(int startDay, int dayCount) { ... }

public void gotoTheMoon(int firstDay, int numberOfDays) { ... }
```

Identify Correct Overloads (continued)

- Which of these overloads is correct?

```
void run() { ... }
int run(int n) { ... }
void run(double d) { ... }
void run(int n, double d) { ... }
void run(double d, int n) { ... }
void run(Window w) { ... }
```

- Although these overloads are valid, they are not well-named.
  - When overloading methods, use parameter names that clearly indicate what the overload does compared to the other overloads.

Use Informative Parameter Names

- These overloads use better parameter names:

```
void run(Window mainWindow) { ... }
void run(Window mainWindow, boolean runInBackground) { ... }
void run(Window mainWindow, StartupOptions startupOptions);
```

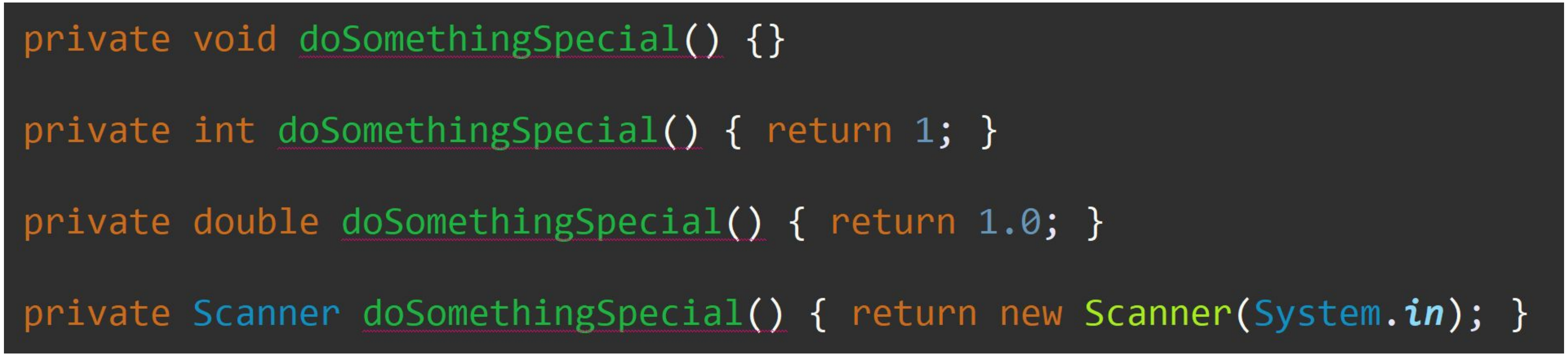
- This same advice applies even when you are not overloading a method!

What Makes Parameter Lists Unique?

- When the number of parameters is different.
  - `method()` is different from `method(int count)`
- When the number of parameters is the same but the parameter types are different:
  - `method(int value)` is different from `method(double value)`
- When the number of parameters is the same and the types are the same, but the types are in a different order:
  - `method(int count, double value) is different from method(double value, int count)`
- When parameter *names* do not matter (for example, these parameter lists are identical:
  - `method(int n, double v)` and `method(int count, double value)`

Why *Return* Type Does Not Matter

- A method's return type is not part of its signature/arguments.
- Consider the below example. Within a single class, these methods are ***NOT*** unique:



```
private void doSomethingSpecial() {}

private int doSomethingSpecial() { return 1; }

private double doSomethingSpecial() { return 1.0; }

private Scanner doSomethingSpecial() { return new Scanner(System.in); }
```

- A method's signature is composed only of its name and its parameters.
- The validity of overloading is based only on the ***signature***.

Why *Return* Type Does Not Matter (continued)

- A method's return type is not part of its signature because the caller of a method is not obligated to use the return value.
- Given these methods defined in one class:

```
public static double sayWhatMattersMost() {
  // TODO
  return 42.0;
}
```

```
public static int sayWhatMattersMost() {
  // TODO
  return 42;
}
```

- How does the compiler choose which method to call here?

```
public static void talkToMe() {
  sayWhatMattersMost();
}
```

The Entry Point and the Call Chain

- The statements within a method do not execute unless the method is explicitly invoked.
- In any program, the process starts when the main method is invoked by the Java Virtual Machine (JVM).
  - The main method is the JVM's ***entry point***.
- The main method can invoke other methods, and those methods can invoke other methods, forming a **call chain**.
- Declaring and implementing a method does not guarantee the method will run.
  - The method must be called from somewhere by another method that is called from somewhere, etc.
  - Methods that are never called from anywhere are called *dead code*.

Static Code Analysis

- Suppose you created an application that grows trees.
  - You have experimented repeatedly with the algorithm that figures out where to best plant the trees.
  - The less-successful of your experiments:
    - You have moved on from those lines of inquiry.
  - Yet, the code is still a formal part of your application:
    - It cannot be called because it is not within any possible call-chain.
    - It is never invoked from anywhere.
- Static code analyzers can discover un-callable code (dead code).
- Eclipse will (by its default settings) help identify methods that are outside of the application's call chain.

Knowledge Check

1. What is an object?
2. What is a class?
3. What is method overloading?
4. What are ‘access specifiers’?
5. What is the difference between Class variables and Local variables?
6. What is a constructor?

TEKsystems
Own change

Summary

Java is an OOP language. Everything in Java is associated with classes and objects, along with its attributes and methods. For example, a car is an object, which has attributes such as weight and color; and use methods, such as drive and brake.

A class is like an object constructor — a *blueprint* for creating objects:

- Modifiers - can be public or have default access.
- Class keyword - used to create a class.
- Class name - begins with an initial letter (capitalized by convention).
- Class parent (superclass) - precedes the keyword extend. A class can only extend (subclass) one parent.
- Interfaces (if any) - comma-separated list of interfaces implemented by the class, if any, and preceded by the keyword implements. A class can implement more than one interface.
- Body - class body surrounded by braces { }.

Questions?

TEKsystems
Own change