Advanced Object Oriented Programming

EECS2030

Section M

```
public class SimplePoint2 {
  public float x;
  public float y;

  /**
   * Sets the x and y coordinate of the point to the argument
   * values.
   *
   * @param x the x coordinate of the point
   * @param y the y coordinate of the point
   */
  public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
  }
```

this.x : the field named **x** of **this** point

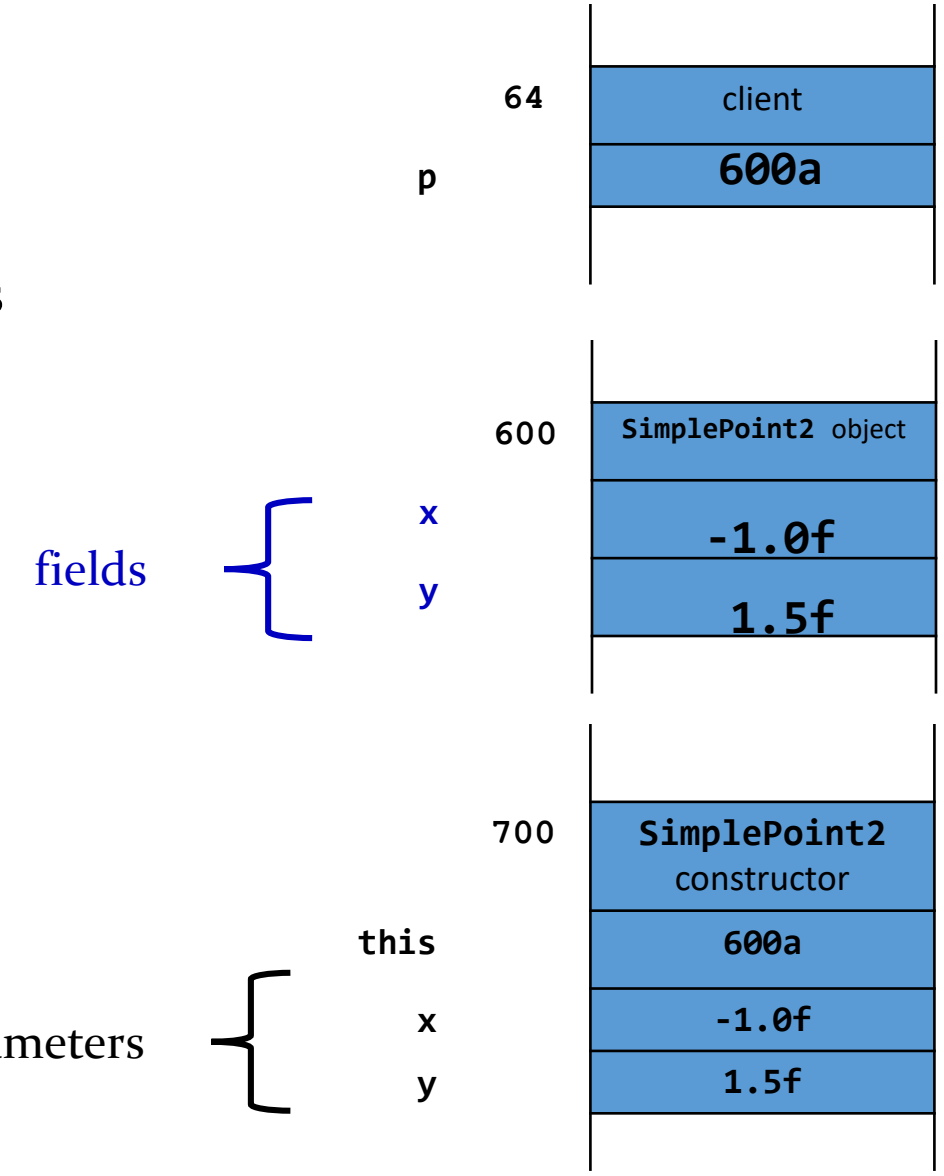
this.y : the field named **y** of **this** point

x : the parameter named **x** of the constructor

y : the parameter named **y** of the constructor

```
SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);
```

1. **new** allocates memory for a **SimplePoint2** object
2. the **SimplePoint2** constructor is invoked by passing the memory address of the object and the arguments **-1.0f** and **1.5f** to the constructor
3. the constructor runs, setting the values of the fields **this.x** and **this.y**
4. the value of **p** is set to the memory address of the constructed object



this

- In our constructor

```
public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
  }
```

there are parameters with the same names as fields when this occurs, the parameter has precedence over the field.

- we say that the parameter *shadows* the field, ***when shadowing occurs you must use `this` to refer to the field***

Custom constructors

Adding the constructor **`SimplePoint2(float x, float y)`** allows the client to simplify their code.

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2();

  // set its coordinates
  p.x = -1.0f;
  p.y = 1.5f;

  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2();
  q.x = p.x;
  q.y = p.y;

  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

Create p SimplePoint2 object

Set the values for p SimplePoint2 object

Create q SimplePoint2 object

Copy values from p SimplePoint2 object

Test if p and q objects are equal to each other

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);

  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(p.x, p.y);

  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

Revised version using custom constructor

Copy constructor

- A copy constructor initializes the state of an object by copying the state of another object (having the same type)
  - *it has a single parameter that is the same type as the class*

```java
public class SimplePoint2 {
  public float x;
  public float y;


  /**
   * Sets the x and y coordinate of this point by copying
   * the x and y coordinate of another point.
   *
   * @param other a point to copy
   */
  public SimplePoint2(SimplePoint2 other) {
    this.x = other.x;
    this.y = other.y;
  }
```

Copy constructor

- Adding a copy constructor allows the client to simplify their code

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);




  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(p.x, p.y);




  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

```
public static void main(String[] args) {
  // create a point
  SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);




  // get its coordinates
  System.out.println("p = (" + p.x + ", " + p.y + ")");

  SimplePoint2 q = new SimplePoint2(p);



  // equals?
  System.out.println("p.equals(q) is: " + p.equals(q));
}
```

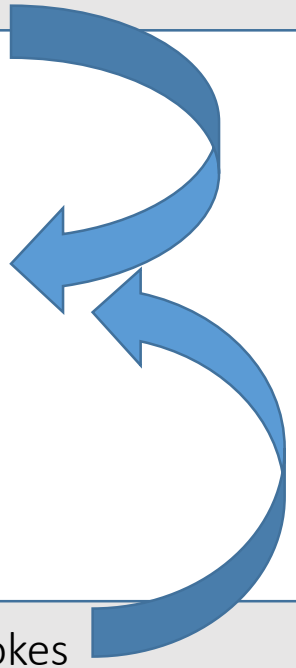
Avoiding Code Duplication

- Notice that the constructor bodies are almost identical to each other
  - all three constructors have 2 lines of code
  - all three constructors set the **`x`** and **`y`** coordinate of the point
- Whenever you see **duplicated code** you should consider ***moving*** the duplicated code into a method
- In this case, one of the constructors already does everything we need to implement the other constructors…

Constructor chaining

- A constructor is allowed to invoke another constructor
- When a constructor invokes another constructor it is called ***constructor chaining***
- To invoke a constructor in the same class you use the `this` keyword
  - ***if you do this then it must occur on the first line of the constructor body***
    - ***but you cannot use `this` in a method to invoke a constructor***
- We can re-write two of our constructors to use constructor chaining...

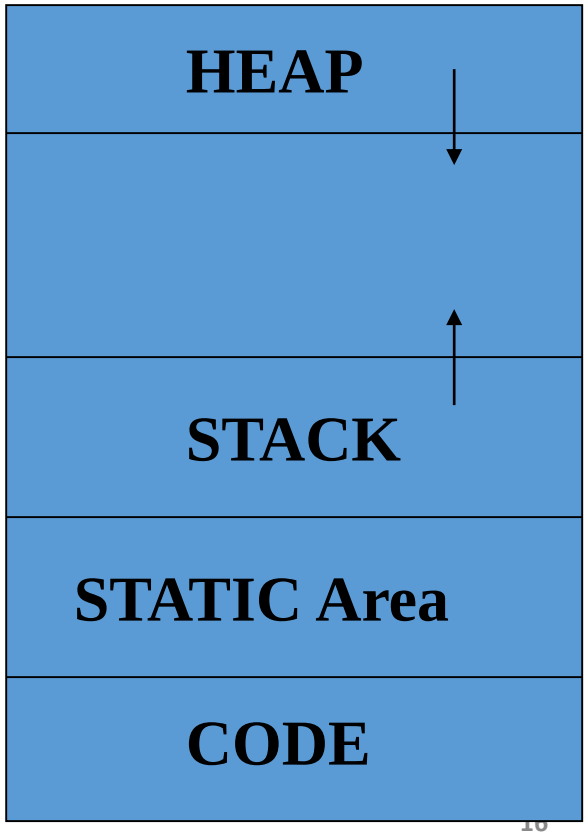
```
public class SimplePoint2 {
  public float x;
  public float y;

  public SimplePoint2() {
    this(0.0f, 0.0f);
  }

  public SimplePoint2(float x, float y) {
    this.x = x;
    this.y = y;
  }

  public SimplePoint2(SimplePoint2 other) {
    this(other.x, other.y);
  }
```



Objects

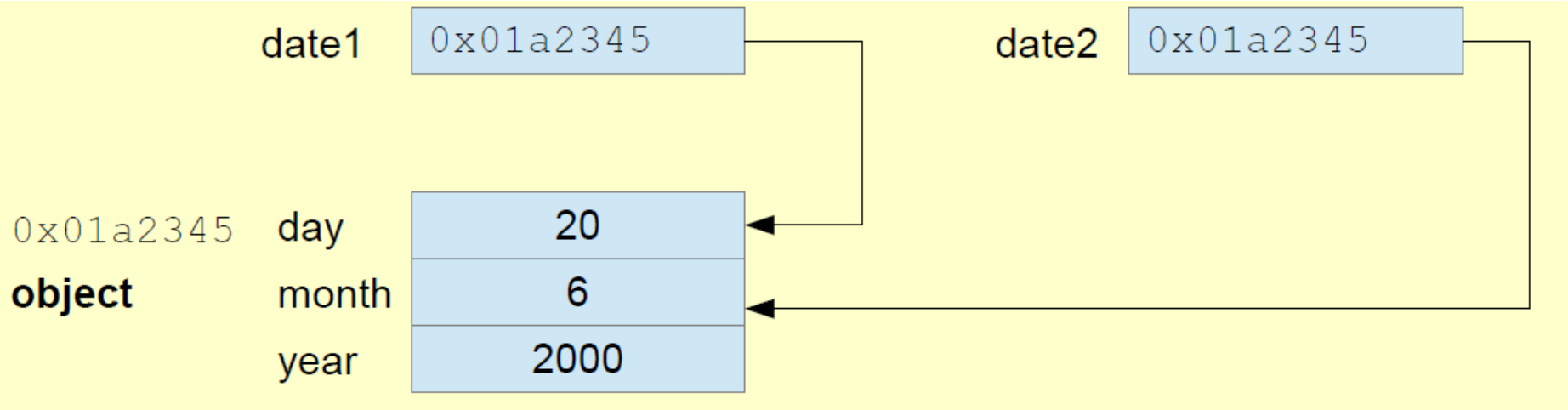
- Objects are **instances** of classes
- Are **allocated on the heap** by using the new operator
- Constructor is invoked automatically on the ***new object.***

```
Counter c = new Counter();
Date d1 = new Date( 2016, 9, 23);
Person p = new Person("John","Smith");
```



Two variables refer to a single object

```
MyDate date1 = new MyDate(20, 6, 2000);
MyDate date2 = date1;
```



Interesting Fact about Constructors

- The constructors can call each other using the special **`this`** keyword.
- It is considered a good practice to chain constructors in such a way as it reduces code duplication and basically leads to having single initialization entry point.
- As an example, let us add another constructor with only one argument.

```
public ConstructorWithArguments(final String arg1) {
            this(arg1, null);
}
```

The `this` Reference

- Usage:
  - To resolve **ambiguity** between ***instance variables*** and ***parameters***
  - To **pass** the current **object as a parameter** to another method.

```
public class MyDate{
	private int day = 26;
	private int month = 9;
	private int year = 2016;
	public MyDate( int day, int month, int year){
		this.day = day;
		this.month = month;
		this.year = year;
	}
	public MyDate( MyDate date){
		this.day = date.day;
		this.month = date.month;
		this.year = date.year;
	}
	public MyDate creteNextDate(int moreDays){
		MyDate newDate = new MyDate(this);
		//... add moreDays
		return newDate;
	}
}
```

Initialization Blocks

- Java has yet another way to provide initialization logic using initialization blocks. This feature is rarely used.
- The initialization block might be treated as anonymous no-arg constructor.
  - But initialization blocks do not replace the constructors
- Some classes may have ***multiple initialization blocks*** and they all will be called in the order they are defined in the code.
- But, ***it is very important to mention that initialization blocks are always called before any constructor***.

```
public class InitializationBlock {
    {
    // initialization code here
    }
}
```

```
public class InitializationBlocks {
    {
    // initialization code here
    }
    {
    // initialization code here
    }

}
```

Static initialization

- Java also supports class-level initialization constructs called **static initializers**.
  - There are very similar to the initialization blocks except for the additional **`static`** keyword.
- A class can have any number of static initializer blocks in the class definition and they will be executed in the order in which they appear in the code.

```
public class StaticInitializationBlock {
   static {
   // static initialization code here
   }
}
```

Methods

Basics

Methods

- A method performs some sort of computation
- A method is reusable
  - anyone who has access to the method can use the method ***without copying*** *the contents of the method*
  - anyone who has access to the method can use the method ***without knowing the contents of the method***
- Methods are described by their API (application program interface); for example:
  - https://www.eecs.yorku.ca/course_archive/2017-18/W/2030Z/lectures/doc/week01/

Example API method entry

isBetween

```
public static boolean isBetween(int min,
                                int max,
                                int value)
```

Returns true if `value` is strictly greater than `min` and strictly less than `max`, and false otherwise.

Parameters:

`min` - a minimum value

`max` - a maximum value

`value` - a value to check

Returns:

`true if value is strictly greater than min and strictly less than max, and false otherwise`

Precondition:

`min is less than or equal to max`

Method header

The first line of a method declaration is sometimes called the *method header*

```
public static boolean isBetween(int min,int max,int value)
```

modifiers | return type | name | parameter list

Method parameter list

The parameter list is the list of types and names that appear inside of the parentheses

```
public static boolean
    isBetween(int min, int max, int value)
```

parameter list

- The names in the parameter list must be unique
  - i.e., ***duplicate parameter names are not allowed***

Method signature

- Every method has a ***signature***
  - the signature consists of the **method name** and **the types in the parameter list**

```
public static boolean isBetween(int min,
                                int max,
                                int value)
```

has the following signature

name — number and types of parameters

```
isBetween(int, int, int)
```

signature

Examples - Method signature

- Other examples from **java.lang.String**
  - headers
    - `String toUpperCase()`
    - `char charAt(int index)`
    - `int indexOf(String str, int fromIndex)`
    - `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  - **signatures**
    - `toUpperCase()`
    - `charAt(int)`
    - `indexOf(String, int)`
    - `getChars(int, int, char[], int)`

Method signature must be unique

- Method signatures in a class must be **unique**
- We can introduce a second method in the same class:

```
public static boolean
   isBetween(double min, double max,
  double value)
```

Method overloaded

- Two or methods with the same name but different signatures are said to be ***overloaded***

```
public static boolean
   isBetween(int min, int max, int value)

public static boolean
   isBetween(double min, double max, double
  value)
```

Method return types

- All Java methods return nothing (**`void`**) or a single type of value
- So, method

```
public static boolean
   isBetween(double min, double max,
double value)
```

has the return type **`boolean`**

Methods

Preconditions and postconditions

You are the head of a programming team and you want one of your programmers to write a function for part of a project.

Client vs. Supplier – programming concept

- A supplier implements/provides a service (also called implementer)
- A client uses a service provided by a given supplier.
  - The client must follow certain instructions to obtain the service.
  - If instructions are followed, the client would **expect** that the **service does what is required**.
  - ***The client does not care how the supplier implements it.***
- **There is a contract between two parties,** violated if:
  - The instructions are not followed. [**Client's fault**]
  - Instructions followed, but service not satisfactory. [**Supplier's fault**]

Preconditions and postconditions

- Precondition:
  - a condition that the ***client*** ***must ensure is true*** immediately ***before*** a method is **invoked**/called.
- Postcondition:
  - a condition that the *method* ***must ensure is true*** immediately ***after*** the method is finished.

Preconditions

- A method precondition is a condition that the ***client*** must ensure is true immediately before invoking a method
  - if the precondition is not true, then the client has no guarantees of what the method will do.
- For **static methods**, preconditions are conditions on the values of the arguments passed to the method
  - ***you need to carefully read the API to discover the preconditions***

isBetween

```
public static boolean isBetween(int min,
                                int max,
                                int value)
```

Returns true if `value` is strictly greater than `min` and strictly less than `max`, and false otherwise.

Parameters:

`min` - a minimum value

`max` - a maximum value

`value` - a value to check

Returns:

`true if value is strictly greater than min and strictly less than max, and false otherwise`

Precondition:

`min is less than or equal to max`

precondition

min2

```
public static int min2(List<Integer> t)
```

Given a list containing exactly 2 integers, returns the smaller of the two integers. The list `t` is not modified by this method. For example:

precondition

```
t             Test2F.min2(t)
---------------------------
[-5, 9]       -5
[3, 3]         3
[12, 6]        6
```

Preconditions?

Parameters:

`t - a list containing exactly 2 integers`

Returns:

`the minimum of the two values in t`

Throws:

`IllegalArgumentException - if the list does not contain exactly 2 integers`

Precondition:

`t is not null`

precondition

Preconditions

- If a method has a parameter that has reference type then it is almost always assumed that a precondition for that parameter is that it is not equal to `null`
- Reminders:
  - reference type means "**not primitive type**"
  - `null` means "refers to no object"
    - **primitive types are never equal to `null`**

Postconditions

- A method postcondition is a condition that the *method* must ensure is true immediately after the method is finished
  - if **the postcondition is not true**, then there is something wrong with the implementation of the method.
- For **static methods**, ***postconditions*** are:
  - conditions on the arguments after the method finishes
  - conditions on the return value.

isBetween

```
public static boolean isBetween(int min,
                                int max,
                                int value)
```

Returns true if `value` is strictly greater than `min` and strictly less than `max`, and false otherwise.

Parameters:

`min` - a minimum value

`max` - a maximum value

`value` - a value to check

Returns:

`true if value is strictly greater than min and strictly less than max, and false otherwise`

Precondition:

`min is less than or equal to max`

Postconditions?

postcondition

min2

```
public static int min2(List<Integer> t)
```

Given a list containing exactly 2 integers, returns the smaller of the two integers. The list `t` is not modified by this method. For example:

postcondition

```
t             Test2F.min2(t)
---------------------------
[-5, 9]       -5
[3, 3]         3
[12, 6]        6
```

Postconditions?

Parameters:

`t - a list containing exactly 2 integers`

Returns:

`the minimum of the two values in t`

postcondition

Throws:

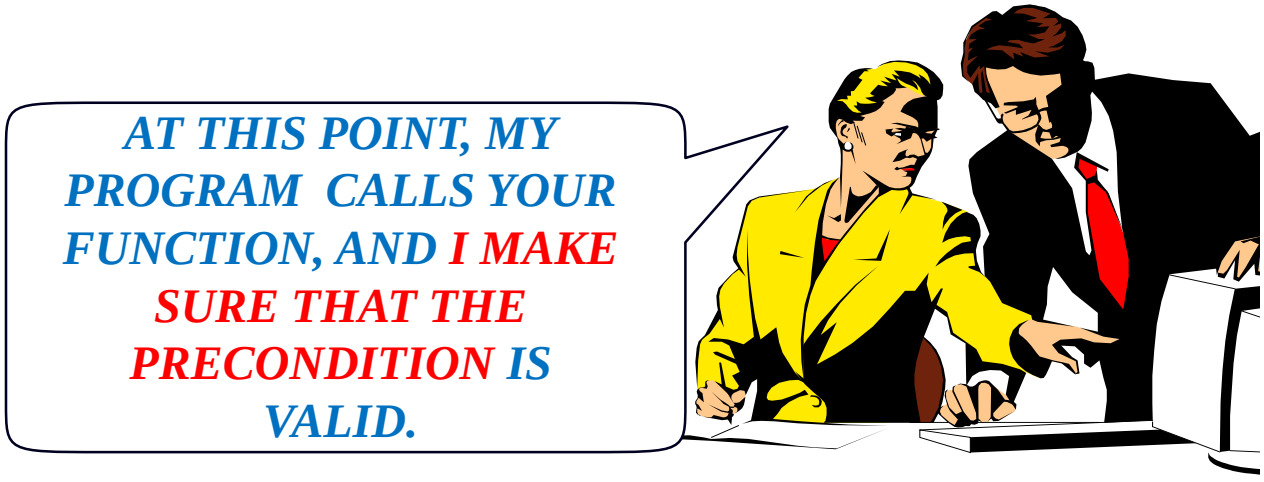
`IllegalArgumentException - if the list does not contain exactly 2 integers`

Precondition:

`t is not null`

Always make sure the precondition is valid . . .

- The programmer who calls the function is responsible for **ensuring that the precondition is valid** when the function is invoked/called.



. . . so the postcondition becomes true at the function's end.

- The programmer who writes the function counts on the precondition being valid, and **ensures that the postcondition becomes true** at the function's end.

Always

- When you write a function, you should make every effort to detect when a **precondition *has been violated***.
- If you detect that a precondition has been violated, **then print an error message and halt the program.**

Methods

Implementation

isBetween

```
public static boolean isBetween(int min,
                                int max,
                                int value)
```

Returns true if `value` is strictly greater than `min` and strictly less than `max`, and false otherwise.

Parameters:

`min` - a minimum value

`max` - a maximum value

`value` - a value to check

Returns:

`true if value is strictly greater than min and strictly less than max, and false otherwise`

Precondition:

`min is less than or equal to max`

Methods and classes

- In Java every method must be defined inside of a class
- We will try to implement our method so that it matches its **API**:
  - the method is inside the class named **`Test2F`**
  - the class **`Test2F`** is inside the package **`eecs2030.test2`**

```java
package eecs2030.test2;

public class Test2F {

}
```

Method body

- A method implementation consists of:
  - **the method header that includes method signature.**
  - a method body
    - the body is a sequence of Java statements inside of a pair of braces **{ }**

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {

    }

}
```

the method header that includes method signature

Methods with parameters

- If a method has parameters, then you can use the parameter names as variables inside your method
  - **you cannot create new variables inside the method that have the same name as a parameter**
  - you cannot use the parameters outside of the method
    - we say that the ***scope*** of the parameters is the method body
- You may create additional variables inside your method if you wish
  - we will create a variable to store the return value of the method

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        boolean result = true;

    }

}
```

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        boolean result = true;
        if (value <= min) {
            result = false;
        }
        if (value >= max) {
            result = false;
        }

    }

}
```

Methods with return values

- If the **method header** says that a type is returned, then the method **must return a value** having the advertised type back to the client
- You use the keyword `return` to return the value back to the client

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        boolean result = true;
        if (value <= min) {
            result = false;
        }
        if (value >= max) {
            result = false;
        }
        return result;
    }

}
```

Method return values

- A method **stops running immediately** if a return statement is run
  - this means that you are not allowed to have additional code if a return statement is reached
  - however, **you can have multiple return statements**

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        if (value <= min) {
            return false;
            // code not allowed here
        }
        if (value >= max) {
            return false;
            // code not allowed here
        }
        return true;
        // code not allowed here
    }

}
```

Alternative implementations

There are many ways to implement this particular method

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        if (value <= min || value >= max) {
            return false;
        }
        return true;
    }

}
```

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        if (value > min && value < max) {
            return true;
        }
        return false;
    }

}
```

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        boolean result = value > min && value < max;
        return result;
    }

}
```

```java
package eecs2030.test2;

public class Test2F {

    public static boolean isBetween(int min, int max, int value) {
        return value > min && value < max;
    }

}
```

min2

```
public static int min2(List<Integer> t)
```

Given a list containing exactly 2 integers, returns the smaller of the two integers. The list `t` is not modified by this method. For example:

```
t            Test2F.min2(t)
--------------------------
[-5, 9]      -5
[3, 3]        3
[12, 6]       6
```

Parameters:

`t - a list containing exactly 2 integers`

Returns:

`the minimum of the two values in t`

Throws:

`IllegalArgumentException - if the list does not contain exactly 2 integers`

Precondition:

`t is not null`

```java
package eecs2030.test2;

import java.util.List;

public class Test2F {

    // implementation of isBetween not shown

    public static int min2(List<Integer> t) {

    }
}
```

```java
package eecs2030.test2;

import java.util.List;

public class Test2F {

    // implementation not shown

    public static int min2(List<Integer> t) {
        if (t.size() != 2) {
            throw new IllegalArgumentException("list size != 2");
        }
        int first = t.get(0);
        int second = t.get(1);
    }
}
```

Check the size of list
Fetch the first and second elements
Still not complete

```java
package eecs2030.test2;

import java.util.List;

public class Test2F {

    // implementation not shown

    public static int min2(List<Integer> t) {
        if (t.size() != 2) {
            throw new IllegalArgumentException("list size != 2");
        }
        int first = t.get(0);
        int second = t.get(1);
        if (first < second) {
            return first;
        }
        return second;
    }
}
```

References

- https://docs.oracle.com/javase/10/docs/api/overview-summary.html
- https://www.eecs.yorku.ca/course_archive/ [look for EECS 2030]