

AP Computer Science A

UNIT 2 TOPIC 2 Overloading Methods



1. Do Now: Warm Up in google classroom!

CollegeBoard Standards

Unit 2 Topic 3

2.3 Calling a Void Method

1.C Determine code that would be used to interact with completed program code.

3.A Write program code to create objects of a class and call methods.

ENDURING UNDERSTANDING

MOD-1

Some objects or concepts are so frequently represented that programmers can draw upon existing code that has already been tested, enabling them to write solutions more quickly and with a greater degree of confidence.

LEARNING OBJECTIVE

MOD-1.E

Call non-static void methods without parameters.

ESSENTIAL KNOWLEDGE

MOD-1.E.1

An object's behavior refers to what the object can do (or what can be done to it) and is defined by methods.

MOD-1.E.2

Procedural abstraction allows a programmer to use a method by knowing what the method does even if they do not know how the method was written.

MOD-1.E.3

A method signature for a method without parameters consists of the method name and an empty parameter list.

MOD-1.E.4

A method or constructor call interrupts the sequential execution of statements, causing the program to first execute the statements in the method or constructor before continuing. Once the last statement in the method or constructor has executed or a return statement is executed, flow of control is returned to the point immediately following where the method or constructor was called.

LEARNING OBJECTIVE

MOD-1.E

Call non-static void methods without parameters.

ESSENTIAL KNOWLEDGE

MOD-1.E.5

Non-static methods are called through objects of the class.

MOD-1.E.6

The dot operator is used along with the object name to call non-static methods.

MOD-1.E.7

Void methods do not have return values and are therefore not called as part of an expression.

MOD-1.E.8

Using a `null` reference to call a method or access an instance variable causes a `NullPointerException` to be thrown.

CollegeBoard Standards

Unit 2 Topics 4 & 5

2.4 Calling a Void Method with Parameters

2.C Determine the result or output based on the statement execution order in a code segment containing method calls.

3.A Write program code to create objects of a class and call methods.

2.5 Calling a Non-void Method

1.C Determine code that would be used to interact with completed program code.

3.A Write program code to create objects of a class and call methods.

ENDURING UNDERSTANDING

MOD-1

Some objects or concepts are so frequently represented that programmers can draw upon existing code that has already been tested, enabling them to write solutions more quickly and with a greater degree of confidence.

LEARNING OBJECTIVE

MOD-1.F

Call non-static void methods with parameters.

ESSENTIAL KNOWLEDGE

MOD-1.F.1

A method signature for a method with parameters consists of the method name and the ordered list of parameter types.

MOD-1.F.2

Values provided in the parameter list need to correspond to the order and type in the method signature.

MOD-1.F.3

Methods are said to be overloaded when there are multiple methods with the same name but a different signature.

ENDURING UNDERSTANDING

MOD-1

Some objects or concepts are so frequently represented that programmers can draw upon existing code that has already been tested, enabling them to write solutions more quickly and with a greater degree of confidence.

LEARNING OBJECTIVE

MOD-1.G

Call non-static non-void methods with or without parameters.

ESSENTIAL KNOWLEDGE

MOD-1.G.1

Non-void methods return a value that is the same type as the return type in the signature. To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression.

Projects Returned!

Overall, excellent work! Class average was 9.9 out of 10

Feedback is included at the end of your document, along with your rubric score.

If you missed any points, you should fix and resubmit for half credit back! Fix code, upload a revised version to GitHub, then *resubmit* assignment in Google Classroom to let me know it's ready for regrading (due on Tuesday)

Do Now: Warm Up

1. T/F: all constructors in a class are methods

TRUE!

"Constructors" are just special methods that create and return the object. But they *are* methods!

(you can tell a method is a constructor by the fact that it's named the same as the class, and has no apparent return type shown, although the return type is actually the

```
public class Bike {  
  
    // instance variables  
    private String model;  
    private int speed;  
  
    // constructor  
    public Bike(String model, int speed) {  
        this.model = model;  
        this.speed = speed;  
    }  
}
```

2. T/F: all methods in a class are constructors

FALSE!

Methods that are not the special constructors are simply "all other methods" which include getters, setters, and any other methods in the class.

Non-constructor methods all have lowercase names and explicit return types (or void)

```
// "setter" methods, for updating values stored in a
// Bike object's instance variables
public void setModel(String newModel) {
    model = newModel;
}

public void setSpeed(int newSpeed) {
    speed = newSpeed;
}

public String bikeInfo() {
    return "model: " + model + ", speed: " + speed;
}
```

3. What would you say is the purpose of a constructor?

- A constructor does **two** things:
 - It **initializes** (assigns initial values to) the values of **instance variables** to the values passed in as parameters or to default starting values:

```
public class Student {  
    /* Instance Variables */  
    private String firstName;  
    private String lastName;  
    private int gradYear;  
    private double accumulatedTestScores;  
    private int testScoreCount;  
  
    /* Constructor; see note below */  
    public Student(String firstName, String lastName, int gradYear) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.gradYear = gradYear;  
        accumulatedTestScores = 0.0;  
        testScoreCount = 0;  
    }  
}
```


3. What would you say is the purpose of a constructor?

- A constructor does **two** things:
 - It **initializes** (assigns initial values to) the values of instance variables to the values passed in as parameters or to default starting values:

```
public class Student {  
    /* Instance Variables */  
    private String firstName;  
    private String lastName;  
    private int gradYear;  
    private double accumulatedTestScores;  
    private int testScoreCount;  
  
    /* Constructor; see note below */  
    public Student(String firstName, String lastName, int gradYear) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.gradYear = gradYear;  
        accumulatedTestScores = 0.0;  
        testScoreCount = 0;  
    }  
}
```

Student is actually the return type because a Student object gets passed back to your program

○ It actually **returns** the fully initialized object back to your program when you use the new keyword: `Student dan = new Student("Dan", "Smith", 2023)`

4. What would you say is the purpose of a class?

- The purpose of a class is to organize a program's data and functionality.

4. What would you say is the purpose of a class?

- The purpose of a class is to organize a program's data and functionality.
- A class serves as a "cookie cutter" or "blueprint" from which we can create multiple objects (instances) of that class, effectively reusing code that stores information (attributes/instance variables) and performs functions (methods)

4. What would you say is the purpose of a class?

- The purpose of a class is to organize a program's data and functionality.
- A class serves as a "cookie cutter" or "blueprint" from which we can create multiple objects (instances) of that class, effectively reusing code that stores information (attributes/instance variables) and performs functions (methods)
- Imagine a program that tracks all 6000 students at BTHS. We *could* write a program that contains 6000 variables for each student's name, 6000 *more* variables for each student's test average, 6000 *more* variables for each student's test count, and then copy/paste the code for each calculations 6000 times --- **UGH!!! THIS WOULD BE AWFUL!** **Instead**, we could create 6000 Student objects, store them in a list ("array," as we will learn), and each object has its own copy of the instance variables and methods. **Much more efficient!**

Today

- Quick topic: Method Overloading
- Finish up U2T2 Lab 2 Part 2 Student Program from Tuesday (due end of period)
 - We will discuss a sample solution halfway through the period
- **Tip Calculator: REFACTORED!** Refactor your Tip Calculator Project so that it moves functionality into a class (similar to how the Student class works in your Student Program)
 - "refactor" just means to *rewrite code*, typically to make it more efficient or to better follow good object oriented programming practices!

Method Signatures & Overloading Methods

Overloading Constructors

- As we have learned, we can **overload** constructors (have more than one):

```
public Bike(String model, int speed) {  
    this.model = model;  
    this.speed = speed;  
}  
  
// another constructor that assigns a default value to speed  
public Bike(String model) {  
    this.model = model;  
    speed = 15; // default value  
}  
  
// a no-parameter constructor that assigns default values to model and speed  
public Bike() {  
    model = "Mountain Bike"; // default value  
    speed = 15; // default value  
}
```

Overloading Constructors

- As we have learned, we can **overload** constructors (have more than one):

```
public Bike(String model, int speed) {  
    this.model = model;  
    this.speed = speed;  
}  
  
// another constructor that assigns a default value to speed  
public Bike(String model) {  
    this.model = model;  
    speed = 15; // default value  
}  
  
// a no-parameter constructor that assigns default values to model and speed  
public Bike() {  
    model = "Mountain Bike"; // default value  
    speed = 15; // default value  
}
```

- How do the constructor's **method headers** seem to differ?

Overloading Constructors

- As we have learned, we can **overload** constructors (have more than one):

```
public Bike(String model, int speed) {  
    this.model = model;  
    this.speed = speed;  
}  
  
// another constructor that assigns a default value to speed  
public Bike(String model) {  
    this.model = model;  
    speed = 15; // default value  
}  
  
// a no-parameter constructor that assigns default values to model and speed  
public Bike() {  
    model = "Mountain Bike"; // default value  
    speed = 15; // default value  
}
```

- How do the constructor's **method headers** seem to differ? **the number of parameters each has (2, 1, and none)**

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor **methods** by giving them the **same name** but *different* overall method **signatures**
- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor methods by giving them the **same name** but *different* overall **method signatures**
- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

The **method signature** of this methods is: `add(int, int, double)` → only a method's name, parameter *types*, and parameter order are part of its **signature**

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor methods by giving them the **same name** but *different* overall **method signatures**
- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

The **method signature** of this methods is: `add(int, int, double)` → only a method's name, parameter *types*, and parameter order are part of its **signature**

- What is in the complete **method header** that is **not** part of the **method signature**?

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor methods by giving them the **same name** but *different* overall **method signatures**
- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

The **method signature** of this methods is: `add(int, int, double)` → only a method's name, parameter *types*, and parameter order are part of its **signature**

- What is in the complete **method header** that is **not** part of the **method signature**?
 - A method's **return type** and the **parameter names** are NOT part of a method signature, nor is the “public” modifier

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor methods by giving them the **same name** but *different* overall **method signatures**

- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

The **method signature** of this methods is: `add(int, int, double)` → only a method's name, parameter *types*, and parameter order are part of its **signature**

- What is in the complete **method header** that is **not** part of the **method signature**?
 - A method's **return type** and the **parameter names** are **NOT** part of a method signature, nor is the “public” modifier
- Java requires all constructor/method signatures to be **UNIQUE**, i.e. no two can be the same. *Two methods with identical method signatures will not be allowed (and will not compile)!*

Method Signatures & Overloading

- Similarly, you can **overload** non-constructor methods by giving them the **same name** but *different* overall **method signatures**

- **Example:** If you have a method with a **method header** that looks like this:

```
public int add(int num1, int num2, double num3)
```

The **method signature** of this methods is: `add(int, int, double)` → only a method's name, parameter *types*, and parameter order are part of its **signature**

- What is in the complete **method header** that is **not** part of the **method signature**?
 - A method's **return type** and the **parameter names** are NOT part of a method signature, nor is the “public” modifier
- Java requires all constructor/method signatures to be **UNIQUE**, i.e. no two can be the same. *Two methods with identical method signatures will not be allowed (and will not compile)!*
- **Overloaded methods** have the same **name** but unique (different) **signatures**, and thus differ in regard to the **parameters**: how **many** there are, what **type** each is, and what **order** they are

Method Overloading

10 minutes! Work through the Method Overriding Warm Up

Calling Overloaded Methods

```
Adder adder = new Adder("My Adder", 10);  
adder.incrementCounter(5);  
adder.incrementCounter();  
adder.incrementCounter(5.0);  
adder.add(7, 5.0);  
adder.add(6, 8, 10);  
adder.add(7.0, 5.0);  
adder.add(7.0, 5);  
adder.add(7, 5);  
adder.add("hello", 5, 6);
```

How do you think Java knows which “version” of an overloaded method or constructor to execute?

Calling Overloaded Methods

```
Adder adder = new Adder("My Adder", 10);  
adder.incrementCounter(5);  
adder.incrementCounter();  
adder.incrementCounter(5.0);  
adder.add(7, 5.0);  
adder.add(6, 8, 10);  
adder.add(7.0, 5.0);  
adder.add(7.0, 5);  
adder.add(7, 5);  
adder.add("hello", 5, 6);
```

➡ add(double, int)

When you make a method call to an **overloaded** method, since it has the same name as at least one other method, Java looks at the *number* of arguments you've provided **and** their *types in order* to determine which “version” of that method to execute.

```
public int add(int num2, int num1)  
{  
    return num1 + num2;  
}  
  
public int add(double num1, int num2)  
{  
    int casted = (int) (num1 + num2);  
    return casted;  
}  
  
public int add(int num1, double num2)  
{  
    int casted = (int) (num1 + num2);  
    return casted;  
}
```

Calling Overloaded Methods

```
Adder adder = new Adder("My Adder", 10);
adder.incrementCounter(5);
adder.incrementCounter();
adder.incrementCounter(5.0);
adder.add(7, 5.0);
adder.add(6, 8, 10);
adder.add(7.0, 5.0);
adder.add(7.0, 5);
adder.add(7, 5);
adder.add("hello", 5, 6);
```

add(double, int)

```
public int add(int num2, int num1)
{
    return num1 + num2;
}
```


```
public int add(double num1, int num2)
{
    int casted = (int) (num1 + num2);
    return casted;
}
```

```
public int add(int num1, double num2)
{
    int casted = (int) (num1 + num2);
    return casted;
}
```

When you make a method call to an **overloaded** method, since it has the same name as at least one other method, Java looks at the *number* of arguments you've provided **and** their *types in order* to determine which “version” of that method to execute.

Calling Overloaded Methods

```
Adder adder = new Adder("My Adder", 10);  
adder.incrementCounter(5);  
adder.incrementCounter();  
adder.incrementCounter(5.0);  
adder.add(7, 5.0);  
adder.add(6, 8, 10);  
adder.add(7.0, 5.0);  
adder.add(7.0, 5);  
adder.add(7, 5);  
adder.add("hello", 5, 6);
```



`add(String, int, int)`

And if there *isn't* a method with the matching signature, Java will let you know!

```
AdderClient.java:14: error: incompatible types: String cannot be converted to int  
    adder.add("hello", 5, 6);  
              ^
```

Agenda

1. Finish up the **U2T2 Lab 2 Part 2 Student Program** (due end of today)
 - We will discuss a sample solution about half way through
1. Tip Calculator Project corrections for half points back!
 - Fix code, upload a revised version to GitHub, then **resubmit** assignment in Google Classroom (due on Tuesday)
1. Begin the **Tip Calculator REFACTORED!** assignment (due on Tuesday)
 - You will **refactor** your Tip Calculator project to have two classes in a similar setup to how you wrote the Student Program.
 - **This is meant to be a challenge, so do your best on it! It will be practice for when we get to later units this year on class writing and class design** 🤖

"Refactor" is a computer science term that means to **"rewrite code,"** typically to make the code more efficient or to better follow object-oriented programming best practices.

Jupiter

Mr. Miller is working to get grades uploaded to Jupiter and should have it updated later this weekend -- thank you for your patience!

Before you go

Please RESTART your laptop

Start → Power → Restart

(leave lid open)