Revised: 4/21/2020

Objects and Classes I

Abstraction and Encapsulation

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Basics of Object Oriented Design
  - Objects
  - Classes
- Instance Variables/Fields
- Constructors
- Access Modifiers
- Methods
  - Mutator Methods
  - Accessor Methods
  - Utility Methods
- Class Diagrams
- Overloaded Methods

Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **Consolas**
Output – Courier New

What is Object Oriented Programming?

- A programming paradigm where software is written so that a program functions as a system of objects.
  - A different way of designing software, as opposed to the procedural programs we've been writing all semester.

- The objects interact with each other to complete the program's tasks.

- Software objects contain information about themselves and allow interaction with other objects.

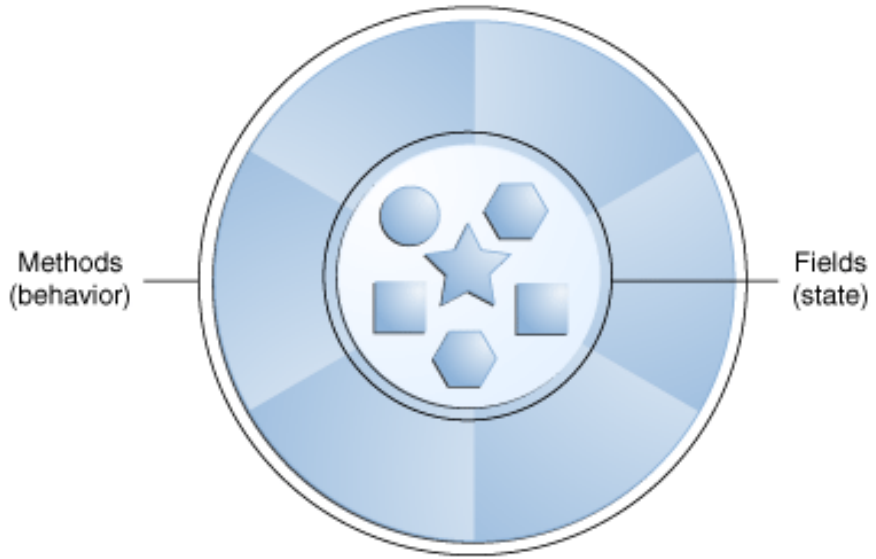
What are Objects?

- A software object is *conceptually* similar to a real world object.
- Real world objects all have two characteristics:
  - They have ***attributes***- properties that make something unique.
    - A bicycle's attributes could be its speed, color, tire size, etc.
  - They have ***behaviors***- actions that it can do.
    - A bicycle's behaviors could be pedaling, braking, changing gear, etc.

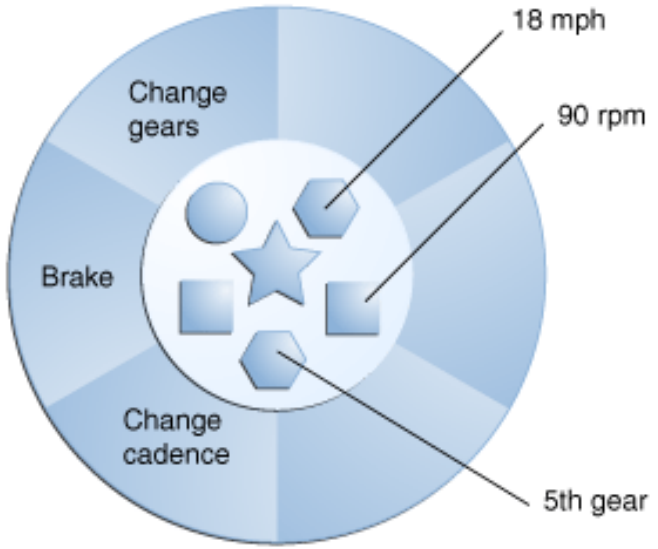
- When we model a software object, it too has attributes and behaviors.
  - Objects store their attributes in variables referred to as ***fields.***
  - Objects expose their behaviors as ***functions.***

A Bicycle Object

Software Object



Bicycle Object



Why apply Object Oriented Programming?

- Modularity
  - The code for an object is written and maintained separately from the code of other objects.
- Information Hiding
  - By interacting with the object's methods, the details of its internal implementation is hidden.
- Reusability
  - You can create multiple instances of an object.

Abstraction

- ***Abstraction*** is an OOP principle that software objects are able to function as individual entities.

- Using fields, the object can hold information about itself.

- Using methods, the object can perform various actions and operations and communicate with other objects.

Encapsulation

- ***Encapsulation*** is an OOP principle that suggests we design objects so that all relevant data (attributes) and behaviors (methods) are together.
  - The object also controls how its information is seen or changed.

- Only attributes and behaviors relevant to the object should be in the object.
  - Other data or functions not related to the object's use should be placed in other objects.
  - For example, it wouldn't make sense for a Bicycle object to have a fuel level attribute (but would, perhaps, make sense in a Moped object.)

What are Classes?

- A ***class*** is the blueprint from which objects are created.

- It is the *source code* of the object.
  - The object is the idea, the code in the class is the implementation of the idea.

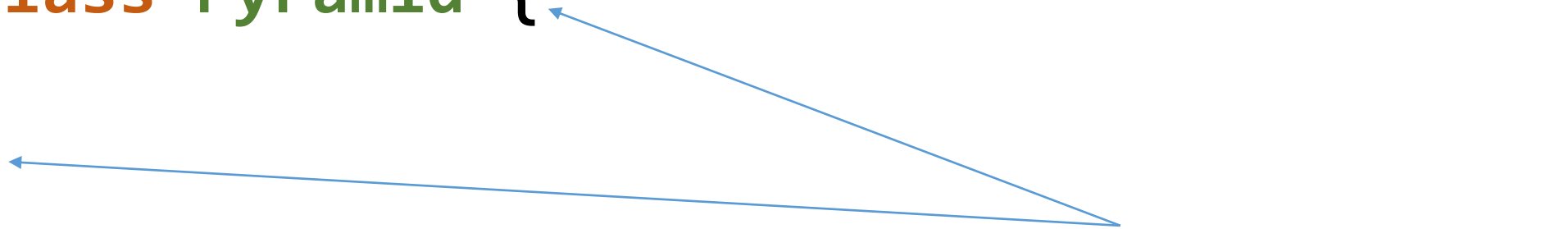
- When a new software object is created from a class, this is referred to as instantiation.
  - *Creating a new instance of an object.*
  - *Instantiating an object.*

Classes

- ***Class Declaration*** (or ***Class Header***) for an object named “Pyramid” is shown below.

```
class Pyramid {

}
```



Instance Variables

- An ***instance variable*** is a field that is accessible by the methods of the class.
  - It is declared inside the class, but outside of any method.

- The data stored in an instance variable is unique to each instance of an object.

Declaring Instance Variables

```
class Pyramid {

   int width;
   int length;
   int height;
   String color;

}
```

- Declared just as we have seen before, but not within any particular method.
- Can be initialized at declaration or in a method.

Creating an Instance of an Object

- In a second class (and in a separate source code file) named PyramidTest, we will ***instantiate*** a Pyramid object in its main method (shown below).
  - ***Instantiation*** is the term used when you create an ***instance*** of an object.

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

    }
}
```

The Default Constructor

- Objects are instantiated using a special method called a ***constructor***.
- Constructors are used to “set up” or *construct* an instance of an object.
- When there are no constructors present in a class, the compiler automatically adds a ***default constructor***.
  - This guarantees every object has a constructor.

```
class Pyramid {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

    }
}
```

Accessing an Instance's Fields

- We can access an instance's fields using dot notation.

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

        //Sets the object's width attribute to 3
        example.width = 3;

    }
}
```

Accessing an Instance's Fields

- We can access an instance's fields using dot notation.

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

        //Sets the object's width attribute to 3
        example.width = 3;
        System.out.println("The pyramid's width is " + example.width);

    }
}
```

```
The pyramid's width is 3
```

Constructors

- Constructors are a special type of method/behavior that prepares the instance of the object.

- Constructors almost always have the same name as the class file.
  - Not always the case as a single source code file can contain multiple classes.

- Classes may have multiple constructors.
  - This is referred to as ***overloading*** or having ***overloaded constructors***.

The No Argument (No-Arg) Constructor

```
class Pyramid {

    int width;
    int length;
    int height;
    String color;

    Pyramid() {
        width = 1;
        length = 1;
        height = 5;
        color = "White";
    }


}
```

- Replaces the default constructor added by the compiler.
  - A class only has a default constructor when the class has no constructors defined.
- The code in the no-arg constructor's body will be executed when the no-arg constructor is called.
- The names of any constructors must always match the class name.

Without the No-Arg Constructor

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

        System.out.println("The pyramid's width is " + example.width);
        System.out.println("The pyramid's length is " + example.length);
        System.out.println("The pyramid's height is " + example.height);
        System.out.println("The pyramid's color is " + example.color);
    }
}
```

```
The pyramid's width is 0
The pyramid's length is 0
The pyramid's height is 0
The pyramid's color is null
```

With the No-Arg Constructor defined

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example = new Pyramid();

        System.out.println("The pyramid's width is " + example.width);
        System.out.println("The pyramid's length is " + example.length);
        System.out.println("The pyramid's height is " + example.height);
        System.out.println("The pyramid's color is " + example.color);
    }
}
```

```
The pyramid's width is 1
The pyramid's length is 1
The pyramid's height is 5
The pyramid's color is white
```

Passing Values into a Constructor

- In many cases, you will want to pass data into a constructor to set the values of the fields.

- ***Parameters*** are variables that represent data that is given (or *passed*) to a constructor.
  - Data given to the constructor are called ***arguments***.

- If a constructor declares a parameter list, a value for each parameter must be present.
  - The list must include the data type for each parameter.

Constructors

```
Pyramid() {
    width = 1;
    length = 1;
    height = 5;
    color = "White";
}
```

No-Arg Constructor

Parameter

```
Pyramid(int widthIn) {
    width = widthIn;
    length = 1;
    height = 5;
    color = "White";
}
```

A Constructor with 1 Parameter

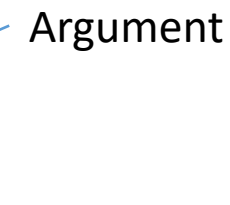
Constructors

- Now, we have a second way to instantiate an instance of a Pyramid object.

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example1 = new Pyramid();
        Pyramid example2 = new Pyramid(5);

        System.out.println("The first pyramid's width is " + example1.width);
        System.out.println("The second pyramid's width is " + example2.width);
    }
}
```




```
The first pyramid's width is 1
The second pyramid's width is 5
```

Constructors

```
Pyramid() {
    width = 1;
    length = 1;
    height = 5;
    color = "White";
}

Pyramid(int widthIn) {
    width = widthIn;
    length = 1;
    height = 5;
    color = "White";
}

Pyramid(int widthIn, int lengthIn) {
    width = widthIn;
    length = lengthIn;
    height = 5;
    color = "White";
}
```

No-Arg Constructor

A Constructor with 1 Parameter

A Constructor with 2 Parameters

Constructors

```
Pyramid(int widthIn, int lengthIn, int heightIn) {
    width = widthIn;
    length = lengthIn;
    height = heightIn;
    color = "White";
}

Pyramid(int widthIn, int lengthIn, int heightIn, String colorIn) {
    width = widthIn;
    length = lengthIn;
    height = heightIn;
    color = colorIn;
}
```

A Constructor with 3 Parameters

A Constructor with 4 Parameters

Constructors

- Now, there are 5 different ways to instantiate an instance of a Pyramid object.

```
class PyramidTest {

    public static void main(String[] args) {
        Pyramid example1 = new Pyramid();
        Pyramid example2 = new Pyramid(5);
        Pyramid example3 = new Pyramid(7, 7, 15, "Blue");

        System.out.println("The third pyramid's height is " + example3.height);
        System.out.println("The third pyramid’s color is " + example3.color);
    }
}
```

```
The third pyramid’s height is 15
The third pyramid’s color is Blue
```

Constructor Signatures

- There is no limit to the number of constructors in a class.
- However, each constructor must have a unique ***signature***.
  - A constructor signature consists of its name and parameter list data types.

```
Pyramid()
Pyramid(int widthIn)
Pyramid(int widthIn, int lengthIn)
Pyramid(int widthIn, int lengthIn, int heightIn)
Pyramid(int widthIn, int lengthIn, int heightIn, String colorIn)
```

Signatures:

Pyramid()
Pyramid(int)
Pyramid(int, int)
Pyramid(int, int, int)
Pyramid(int, int, int, String)

Constructor Signatures

- If our Pyramid class had the following constructors, it would not compile.
  - Their signatures are not unique.

```
Pyramid(int widthIn) {
    width = widthIn;
    length = 1;
    height = 5;
    color = "White";
}
```

→ Pyramid(int)

```
Pyramid(int lengthIn) {
    width = 1;
    length = lengthIn;
    height = 5;
    color = "White";
}
```

→ Pyramid(int)

```
Pyramid example = new Pyramid(5);
```

There's no way to know which constructor to use.

Access Modifiers

- Access Modifiers specify how classes, fields, and methods can be accessed by other objects.
  - This limits other objects from making changes to the data in the fields or using methods that pertain only to the object's internal implementation.

public

- Modifier that allows a field or method to be accessible to all other objects.

private

- Modifier that does not allow a field or method to be accessible to other objects.

Private Fields

```
public class Rectangle {

   private int length;
   private int width;

}
```

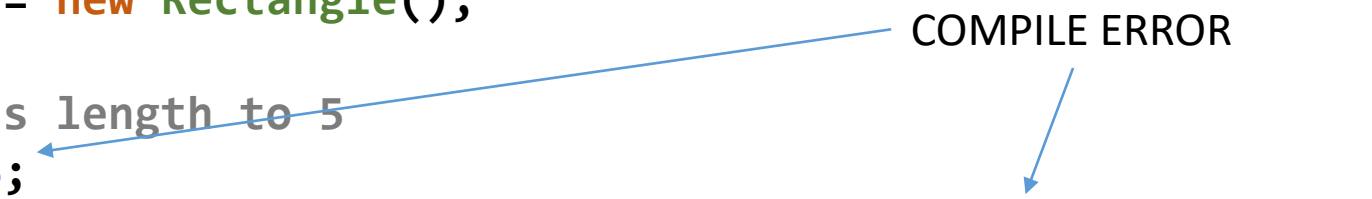
- This Rectangle class has two private fields.
- These fields cannot be accessed anywhere except from within the Rectangle class.

Private Fields

```
public class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();

        //Sets the object's length to 5
        example.length = 5;
        System.out.println("The third pyramid's height is " + example.length);
  }
}
```



- A private field is not accessible.
- The only way to change or retrieve the value of a field would be to use a *method*.

Methods

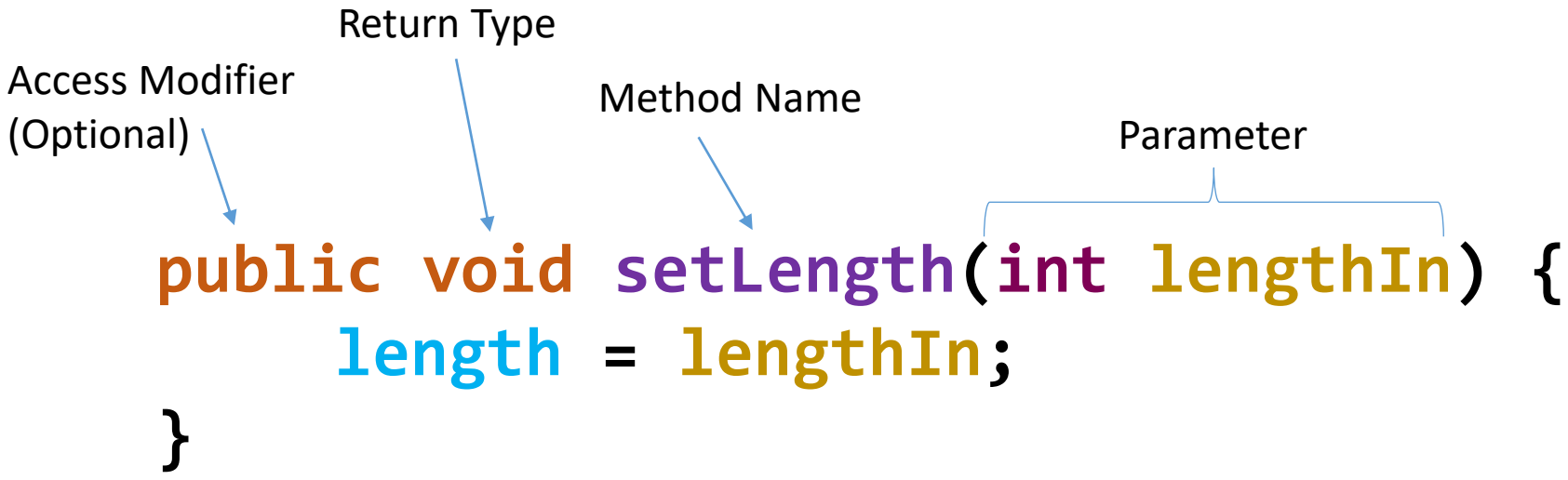
- A ***method*** is a subroutine that operates on an object's internal data.
  - After the method finishes executing its code, it may or may not return data.

- Most methods in an object are either a mutator method or an accessor method.

- A **mutator method** changes (or "sets") data in an object.
  - Colloquially called a "setter" method.
- An **accessor method** retrieves (or "gets") data from an object.
  - Colloquially called a "getter" method.

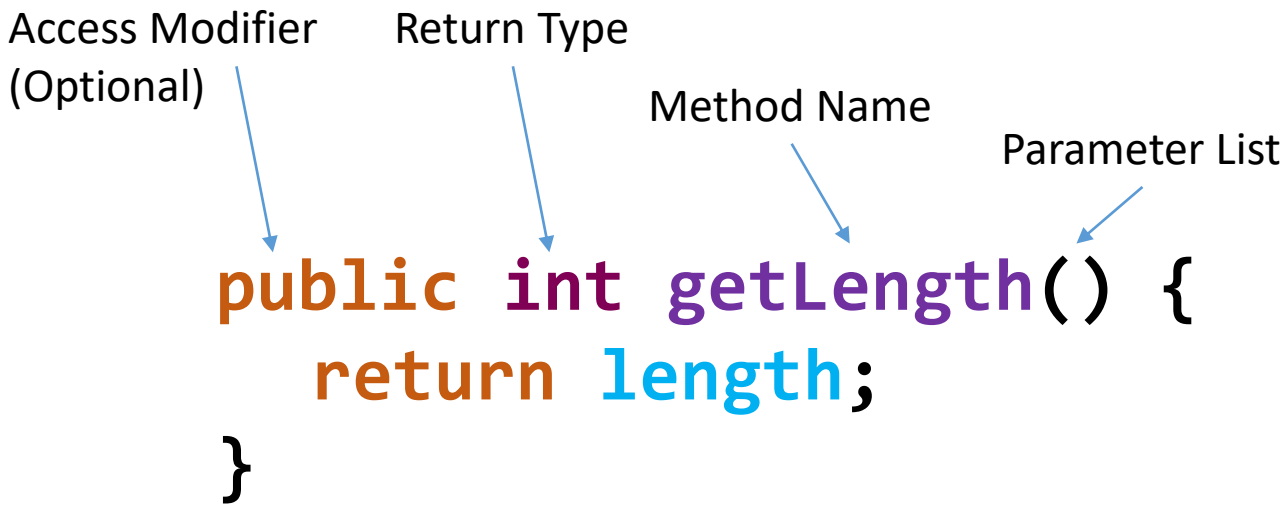
- The use of mutators and accessors allows the object to specify how (and if) its private data is accessed or changed.

Mutator Method (Length)



- This method will change the current value of the object's length field.
- A method that does not return data must have a **void** return type.
  - Mutators typically do not return data. They "set" data- they don't "get" data.

Accessor Method (Length)



- This method will return the current value of our gear instance variable.
- A ***return statement*** indicates the value that is returned by the method.
- The type of data returned must match the method's return type.
  - Since this method's return type is int, the method can only return an int value when called.

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();

        example.setLength(7);
        System.out.println("The rectangle's length is " + example.getLength());
    }

}
```

Argument

Parameter

```
public void setLength(int lengthIn) {
     length = lengthIn;
}

public int getLength() {
     return length;
}
```

```
The rectangle's length is 7
```

Accessor and Mutator Method (Width)

```
public void setWidth(int widthIn) {
   width = widthIn;
}

public int getWidth() {
   return width;
}
```

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();

        example.setLength(5);
        example.setWidth(10);
        System.out.println("The rectangle's length is " + example.getLength());
        System.out.println("The rectangle's width is " + example.getWidth());
    }

}
```

```
The rectangle's length is 5
The rectangle's width is 10
```

Checking Data

- Using mutators to change the state of the object gives the object control of its state.

- For example, we can add checks to the length and width field mutators to check for negative distances or distances of zero.
  - The object (through its methods) can decide what happens.

Checking Data

- Additional code can be added to the mutator to check the argument passed to it.
  - Now, a zero or negative argument will cause the length field to be assigned 1.

```
public void setLength(int lengthIn) {
   if(lengthIn <= 0) {
      length = 1;
   }
    else {
      length = lengthIn;
   }
}
```

Checking Data

- A similar check can be added in the mutator for the width field.

```
public void setWidth(int widthIn) {
   if(widthIn <= 0) {
      width = 1;
   }
    else {
      width = widthIn;
   }
}
```

Using Method Calls to set/get an Instance's Data

```
public class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();

        example.setLength(-66);
        example.setWidth(0);
        System.out.println("The rectangle's length is " + example.getLength());
        System.out.println("The rectangle's width is " + example.getWidth());
    }

}
```

```
The rectangle's length is 1
The rectangle's width is 1
```

Utility Methods

- Another type of method is a **utility method**.
  - These are methods that aren't wholly concerned with setting or getting data to/from an object, but perform some other operation or function.
  - A utility method may or may not return data.

- A Rectangle object might have a utility method to calculate the area or perimeter of the shape.

Utility Method (Area)

- This calculateArea method will calculate and return the area of the shape.

```
public int calculateArea() {
    int area = length * width;
    return area;
}
```

Utility Method (Perimeter)

- This calculatePerimeter method will calculate and return the perimeter of the shape.

```
public int calculatePerimeter() {
    int perimeter = 2 * (length + width);
    return perimeter;
}
```

Utility Methods

```
public class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();

        example.setLength(6);
        example.setWidth(5);
        int calc1 = example.calculateArea();
        int calc2 = example.calculatePerimeter();
        System.out.println("The rectangle's area is " + calc1);
        System.out.println("The rectangle's perimeter is " + calc2);
    }

}
```

```
The rectangle's area is 30
The rectangle's perimeter is 22
```

Method Naming

- By convention (in Java), method names are in camel-case.

- A method's name should always describe what it does.
  - Accessor methods normally start with "get".
    - For example: getValue, getName
  - Mutator methods normally start with "set".
    - For example: setValue, setName
  - Utility functions will vary in name.
    - For example: fixHeaders, validateData, updateValues, toSquareFeet
  - Methods that return a boolean normally start with "is" or "has".
    - For example: isFinished, isClosed, hasCorrectData

Method Access

- Accessor and mutator methods are normally public.
  - Other objects need to call these methods to interact with the object.

- Utility methods may or may not be public.
  - It depends on the object's design.
  - Make the method private if you don't want other objects to call it.

Method Signatures

- Like constructors, methods have signatures.
- A method signature consists of its name and parameter list data types.
- All methods in a class must have unique signatures.

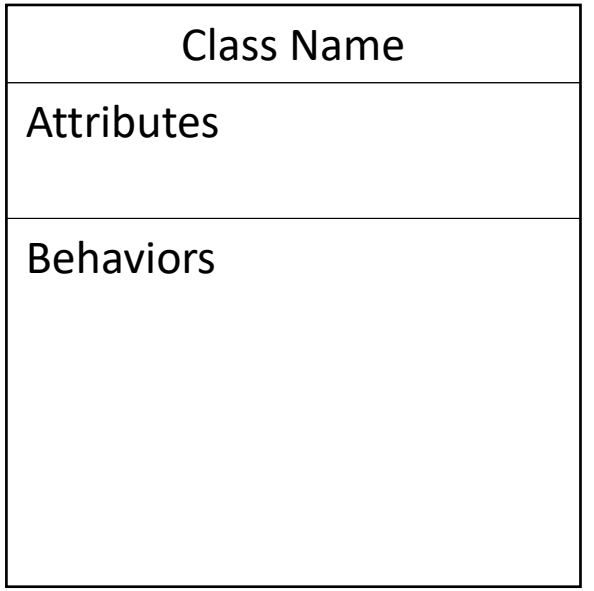
```
public void exampleMethod(int arg1, String arg2)
```
→ exampleMethod(int, String)

```
public int exampleMethod2()
```
→ exampleMethod2()

```
public String exampleMethod3(String[] arg1)
```
→ exampleMethod3(String[])

Class Diagrams

- Unified Modeling Language provides a set of standard diagrams for graphically depicting an object oriented system.
- In UML, each class is shown as a box, with three sections:
  - The Class Name
  - Class Attributes (Fields/Variables)
  - Class Behaviors (Constructors and Methods)



Class Diagrams

- When displaying fields (and parameter names in methods) in a class diagram, the format to use is:
  - name : type

- Access specifier symbols:
  - \+ public fields/methods
  - \- private fields/methods
  - None (or ~) No access specifier.

Class Diagram (Fields)

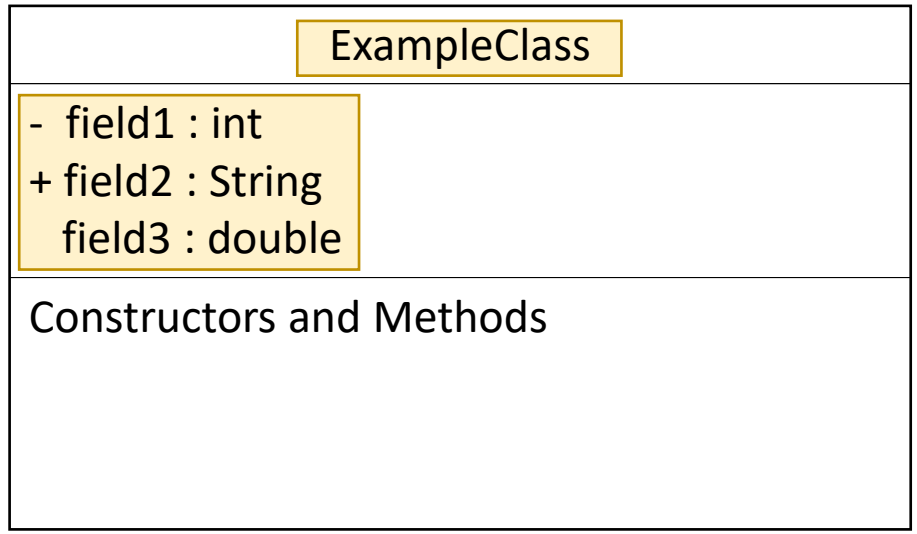
```
class ExampleClass {
    private int field1;
    public String field2;
    double field3;

    public ExampleClass(int argIn) {
        //Constructor code
    }

    public int method1(double arg1In, int arg2In) {
        //method1 code
    }

    private void method2() {
        //method2 code
    }

}
```



Class Diagrams (Constructors and Methods)

- When displaying a constructor in a class diagram, the format to use is:
  - name(arg : type, …)

- When displaying a method in a class diagram, the format to use is:
  - name(arg : type, …) : returnType

- Access modifier symbols used are the same as for fields.

Class Diagram (Constructors and Methods)

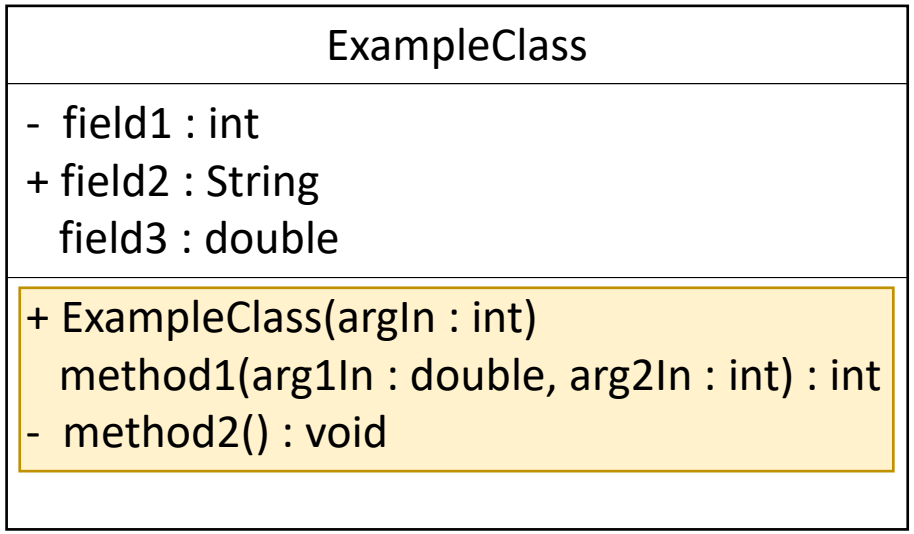
```
class ExampleClass {
    private int field1;
    public String field2;
    double field3;

    public ExampleClass(int argIn) {
        //Constructor code
    }

    int method1(double arg1In, int arg2In) {
        //method1 code
    }

    private void method2() {
        //method2 code
    }

}
```



Overloaded Methods

- A method is ***overloaded*** when two or more methods share the same name but have *different parameter types/lists*.

- No limit to the number of times a method can be overloaded.

```
public void setLength(int lengthIn) {
    if(lengthIn <= 0) {
        length = 1;
    }
   else {
        length = lengthIn;
    }
}

public void setLength(double lengthIn) {
    if(lengthIn <= 0) {
        length = 1;
    }
   else {
        length = (int)Math.round(lengthIn);
    }
}
```

Overloaded Methods

- Overloaded methods give the appearance of one method that handles multiple types of arguments.
  - Overloaded methods are also known as ***polymorphic methods*** for this reason.

```
class RectangleTest {

    public static void main(String[] args) {
        Rectangle example = new Rectangle();
        example.setLength(7);
        example.setLength(4.3); //Changes the length to 4.3 (rounded to 4)
    }
}
```

Overloaded Methods

- It's common practice to chain overloaded methods together.

```
public void setLength(int lengthIn) {
    if(lengthIn <= 0) {
        length = 1;
    }
    else {
        length = lengthIn;
    }
}

public void setLength(double lengthIn) {
    setLength((int)Math.round(lengthIn));
}

public void setLength(String lengthIn) {
    setLength(Integer.parseInt(lengthIn));
}
```