Advanced Object Oriented Programming

EECS2030Z

Who Am I?

- Dr. Burton Ma
- office
  - Lassonde 2046
  - hours : to be updated on the syllabus page
- email
  - **`burton@cse.yorku.ca`**

Course Format

- everything you need to know will eventually be on the York University Moodle site (not the learn.lassonde Moodle site)

Labs

- in Prism computing labs (LAS1006)
- Lab Zero starts in Week 1
  - self-guided, can be done anytime before the start of Week 2
  - using the Prism lab environment
  - using eclipse
- Labs 1-8 consist of a different set of programming problems for each lab
- *it is expected that you know how to use the lab computing environment*

Labs

- group lab work is allowed and strongly encouraged for Labs 1-8 (not Lab 0)
  - groups of up to size 3
  - see *Academic Honesty* section of syllabus
    - TLDR Do not submit work that is not wholly your own

Tests

- all testing occurs during your regularly scheduled lab using the EECS labtest environment

Test	Weight
Test 1	1%
Test 2	15%
Test 3	15%
Test 4	15%
Exam	30%

- miss a test for an acceptable reason?
  - see *Evaluation: Missed tests* section of syllabus

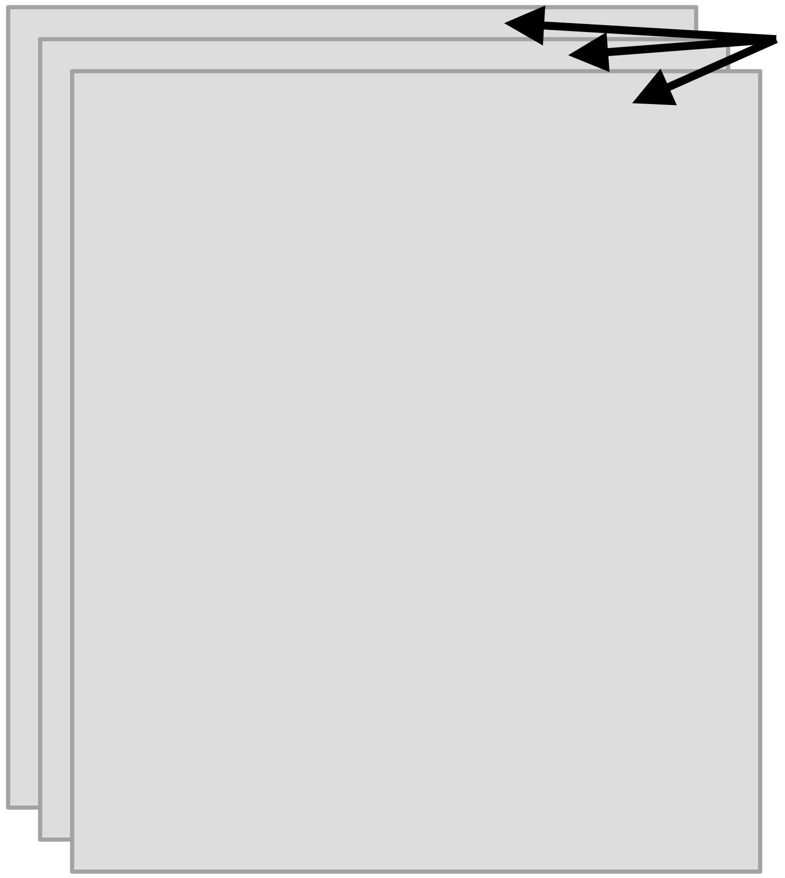
Textbook

- a set of freely available electronic notes is available from the Moodle site
- recommended textbooks
  - *Building Java Programs*, 4th Edition, S Roges and M Stepp
  - *Introduction to Programming in Java*, 2nd Edition, R Sedgewick and K Wayne
    - does not cover inheritance
  - *Absolute Java*, 6th Edition, W Savitch
- recommended references
  - *Java 8 Pocket Guide*, Liguori and Liguori
  - *Effective Java*, 3rd Edition, J Bloch

Organization of a Java Program

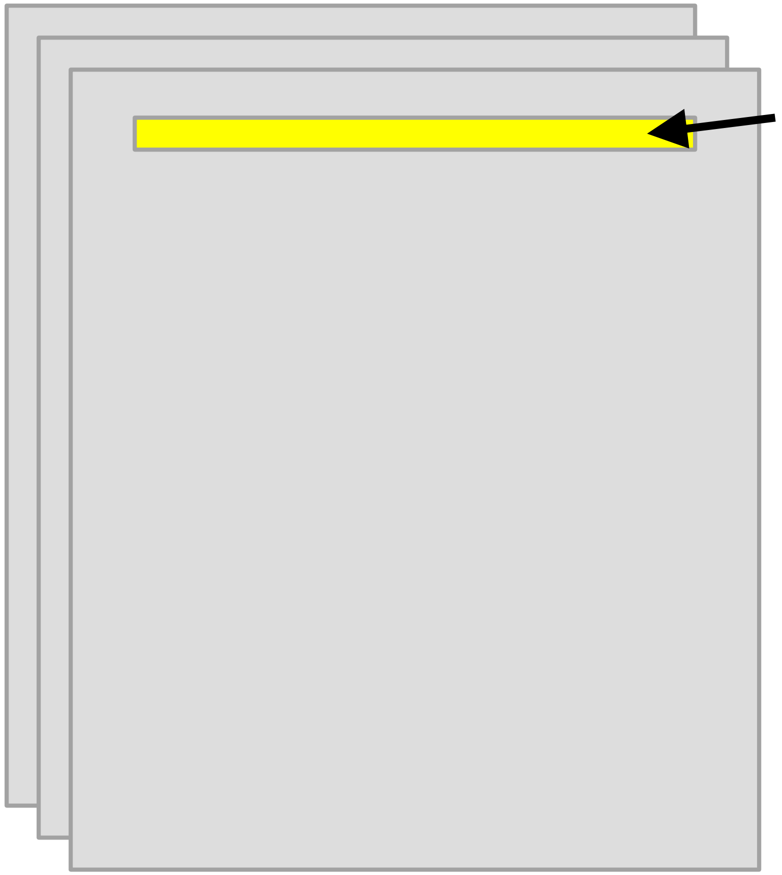
Packages, classes, fields, and methods

Organization of a Typical Java Program



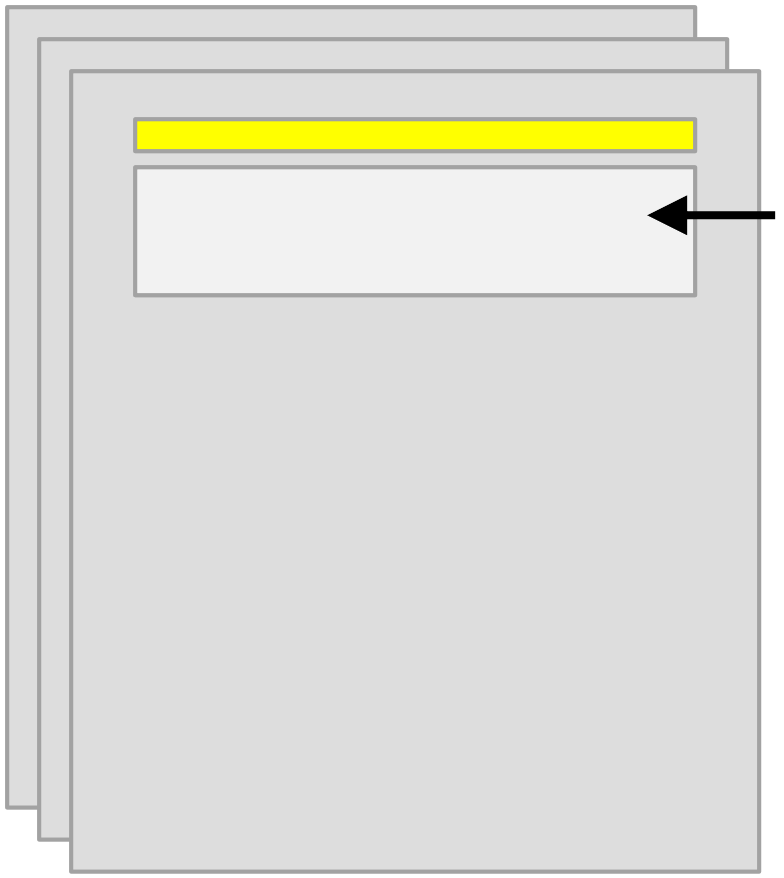
- one or more files

Organization of a Typical Java Program



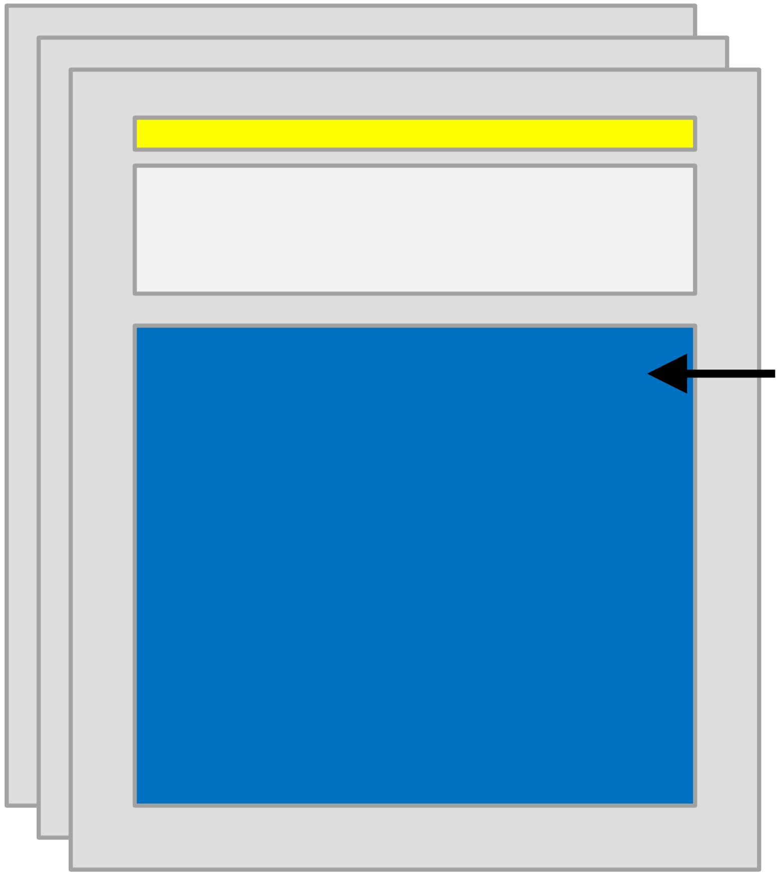
- one or more files
- zero or one package name

Organization of a Typical Java Program



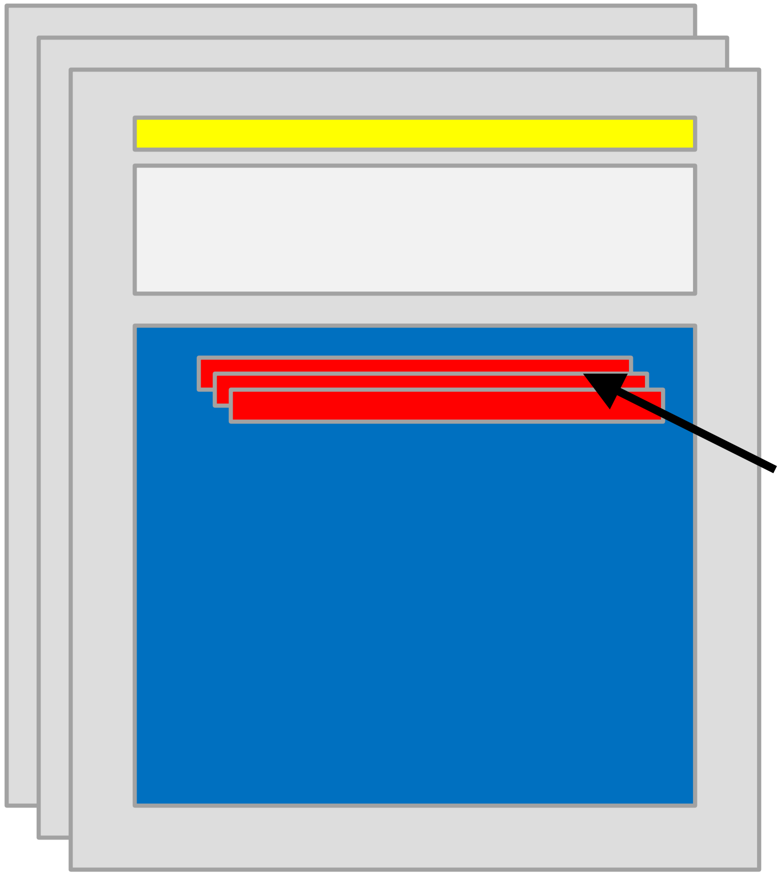
- one or more files
- zero or one package name
- zero or more import statements

Organization of a Typical Java Program



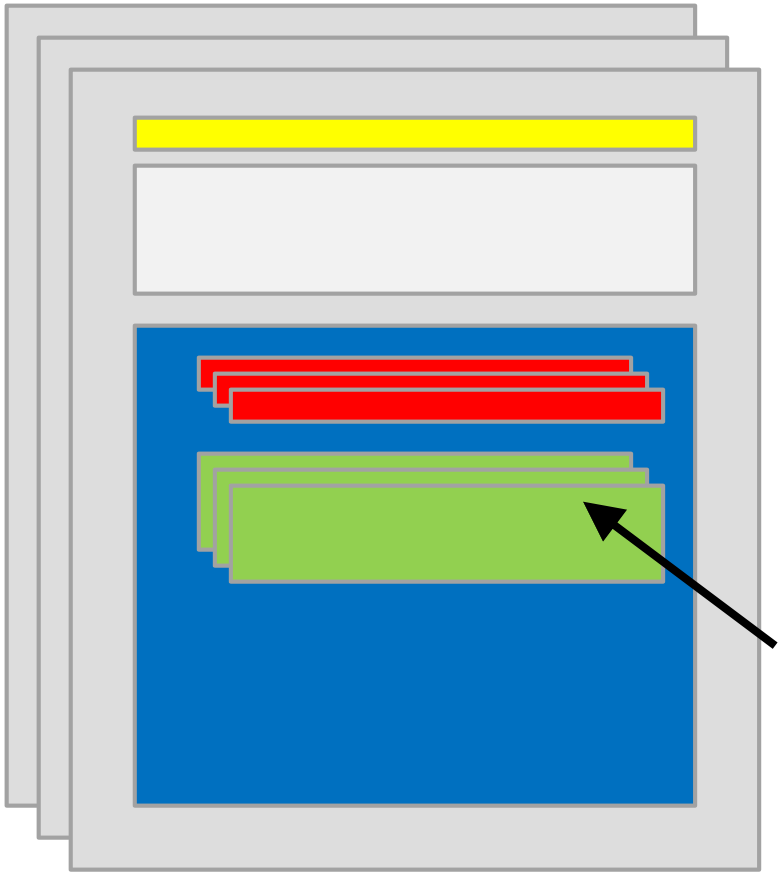
- one or more files
- zero or one package name
- zero or more import statements
- one class

Organization of a Typical Java Program



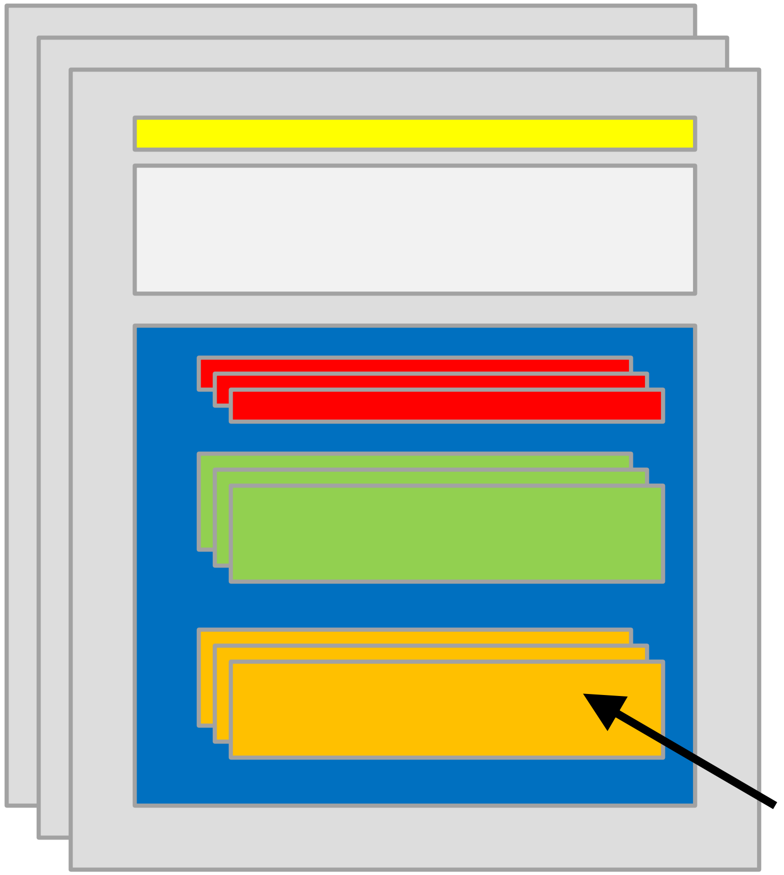
- one or more files
- zero or one package name
- zero or more import statements
- one class
- one or more fields (class variables)

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors
- zero or more methods

Organization of a Typical Java Program

- it's actually more complicated than this
  - static initialization blocks
  - non-static initialization blocks
  - classes inside of classes (inside of classes ...)
  - classes inside of methods
  - anonymous classes
  - lambda expressions (in Java 8)

- see http://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Packages

- packages are used to organize Java classes into namespaces
- a namespace is a container for names
  - the namespace also has a name

Packages

- packages are use to organize related classes and interfaces
  - e.g., all of the Java API classes are in the package named **`java`**

Packages

- packages can contain subpackages
  - e.g., the package **`java`** contains packages named **`lang`**, **`util`**, **`io`**, etc.

- the fully qualified name of the subpackage is the fully qualified name of the parent package followed by a period followed by the subpackage name
  - e.g., **`java.lang`**, **`java.util`**, **`java.io`**

Packages

- packages can contain classes and interfaces
  - e.g., the package **`java`**.**`lang`** contains the classes **`Object`**, **`String`**, **`Math`**, etc.

- the fully qualified name of the class is the fully qualified name of the containing package followed by a period followed by the class name
  - e.g., **`java.lang.Object`**, **`java.lang.String`**, **`java.lang.Math`**

Packages

- packages are supposed to ensure that fully qualified names are unique
- this allows the compiler to disambiguate classes with the same unqualified name, e.g.,

```
your.String s = new your.String("hello");
String t = "hello";
```

Packages

- how do we ensure that fully qualified names are unique?
- package naming convention
  - packages should be organized using your domain name in reverse, e.g.,
    - EECS domain name **`eecs.yorku.ca`**
    - package name **`ca.yorku.eecs`**
- we might consider putting everything for this course under the following package
  - **`eecs2030`**

Packages

- we might consider putting everything for this course under the following package
  - **`eecs2030`**
- labs might be organized into subpackages:
  - **`eecs2030.lab0`**
  - **`eecs2030.lab1`** and so on
- tests might be organized into subpackages:
  - **`eecs2030.test1`**
  - **`eecs2030.test2`** and so on

Packages

- most Java implementations assume that your directory structure matches the package structure, e.g.,
  - there is a folder **`eecs2030`** inside the project **`src`** folder
    - there is a folder **`lab0`** inside the **`eecs2030`** folder
    - there is a folder **`lab1`** inside the **`eecs2030`** folder, and so on

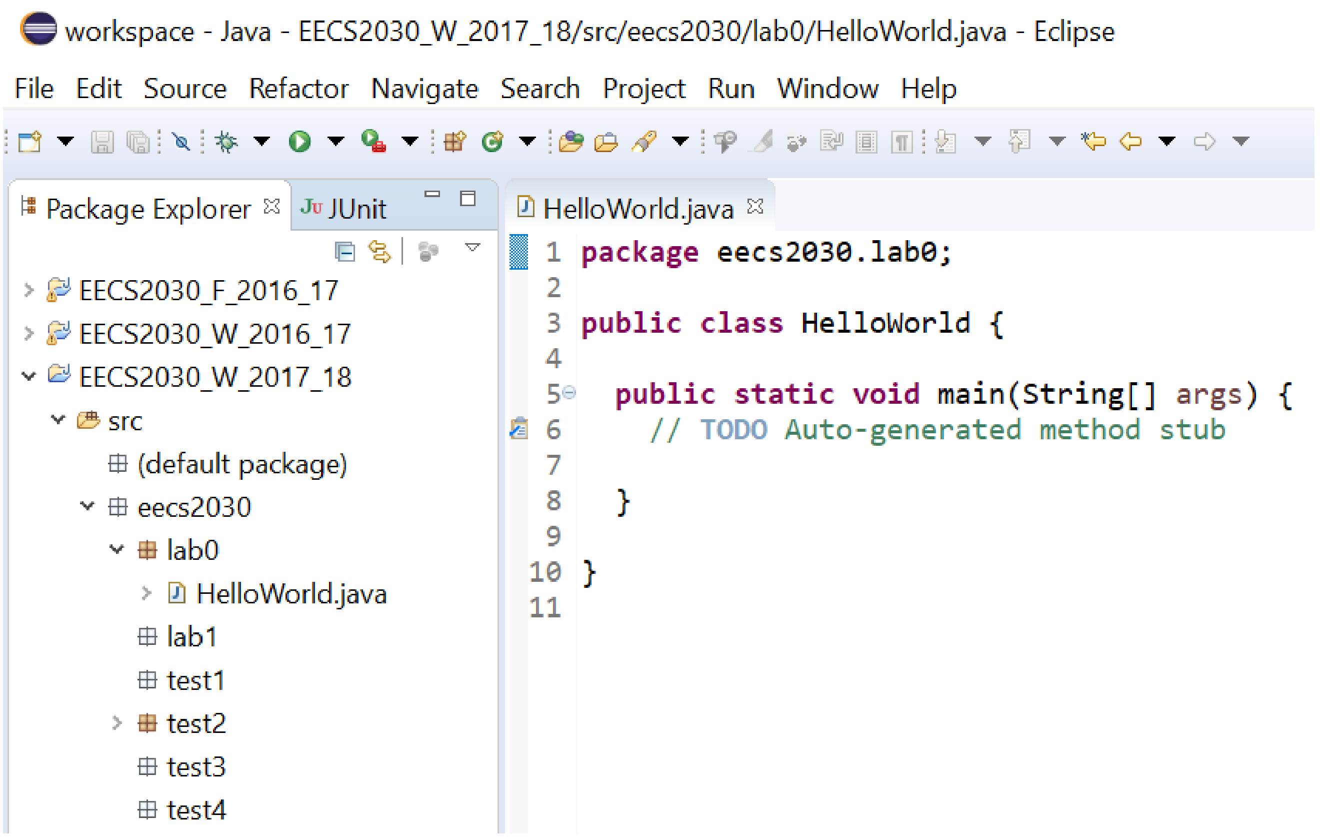
Packages

project folder

project sources folder

eecs2030 folder

lab0 folder



```
package eecs2030.lab0;

public class HelloWorld {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}

```

Methods

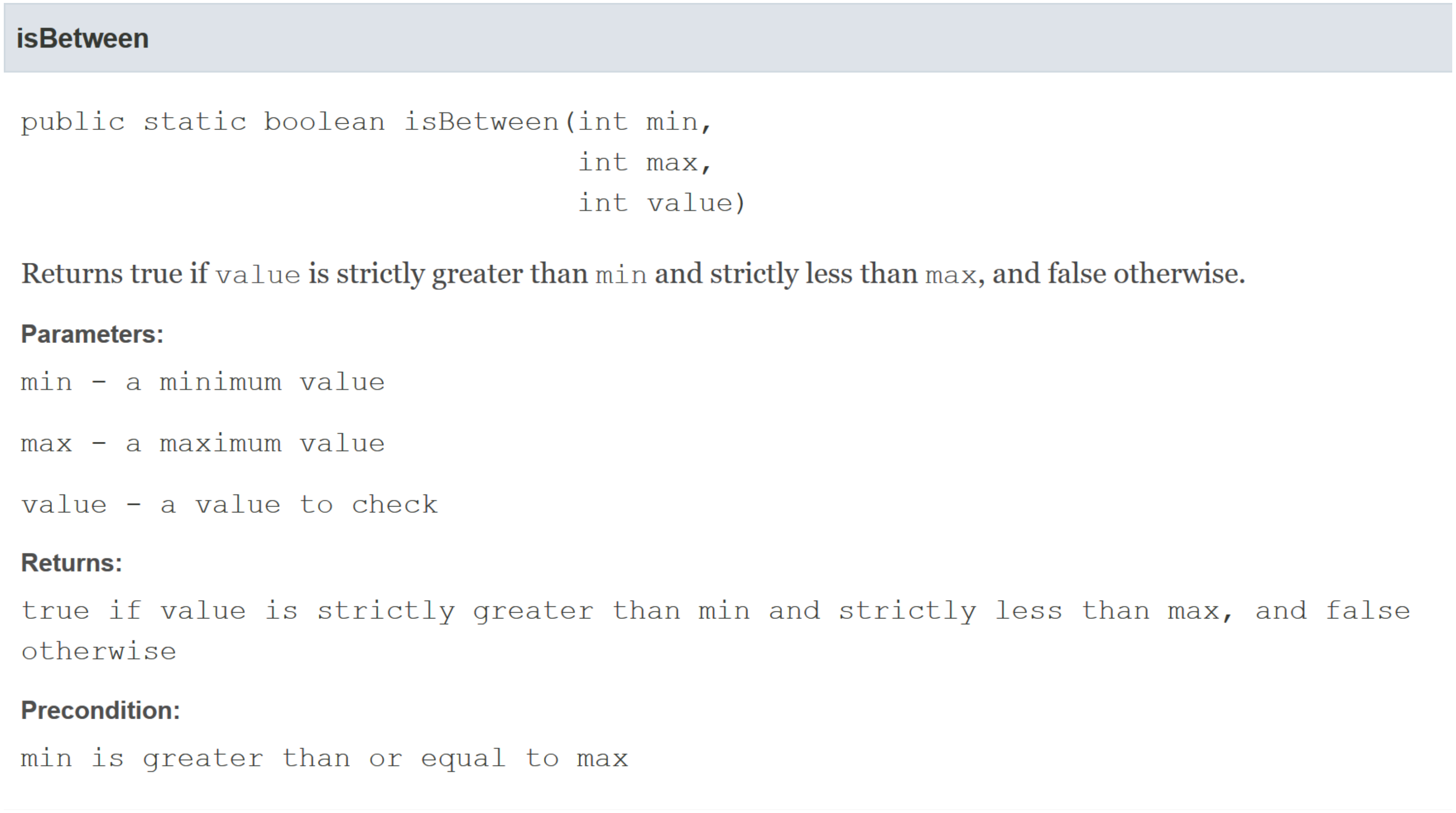
Basics

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 greater 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)
  ```

  $\underbrace{\texttt{int min, int max, int value}}_{\textbf{parameter list}}$

- 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

$$\underbrace{\overbrace{\texttt{isBetween}}^{\text{name}}\texttt{(}\overbrace{\texttt{int, int, int}}^{\text{number and types of parameters}}\texttt{)}}_{\text{signature}}$$

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

- 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)
```

- but not this one:

```
public static boolean
  isBetween(int value, int lo, int hi)
```

Method signature

- 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
- our method

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

has the return type **`boolean`**

Methods

Preconditions and postconditions

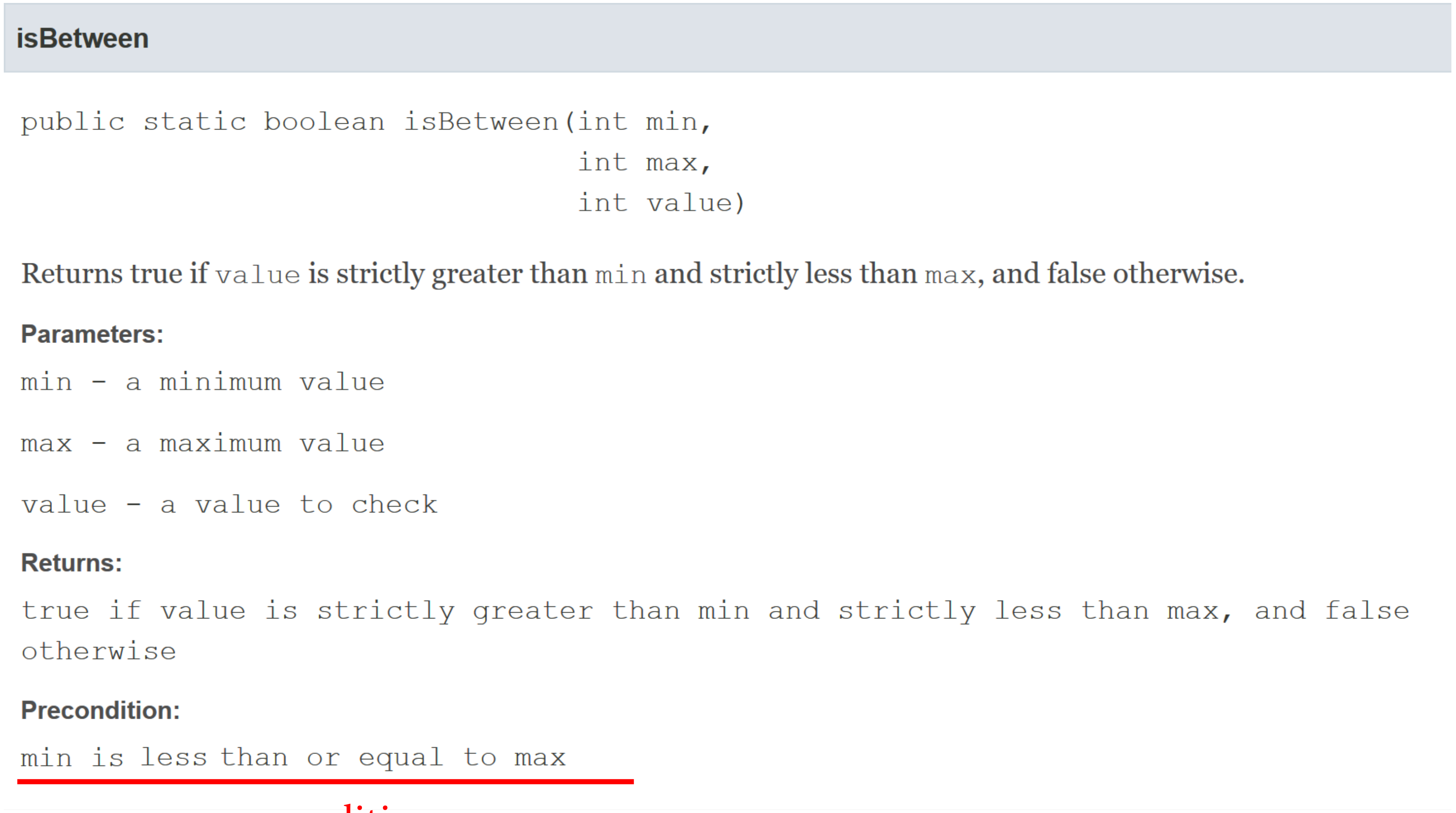
Preconditions and postconditions

- recall the meaning of method pre- and postconditions
- precondition
  - a condition that the *client* must ensure is true immediately before a method is invoked
- postcondition
  - a condition that the *method* must ensure is true immediately after the method is invoked

Preconditions

- recall that 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`

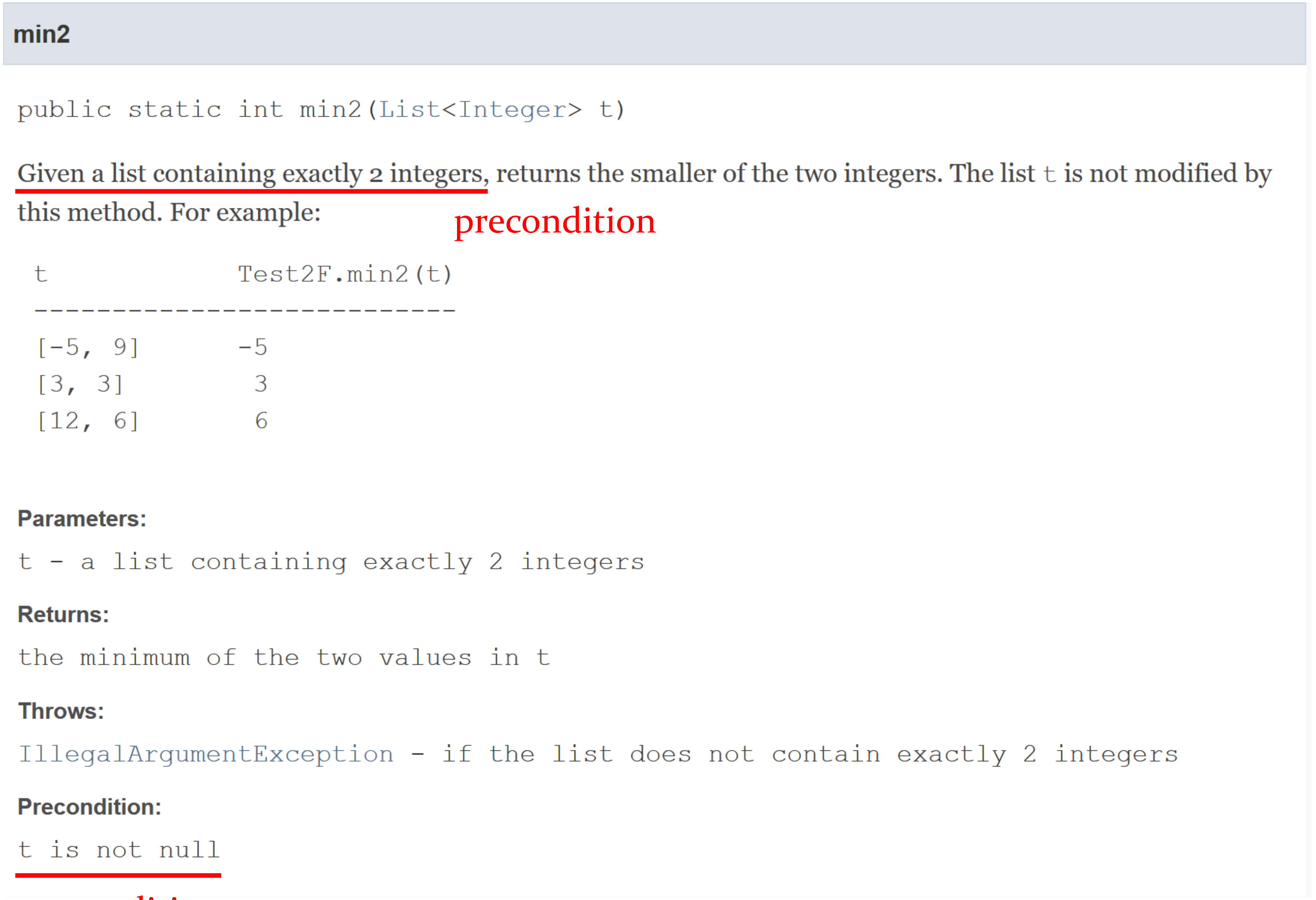
Returns:

`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
```

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

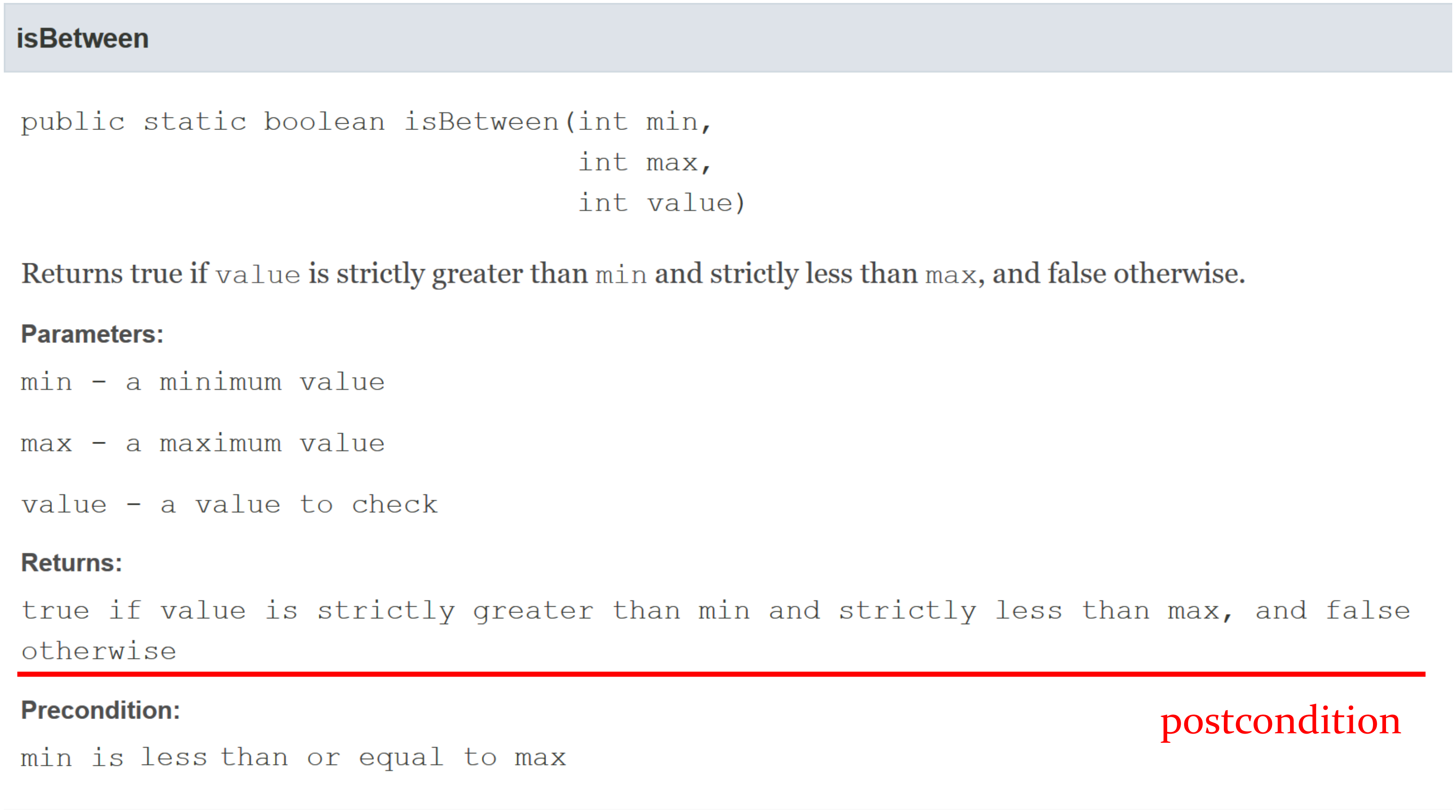
Preconditions

- 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

- recall that a method postcondition is a condition that the *method* must ensure is true immediately after the method is invoked
  - 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`

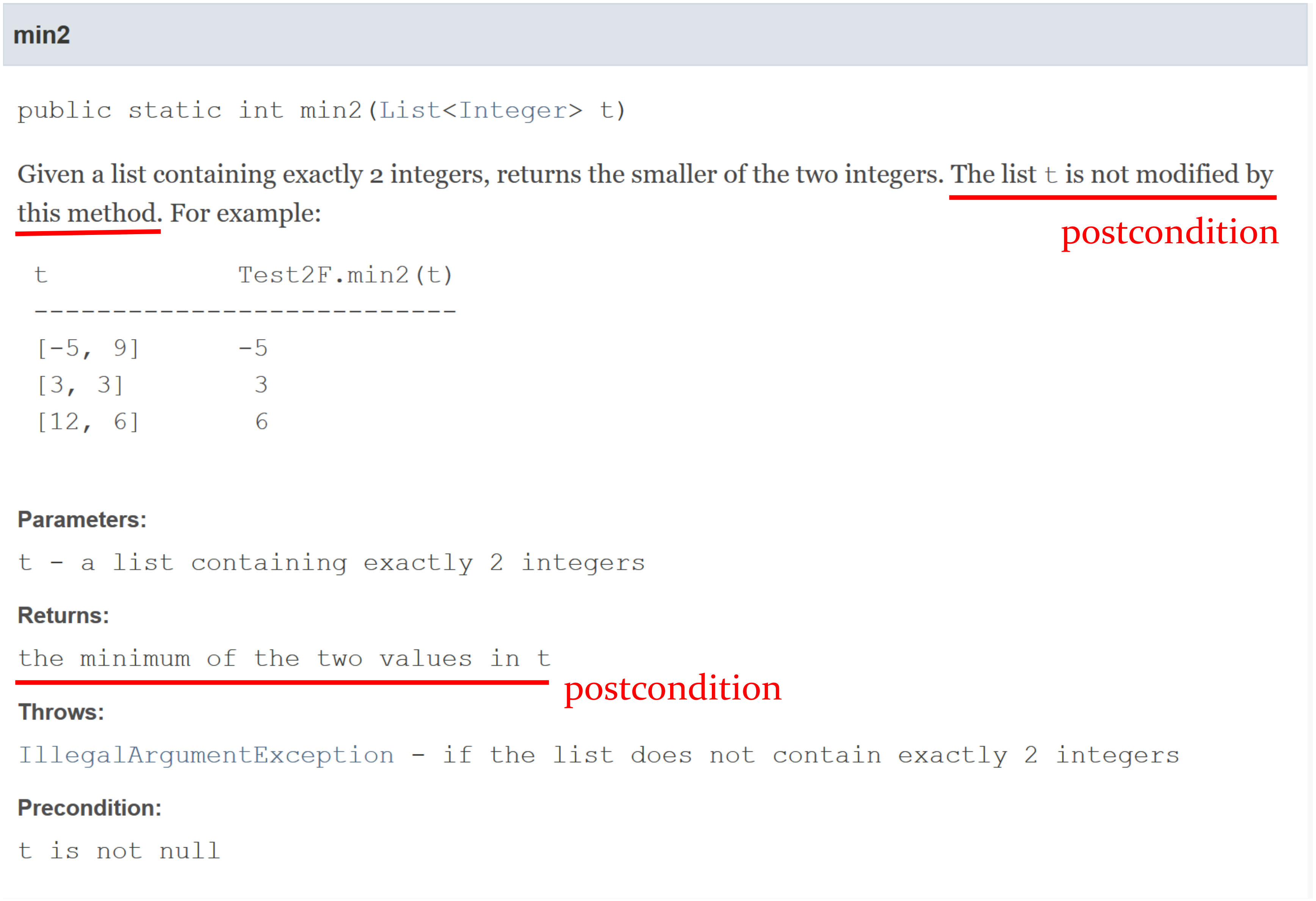
Returns:

`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`

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
```

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`

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`**

- eclipse demonstration here

```java
package eecs2030.test2;

public class Test2F {

}
```

Method body

- a method implementation consists of:
  - the method header
  - a method body
    - the body is a sequence of Java statements inside of a pair of braces **`{ }`**

```
package eecs2030.test2;

public class Test2F {

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

    }

}
```

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

```
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`

```
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);
    }
}
```

```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;
    }
}
```

Invoking methods

Pass-by-value

`static` Methods

- a method that is **`static`** is a per-class member
  - client does not need an object reference to invoke the method
  - client uses the class name to access the method

```
boolean isBetween = Test2F.isBetween(0, 5, 2);
```

  - **`static`** methods are also called *class methods*

[notes 1.2.4]

Invoking methods

- a client invokes a method by passing arguments to the method
  - the types of the arguments must be compatible with the types of parameters in the method signature
  - the values of the arguments must satisfy the preconditions of the method contract

```
List<Integer> t = new ArrayList<Integer>();
t.add(100);
t.add(-99);
int min = Test2F.min2(t);
```

argument

Pass-by-value

- Java uses pass-by-value to:
  - transfer the value of the arguments to the method
  - transfer the return value back to the client

- consider the following utility class and its client…

```java
import type.lib.Fraction;

public class Doubler {

  private Doubler() {
  }

  // tries to double x
  public static void twice(int x) {
    x = 2 * x;
  }

  // tries to double f
  public static void twice(Fraction f) {
    long numerator = f.getNumerator();
    f.setNumerator( 2 * numerator );
  }
}
```

assume that a **`Fraction`** represents a fraction (i.e., has an integer numerator and denominator)

```java
import type.lib.Fraction;

public class TestDoubler {

  public static void main(String[] args) {
    int a = 1;
    Doubler.twice(a);

    Fraction b = new Fraction(1, 2);
    Doubler.twice(b);

    System.out.println(a);
    System.out.println(b);
  }

}
```

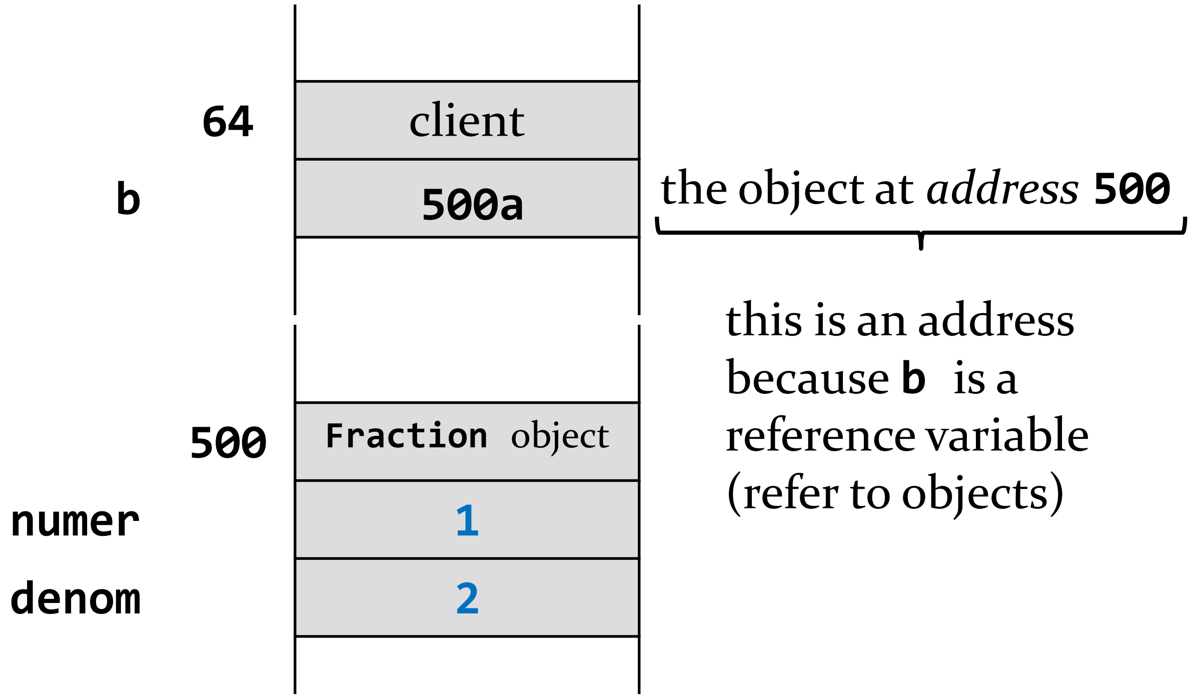
Pass-by-value

- what is the output of the client program?
  - try it and see

- an invoked method runs in its own area of memory that contains storage for its parameters
- each parameter is initialized with *the value* of its corresponding argument

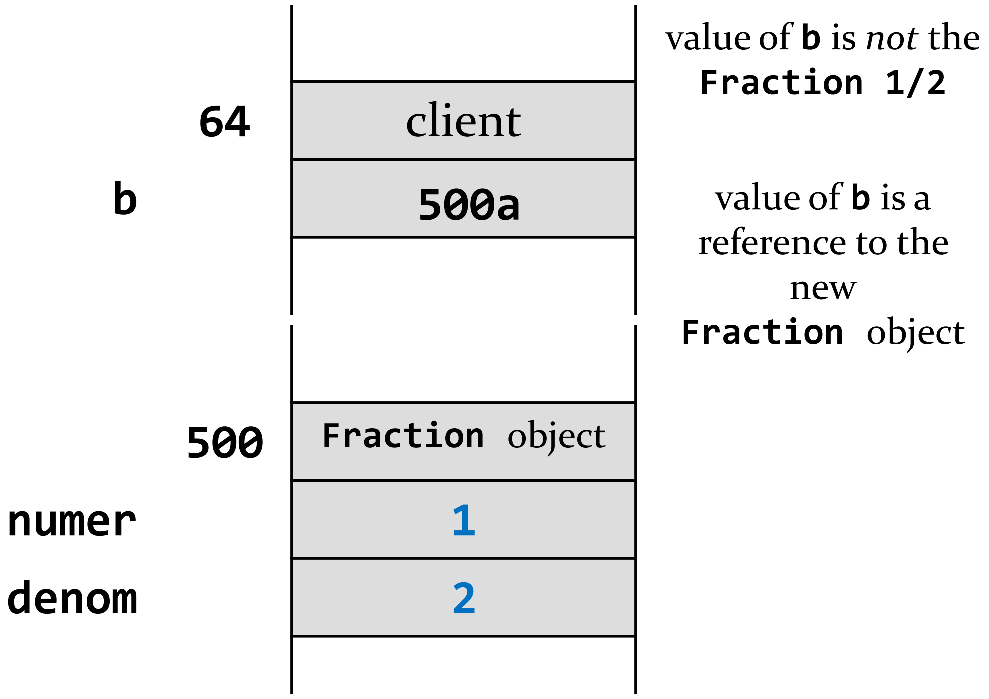
Pass-by-value with reference types

```
Fraction b =
  new Fraction(1, 2);
```


64
client
b
500a
the object at *address* 500
this is an address because b is a reference variable (refer to objects)
500
Fraction object
numer
1
denom
2

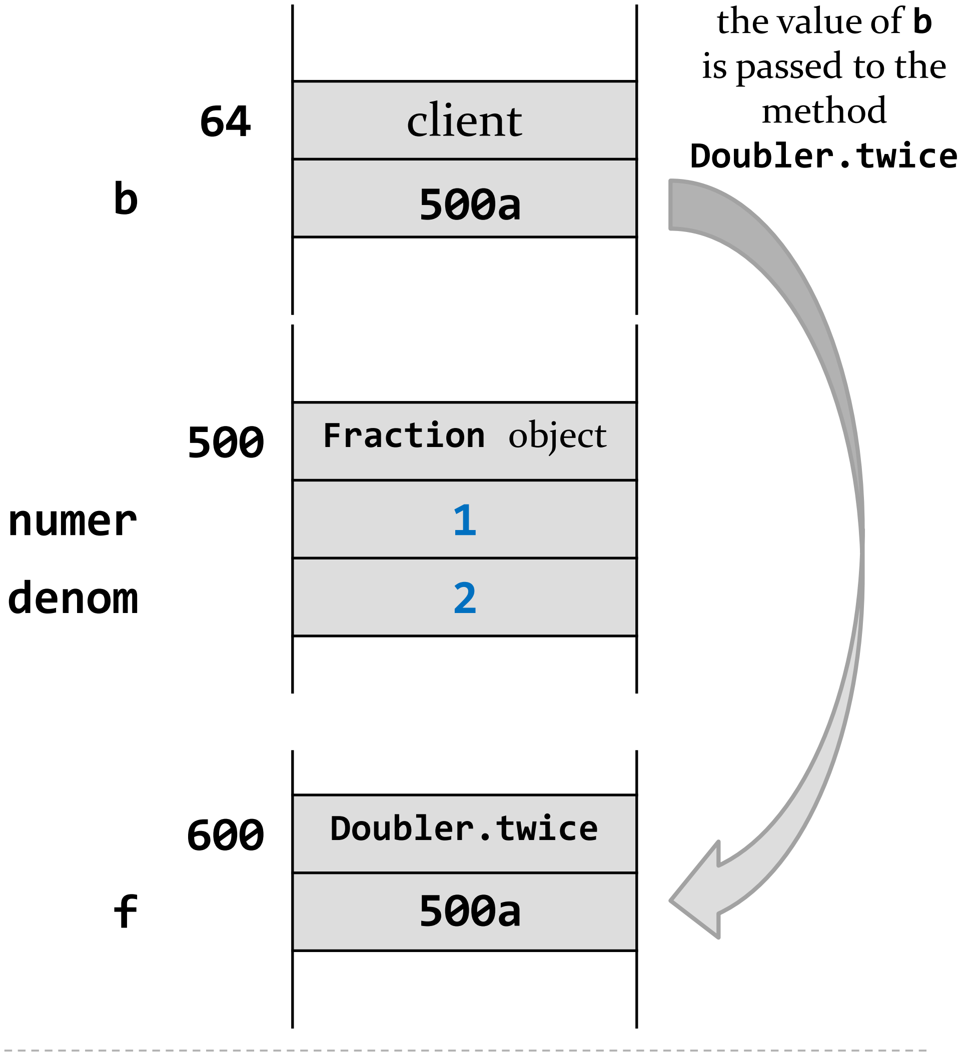
Pass-by-value with reference types

```
Fraction b =
   new Fraction(1, 2);
```


value of b is not the Fraction 1/2
64
client
b
500a
value of b is a reference to the new Fraction object
500
Fraction object
numer
1
denom
2

Pass-by-value with reference types

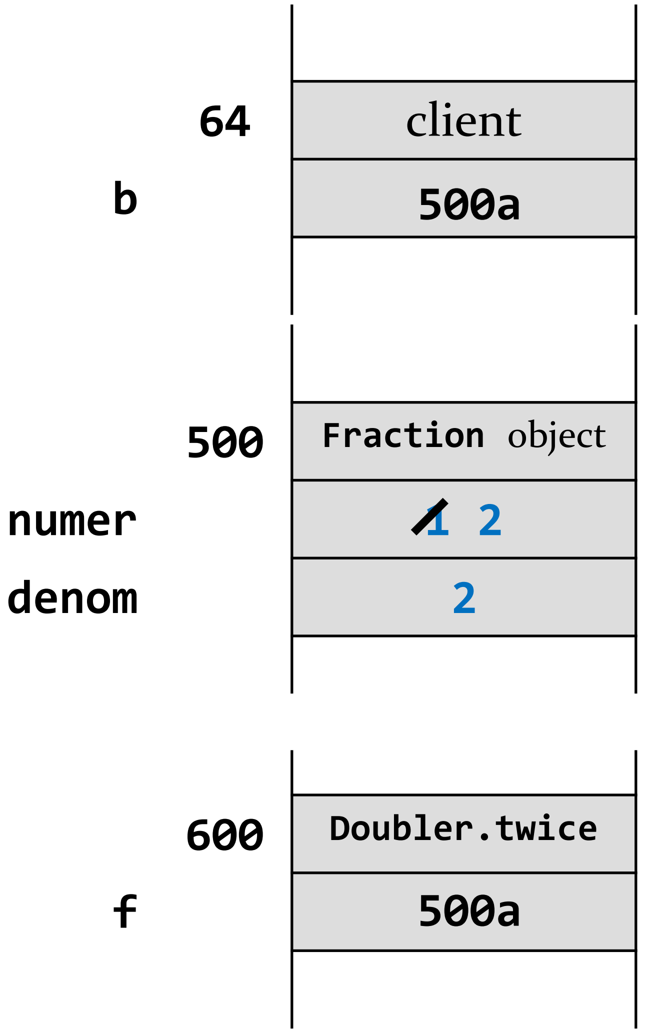
```
Fraction b =
  new Fraction(1, 2);
Doubler.twice(b);
```



parameter **f** *is an independent copy* of the value of argument **b** (a reference)

Pass-by-value with reference types

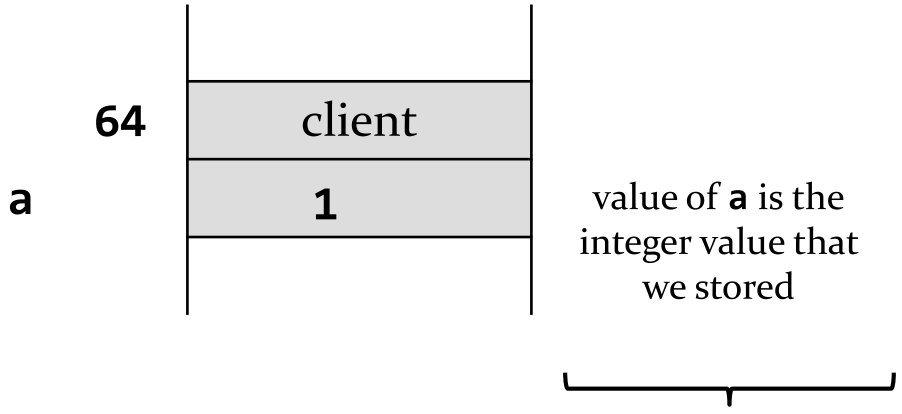
```
Fraction b =
  new Fraction(1, 2);
Doubler.twice(b);
```



Doubler.twice multiplies the numerator of the **Fraction** object by **2**

Pass-by-value with primitive types

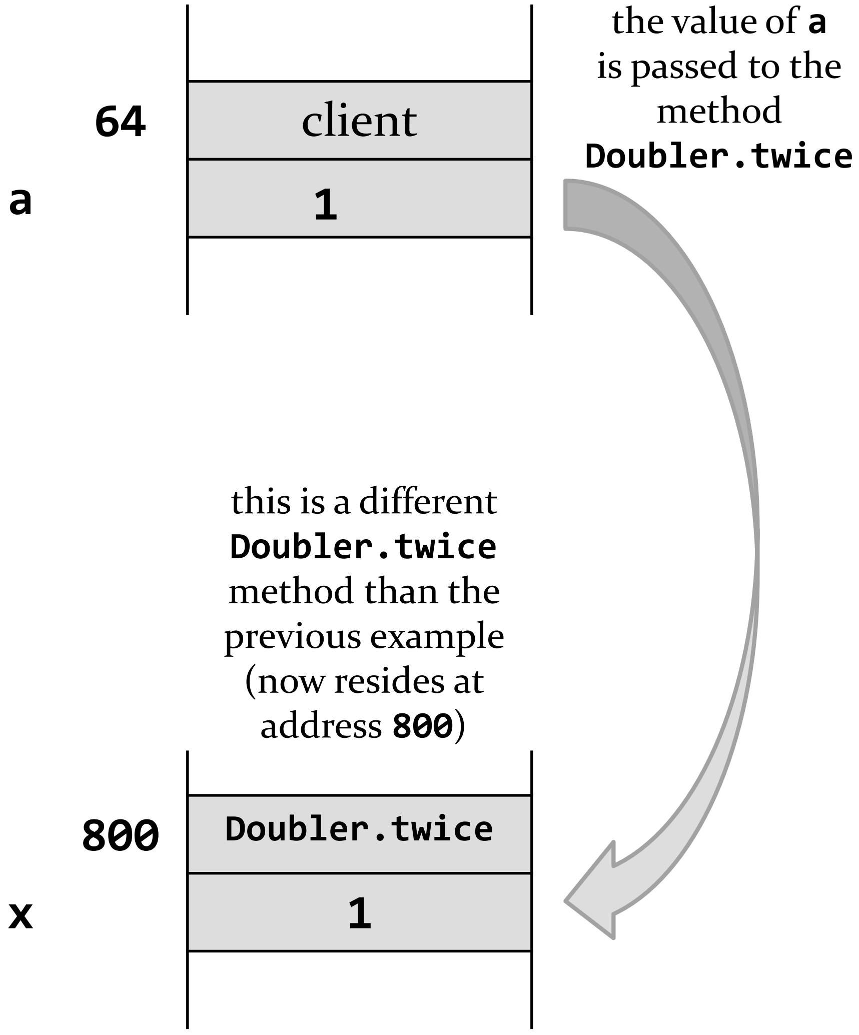
```
int a = 1;
```



this is the numeric value because **a** is a primitive variable

Pass-by-value with primitive types

```
int a = 1;
Doubler.twice(a);
```

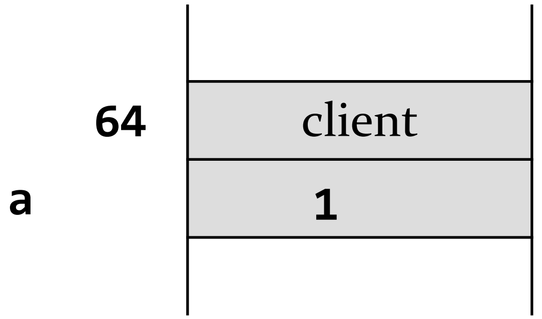


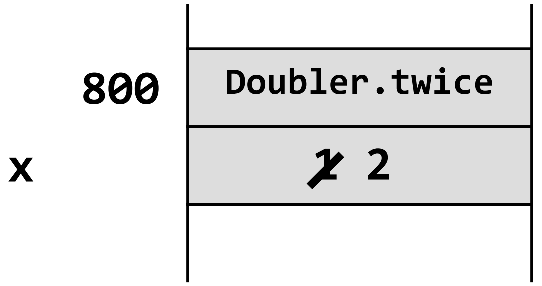
this is a different **Doubler.twice** method than the previous example (now resides at address **800**)

parameter **x** *is an independent copy* of the value of argument **a** (a primitive)

Pass-by-value with primitive types

```
int a = 1;
Doubler.twice(a);
```





`Doubler.twice` multiplies the value of **`x`** by **`2`**; that's it, nothing else happens

Pass-by-value

- Java uses pass-by-value for *all* types (primitive and reference)
  - an argument of primitive type cannot be changed by a method
  - an argument of reference type can have its state changed by a method
- pass-by-value is used to return a value from a method back to the client

Documenting a method

Javadoc

Documenting

- documenting code was not a new idea when Java was invented
  - however, Java was the first major language to embed documentation in the code and extract the documentation into readable electronic APIs

- the tool that generates API documents from comments embedded in the code is called Javadoc

Documenting

- Javadoc processes *doc comments* that immediately precede a class, attribute, constructor or method declaration
  - doc comments delimited by **`/**`** and **`*/`**
  - doc comment written in HTML and made up of two parts
    1. a description
       - ☐ first sentence of description gets copied to the summary section
       - ☐ only one description block; can use **`<p>`** to create separate paragraphs
    2. block tags
       - ☐ begin with **`@`** **`(@param`**, **`@return`**, **`@throws`** and many others)
       - ☐ **`@pre.`** is a non-standard (custom tag used in EECS1030) for documenting preconditions

Method documentation example

Eclipse will generate an empty Javadoc comment for you if you right-click on the method header and choose **Source→Generate Element Comment**

```
/**
 * @param min
 * @param max
 * @param value
 * @return
 */
public static boolean isBetween(int min, int max, int value) {
    // implementation not shown
}
```

Method documentation example

The first sentence of the documentation should be short summary of the method; this sentence appears in the method summary section.

```
/**
 * Returns true if value is strictly greater than min and strictly
 * less than max, and false otherwise.
 *
 * @param min
 * @param max
 * @param value
 * @return
 */
public static boolean isBetween(int min, int max, int value) {
    // implementation not shown
}
```

Method documentation example

You should provide a brief description of each parameter.

```
/**
 * Returns true if value is strictly greater than min and strictly
 * less than max, and false otherwise.
 *
 * @param min a minimum value
 * @param max a maximum value
 * @param value a value to check
 * @return
 */
public static boolean isBetween(int min, int max, int value) {
    // implementation not shown
}
```

Method documentation example

Provide a brief description of the return value if the return type is not void. This description often describes a postcondition of the method.

```
/**
 * Returns true if value is strictly greater than min and strictly
 * less than max, and false otherwise.
 *
 * @param min a minimum value
 * @param max a maximum value
 * @param value a value to check
 * @return true if value is strictly greater than min and strictly
 * less than max, and false otherwise
 */
public static boolean isBetween(int min, int max, int value) {
    // implementation not shown
}
```

Method documentation example

- if a method has one or more preconditions, you should use the EECS2030 specific **@pre.** tag to document them

Method documentation example

Describe any preconditions using the EECS2030 specific @pre. tag. You have to manually do this.

```
/**
 * Returns true if value is strictly greater than min and strictly
 * less than max, and false otherwise.
 *
 * @param min a minimum value
 * @param max a maximum value
 * @param value a value to check
 * @return true if value is strictly greater than min and strictly
 * less than max, and false otherwise
 * @pre min is less than or equal to max
 */
public static boolean isBetween(int min, int max, int value) {
    // implementation not shown
}
```

Method documentation example

- if a method throws an exception then you should use the **@throws** tag to document the exception

```
/**
 * Given a list containing exactly 2 integers, returns the smaller of the
 * two integers. The list <code>t</code> is not modified by this method.
 * For example:
 *
 * <pre>
 * t              Test2F.min2(t)
 * ---------------------------
 * [-5, 9]        -5
 * [3, 3]          3
 * [12, 6]         6
 * </pre>
 *
 * @pre t is not null
 * @param t a list containing exactly 2 integers
 * @return the minimum of the two values in t
 * @throws IllegalArgumentException if the list does not contain exactly 2
 * integers
 */
public static int min2(List<Integer> t) {
}
```

HTML markup is also allowed

Utility classes

Review: Java Class

- a class is a model of a thing or concept
- in Java, a class is usually a blueprint for creating objects
  - fields (or attributes)
    - the structure of an object; its components and the information (data) contained by the object
  - methods
    - the behaviour of an object; what an object can do

Utility classes

- sometimes, it is useful to create a class called a *utility class* that is not used to create objects
  - such classes have no constructors for a client to use to create objects
- in a utility class, all features are marked as being **`static`**
  - you use the class name to access these features
- examples of utility classes:
  - **`java.lang.Math`**
  - **`java.util.Arrays`**
  - **`java.util.Collections`**

Utility classes

- the purpose of a utility class is to group together related fields and methods where creating an object is not necessary

- **`java.lang.Math`**
  - groups mathematical constants and functions
  - do not need a **`Math`** object to compute the cosine of a number
- **`java.util.Collections`**
  - groups methods that operate on Java collections
  - do not need a **`Collections`** object to sort an existing **`List`**

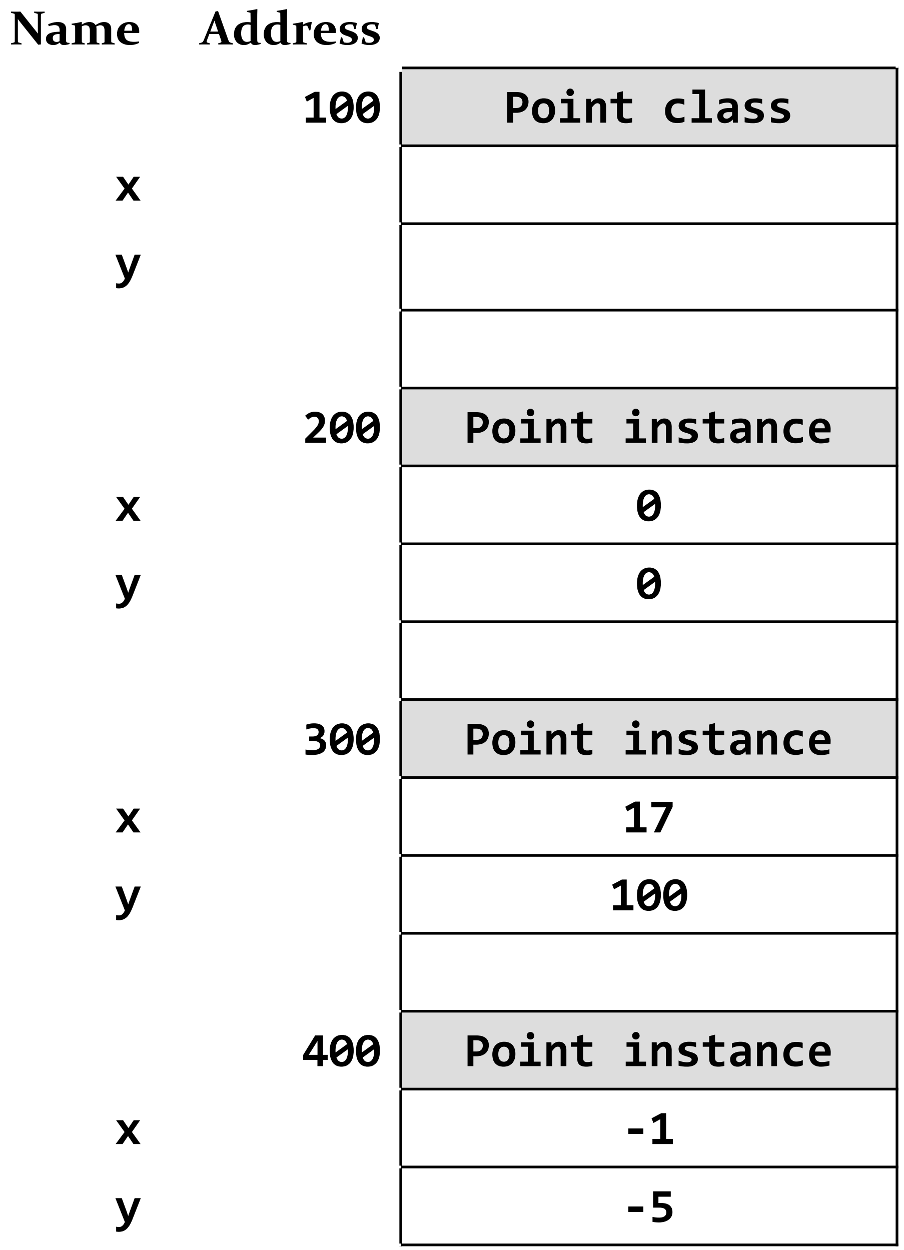
Class versus utility class

- a class is used to create *instances* of objects where each instance has its own *state*
- for example:
    - the class **`java.awt.Point`** is used to create instances that represent a location **`(x, y)`** where **`x`** and **`y`** are integers

```
public static void main(String[] args) {

  Point p = new Point(0, 0);     // point (0, 0)
  Point q = new Point(17, 100);  // point (17, 100)
  Point r = new Point(-1, -5);   // point (-1, -5)
}
```

    - each instance occupies a separate location in memory which we can illustrate in a memory diagram



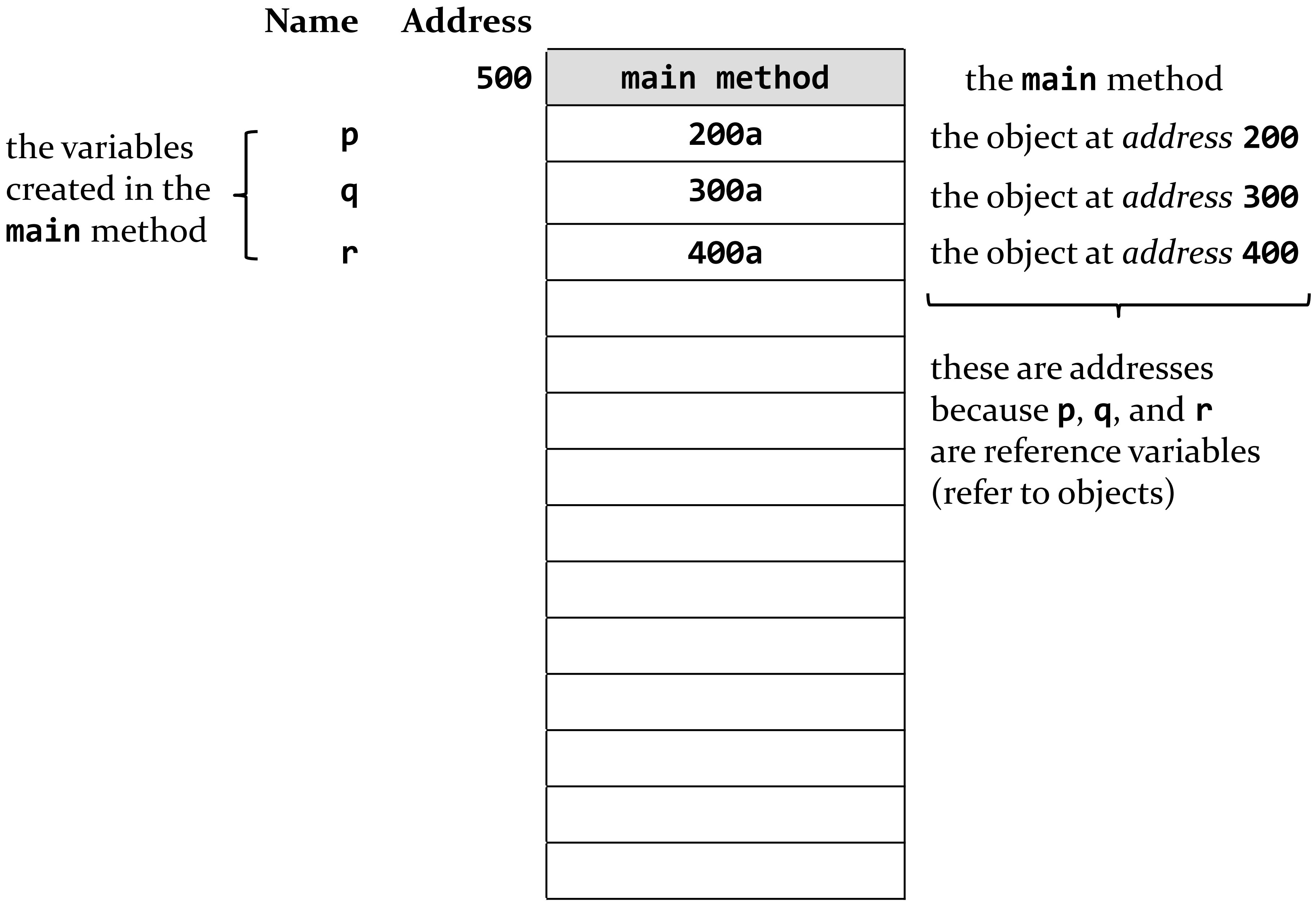
`Point` class is loaded into memory

`Point` instance with state **`(0, 0)`**

`Point` instance with state **`(17, 100)`**

`Point` instance with state **`(-1, -5)`**

continued on next slide


Name
Address
500
main method
the main method
the variables created in the main method
p
q
r
200a
300a
400a
the object at address 200
the object at address 300
the object at address 400
these are addresses because p, q, and r are reference variables (refer to objects)

Class versus utility class

- a utility class is *never* used to create objects
- when you use a utility class only the class itself occupies any memory

```
public static void main(String[] args) {

  double x = Math.cos(Math.PI / 3.0);
  double y = Math.sin(Math.PI / 3.0);

  // notice that we never created a Math object
}
```

Name	Address	
	100	`Math class`
PI		**3.1415....**
E		**2.7182....**
	200	`main method`
x		**0.8660....**
y		**0.5**

`Math` class is loaded into memory but there are no **`Math`** instances

the *value* **`cos(π/3)`**

the *value* **`sin(π/3)`**

these are values (not addresses) because **`x`** and **`y`** are primitive variables (**`double`**)

A simple utility class

- implement a utility class that helps you calculate Einstein's famous mass-energy equivalence equation $E = mc^2$ where
  - m is mass (in kilograms)
  - c is the speed of light (in metres per second)
  - E is energy (in joules)

Start by creating a package, giving the class a name, and creating the class body block.

```java
package ca.yorku.eecs.eecs2030;

public class Relativity {







}
```

Add a field that represents the speed of light.

```
package ca.yorku.eecs.eecs2030;

public class Relativity {

  public static final double C = 299792458;




}
```

Add a method to compute $E = mc^2$.

```java
package ca.yorku.eecs.eecs2030;

public class Relativity {

  public static final double C = 299792458;

  public static double massEnergy(double mass) {
    double energy = mass * Relativity.C * Relativity.C;
    return energy;
  }

}
```

Add a method to compute $E = mc^2$.

```
package ca.yorku.eecs.eecs2030;

public class Relativity {

  public static final double C = 299792458;

  public static double massEnergy(double mass) {
    double energy = mass * Relativity.C * Relativity.C;
    return energy;
  }

}
```

Here's a program that uses (a client) the **`Relativity`** utility class.

```
package ca.yorku.eecs.eecs2030;

public class OneGram {

  public static void main(String[] args) {
    double mass = 0.001;
    double energy = Relativity.massEnergy(mass);
    System.out.println("1 gram = " + energy + " Joules");
  }

}
```

Fields

```java
public static final double C = 299792458;
```

- a field is a member that holds data
- a constant field is usually declared by specifying
  1. modifiers
     1. access modifier `public`
     2. static modifier `static`
     3. final modifier `final`
  2. type `double`
  3. name `C`
  4. value `299792458`

Fields

- field names must be unique in a class
- the scope of a field is the entire class
- [notes] use the term “field” only for **`public`** fields

`public` Fields

- a **`public`** field is visible to all clients

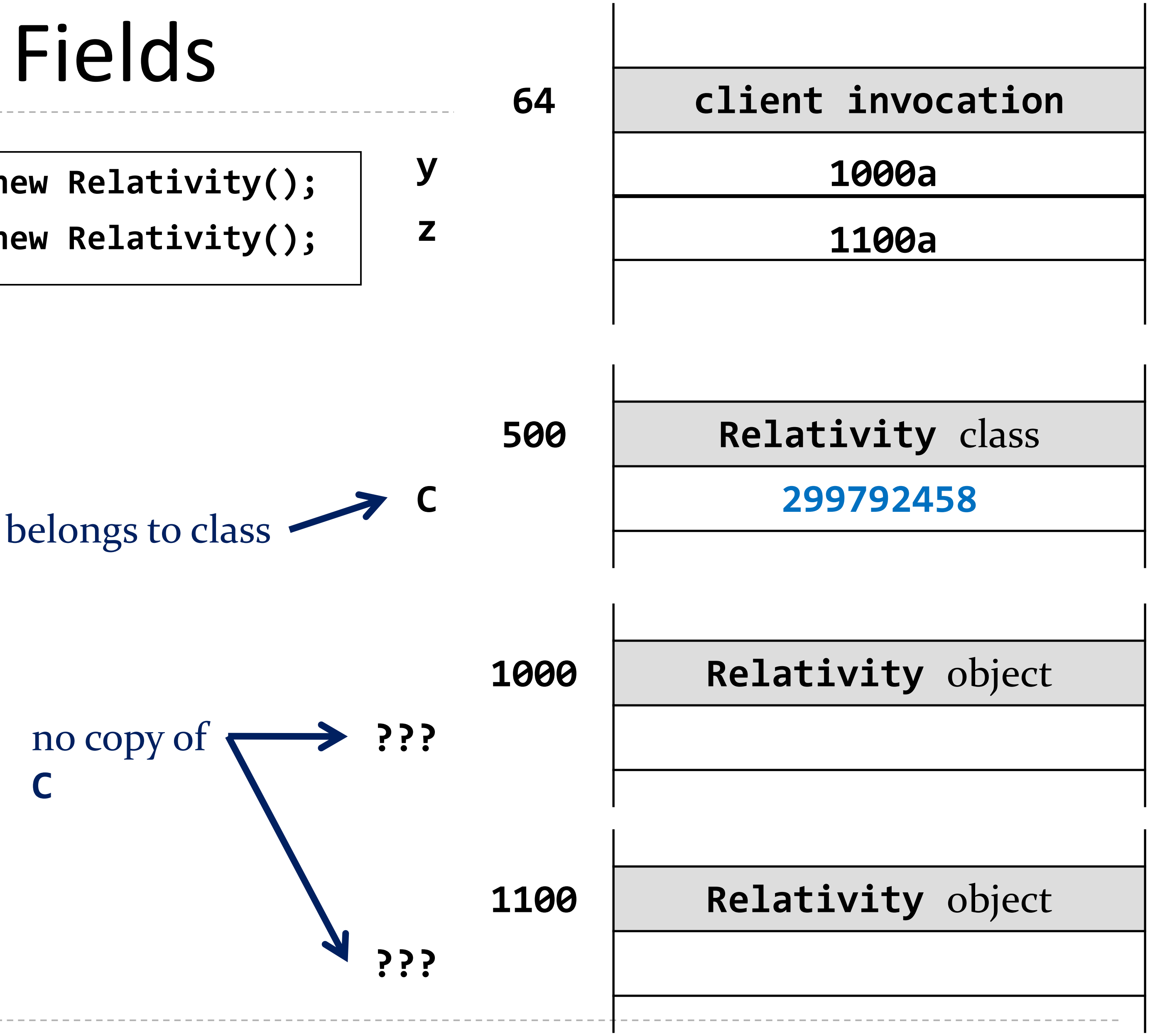
```
// client of Relativity
int speedOfLight = Relativity.C;
```

`static` Fields

- a field that is **`static`** is a per-class member
  - only one copy of the field, and the field is associated with the class
    - every object created from a class declaring a static field shares the same copy of the field
  - textbook uses the term *static variable*
  - also commonly called *class variable*

`static` Fields

```
Relativity y = new Relativity();
Relativity z = new Relativity();
```



static Field Client Access

- a client should access a **public static** field without using an object reference
  - use the class name followed by a period followed by the attribute name

```
public static void main(String[] args) {
    double sunDistance = 149.6 * 1e9;
    double seconds = sunDistance / Relativity.C;
    System.out.println(
        "time for light to travel from sun to earth " +
        seconds + " seconds");
}
```

```
time for light to travel from sun to earth 499.01188641643546 seconds
```

`static` Attribute Client Access

- it is legal, *but considered bad form*, to access a **`public static`** attribute using an object

```
public static void main(String[] args) {
    double sunDistance = 149.6 * 1e9;
    Relativity y = new Relativity();
    double seconds = sunDistance / y.C;
    System.out.println(
        "time for light to travel from sun to earth " +
        seconds + " seconds");
}
```

```
time for light to travel from sun to earth 499.01188641643546 seconds
```

`final` Fields

- a field that is **`final`** can only be assigned to once
  - **`public static final`** fields are typically assigned when they are declared

```
public static final double C = 299792458;
```

  - **`public static final`** fields are intended to be constant values that are a meaningful part of the abstraction provided by the class

`final` Fields of Primitive Types

- **`final`** fields of primitive types are constant

```
public class Relativity {
  public static final double C = 299792458;
}
```

```
// client of Relativity
public static void main(String[] args) {

  Relativity.C = 100;  // will not compile;
                       // field C
                       // is final and
                      // previously assigned
}
```

`final` Fields of Immutable Types

- **`final`** fields of immutable types are constant

```
public class NothingToHide {
  public static final String X = "peek-a-boo";
}
```

```
// client of NothingToHide
public static void main(String[] args) {
  NothingToHide.X = "i-see-you";
                           // will not compile;
                             // field X is final and
                             // previously assigned
}
```

- **`String`** is immutable
  - it has no methods to change its contents

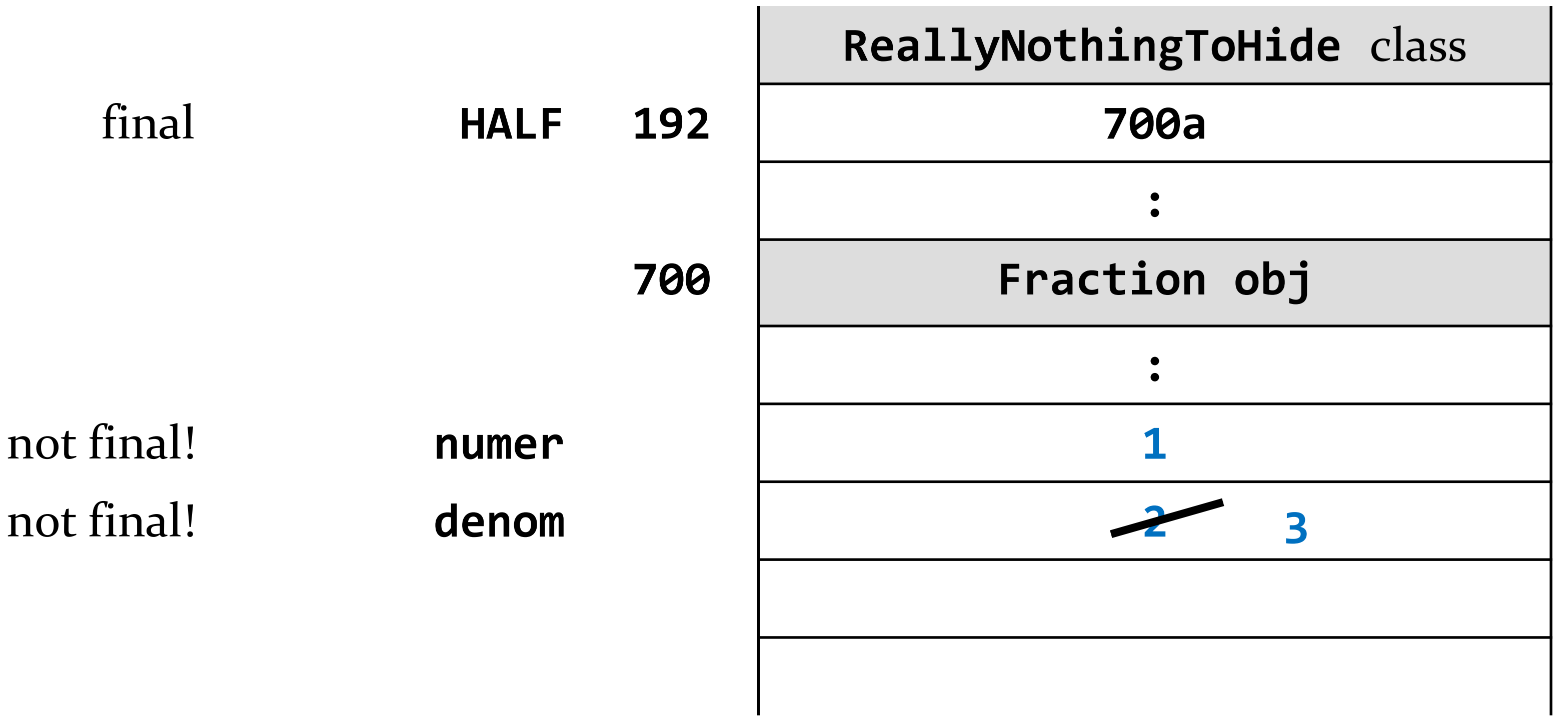
`final` Fields of Mutable Types

- **`final`** fields of mutable types are not logically constant; their state can be changed

```
public class ReallyNothingToHide {
  public static final Fraction HALF =
                                   new Fraction(1, 2);
}
```

```
// client of ReallyNothingToHide
public static void main(String[] args) {
  ReallyNothingToHide.HALF.setDenominator(3);
                                   // works!!
                                   // HALF is now 1/3
}
```

`final` Fields of Mutable Types



```
ReallyNothingToHide.HALF.setDenominator(3);
```

`final` fields

- avoid using mutable types as **`public`** constants
  - they are not logically constant

`new Relativity` objects

- our **`Relativity`** class does not expose a constructor
  - but

    ```
    Relativity y = new Relativity();
    ```

    is legal

- if you do not define any constructors, Java will generate a default no-argument constructor for you
  - e.g., we get the **`public`** constructor

    ```
    public Relativity() { }
    ```

    even though we did not implement it

Preventing instantiation

- in a utility class you can prevent a client from making new instances of your class by declaring a **`private`** constructor

- a **`private`** field, constructor, or method can only be used inside the class that it is declared in

```java
package ca.yorku.eecs.eecs2030;

public class Relativity {

  public static final double C = 299792458;

  private Relativity() {
    // private and empty by design
  }

  public static double massEnergy(double mass) {
    double energy = mass * Relativity.C * Relativity.C;
    return energy;
  }

}
```

Preventing instantiation

- every utility class should have a private empty no-argument constructor to prevent clients from making objects using the utility class

Introduction to Testing

Testing

- testing code is a vital part of the development process
- the goal of testing is to find defects in your code
  - Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
    —Edsger W. Dijkstra

Testing with a main method

- if I had asked you to test your worksheet 1 methods you probably would have written a main method

```java
public static void main(String[] args) {
    // avg
    int a = 1;
    int b = 1;
    int c = 1;
    System.out.println(
        String.format("average of %d, %d, and %d : ", a, b, c) +
                     Test2E.avg(a, b, c));

    // swap2
    List<Integer> t = new ArrayList<Integer>();
    t.add(3);
    t.add(5);
    String s = t.toString();
    Test2E.swap2(t);
    System.out.println(
        String.format("swap2(%s) : %s", s, t.toString()));
        // allGreaterThan
        t.clear();
        t.add(4);
        t.add(5);
        t.add(6);
        t.add(7);
        t.add(8);
        System.out.println(
             String.format("allGreaterThan(%s, %s) : %s",
                              t.toString(), 5, Test2E.allGreaterThan(t, 5)));

        // toInt
        t.clear();
        t.add(1);
        t.add(2);
        t.add(3);
        System.out.println(
            String.format("toInt(%s) : %d",
                            t.toString(), Test2E.toInt(t)));
    }
```

Testing with a main method

- running the main method results in the following output:

```
average of 1, 1, and 1 : 1.0
swap2([3, 5]) : [5, 3]
allGreaterThan([4, 5, 6, 7, 8], 5) : [6, 7, 8]
toInt([1, 2, 3]) : 123
```

Testing with a main method

- testing using a single main method has some disadvantages:
  - someone has to examine the output to determine if the tests have passed or failed
  - all of the tests are in one method
    - we can’t run tests independently from one another
    - there is no easy way to pick which tests we want to run

JUnit

- JUnit is a unit test framework
- "A framework is a semi-complete application. A framework provides a reusable, common structure to share among applications."
  - from the book JUnit in Action

JUnit

- “A unit test examines the behavior of a distinct unit of work. Within a Java application, the “distinct unit of work” is often (but not always) a single method. … A unit of work is a task that isn't directly dependent on the completion of any other task.”
  - from the book JUnit in Action

A JUnit test example

- let's write a test for the worksheet 1 method avg

- we need a class to write the test in
- we need to import the JUnit library
- we need to write a method that implements the test

- happily, eclipse helps you do all of this
  - in the Package Explorer, right click on the class that you want to test and select New > JUnit Test Case

```java
package eecs2030.test2;

import static org.junit.Assert.*;
import org.junit.Test;

public class Test2ETest {

    @Test
    public void test_avg() {
        int a = -99;
        int b = 100;
        int c = -11;
        double expected = -10.0 / 3;
        double actual = Test2E.avg(a, b, c);
        double delta = 1e-9;
        assertEquals(expected, actual, delta);
    }
```

static import: allows you to use static methods from the class org.junit.Assert without specifying the class name

Avoid the widespread use of static imports. Although it is convenient being able to not include the class name in front of the method name, it makes it difficult to tell which class the method comes from*.

*https://docs.oracle.com/javase/8/docs/technotes/guides/language/static-import.html

```java
package eecs2030.test2;

import static org.junit.Assert.*;
import org.junit.Test;

public class Test2ETest {

    @Test
    public void test_avg() {
        int a = -99;
        int b = 100;
        int c = -11;
        double expected = -10.0 / 3;
        double actual = Test2E.avg(a, b, c);
        double delta = 1e-9;
        assertEquals(expected, actual, delta);
    }
```

An annotation; JUnit uses the @Test annotation to determine which methods are unit tests.

```
package eecs2030.test2;

import static org.junit.Assert.*;
import org.junit.Test;

public class Test2ETest {

    @Test
    public void test_avg() {
        int a = -99;
        int b = 100;
        int c = -11;
        double expected = -10.0 / 3;
        double actual = Test2E.avg(a, b, c);
        double delta = 1e-9;
        assertEquals(expected, actual, delta);
    }
```

A JUnit method that throws an exception if `expected` and `actual` differ by more than `delta`. JUnit handles the exception and reports the test failure to the user.

A JUnit test example

- consider testing **`swap2`**
  - **`swap2`** does not return a value
  - **`swap2`** modifies the state of the argument list
    - therefore, we need to test that the argument list has the expected state after **`swap2`** finishes running

- a method that modifies the state of an argument to the method is said to have a *side effect*

```
@Test
public void test_swap2() {
    List<Integer> actual = new ArrayList<Integer>();
    actual.add(-99);
    actual.add(88);

    List<Integer> expected = new ArrayList<Integer>();
    expected.add(88);
    expected.add(-99);

    Test2E.swap2(actual);
    assertEquals(expected, actual);
}
```

A JUnit method that throws an exception if `expected` and `actual` are not equal. JUnit handles the exception and reports the test failure to the user.

Creating tests

- based on the previous example, when you write a test in you need to determine:
  - what arguments to pass to the method
  - what the expected return value is when you call the method with your chosen arguments
    - if the method does not return a value then you need to determine what the expected results are of calling the method with your chosen arguments

Creating tests

- for now, we will define a *test case* to be:
  - a specific set of arguments to pass to the method
  - the expected return value (if any) and the expected results when the method is called with the specified arguments

Creating tests

- to write a test for a static method in a utility class you need to consider:
  - the preconditions of the method
  - the postconditions of the method
  - what exceptions the method might throw

Creating tests: Preconditions

- recall that method preconditions often place restrictions on the values that a client can use for arguments to the method

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`

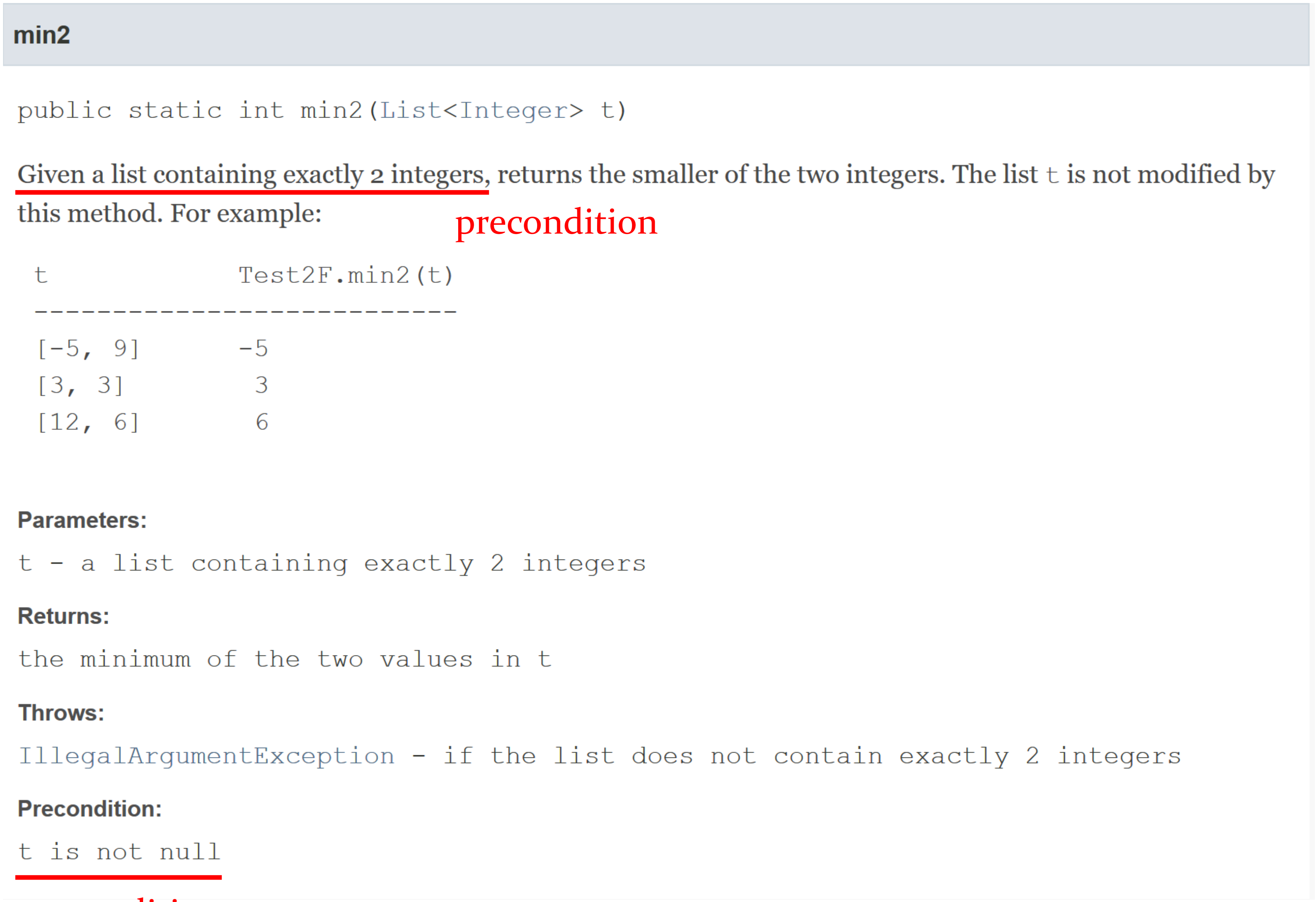
Returns:

`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
```

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

Creating tests: Preconditions

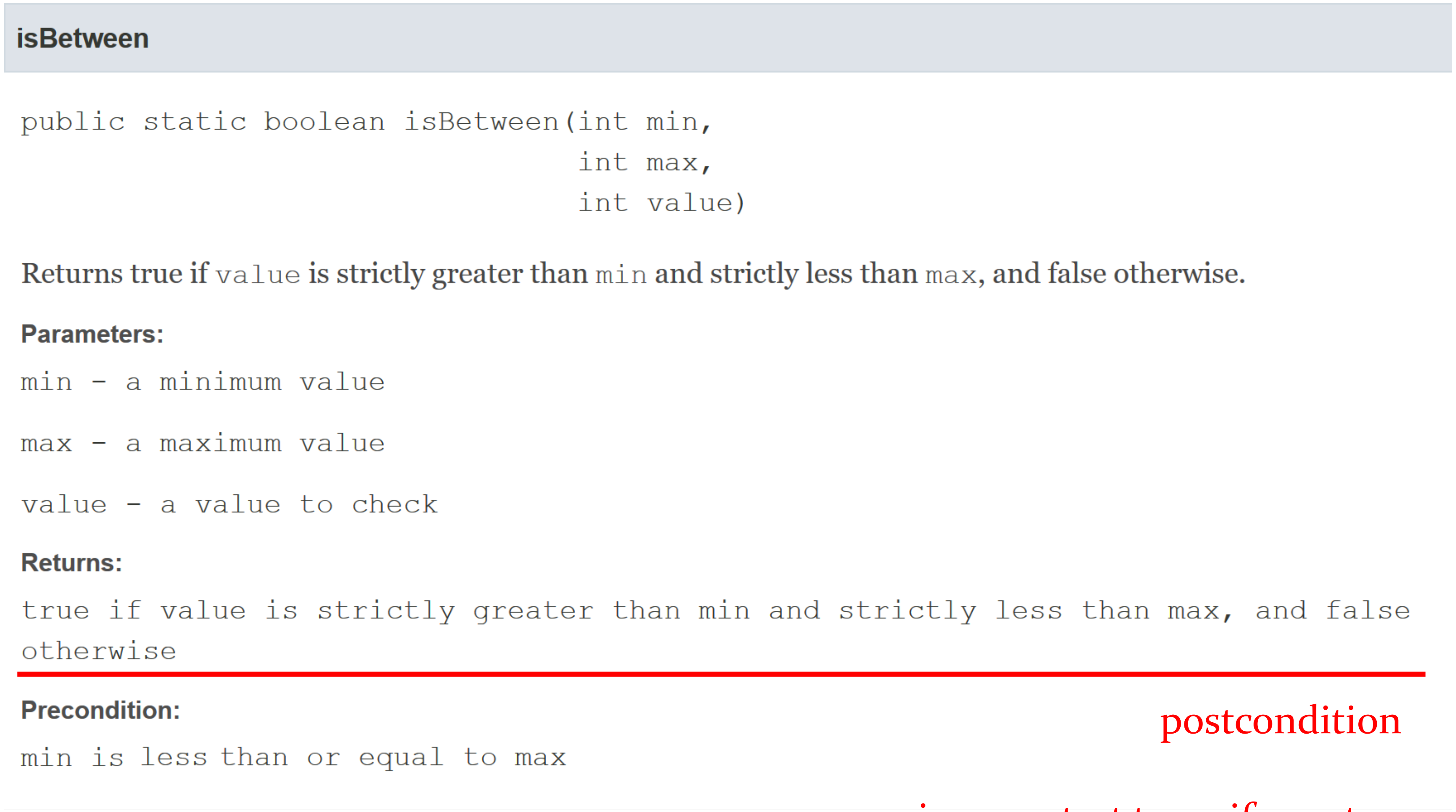
- the arguments you choose for the test should satisfy the preconditions of the method
  - but see the slides on testing exceptions!

- it doesn't make sense to use arguments that violate the preconditions because the postconditions are not guaranteed if you violate the preconditions

Creating tests: Postconditions

- recall that a postcondition is what the method promises will be true after the method completes running

- a test should confirm that the postconditions are true
- many postconditions require more than one test to verify



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:

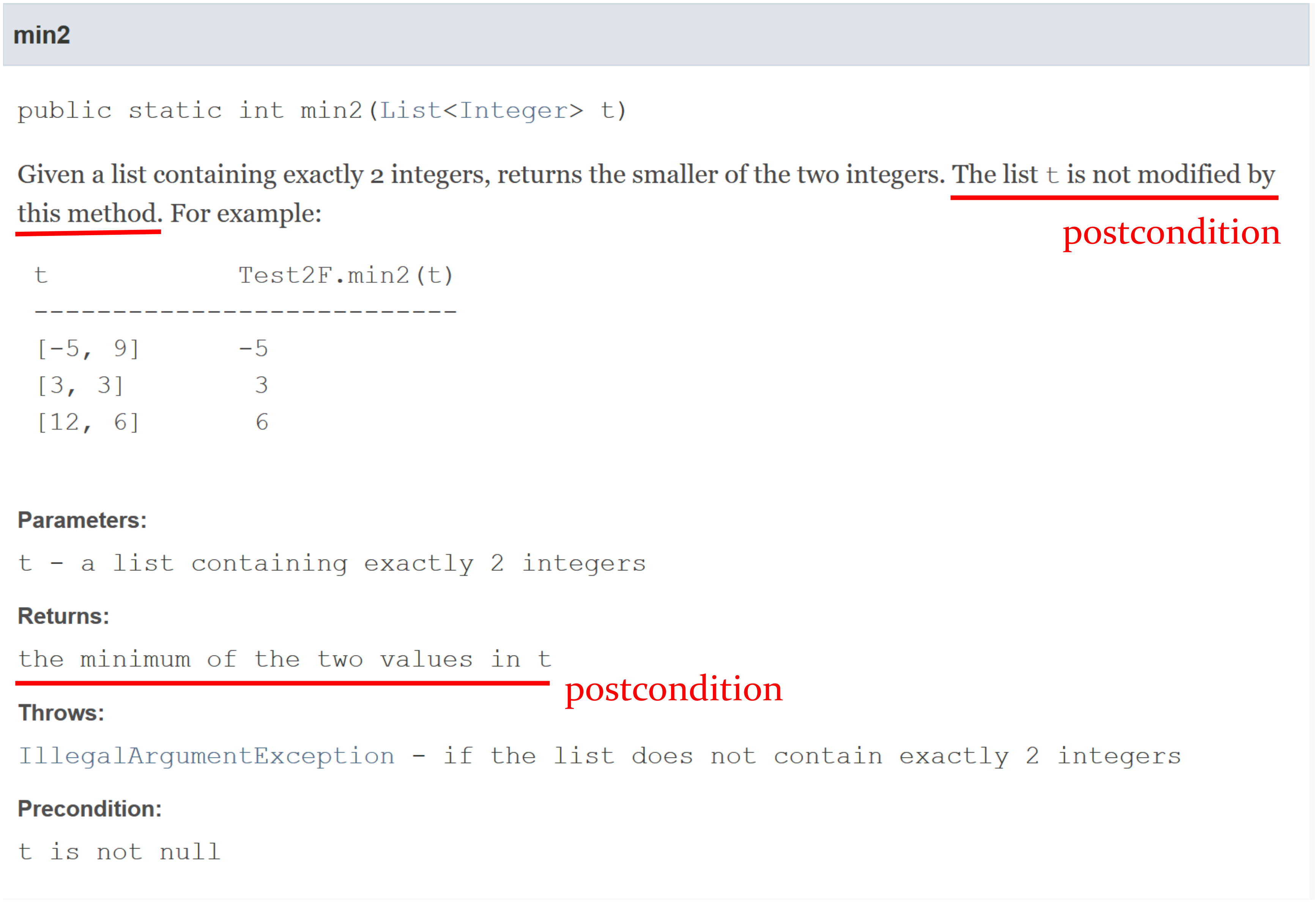
`true if value is strictly greater than min and strictly less than max, and false otherwise`

Precondition:

`min is less than or equal to max`

postcondition

requires one test to verify a return value of true and a second test to verify a return value for false



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
```

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`

Creating tests: Exceptions

- some methods having preconditions throw an exception if a precondition is violated
- if the API for the method states that an exception is thrown under certain circumstances then you should test those circumstances
  - even if writing such a test requires violating a precondition

```
@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws() {
    List<Integer> t = new ArrayList<Integer>();
    Test2E.swap2(t);
}

@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws2() {
    List<Integer> t = new ArrayList<Integer>();
    t.add(10000);
    Test2E.swap2(t);
}
```

A JUnit test that is expected to result in an **`IllegalArgumentException`** being thrown. The test fails if an **`IllegalArgumentException`** is not thrown.

```
@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws() {
    List<Integer> t = new ArrayList<Integer>();
    Test2E.swap2(t);
}

@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws2() {
    List<Integer> t = new ArrayList<Integer>();
    t.add(10000);
    Test2E.swap2(t);
}
```

swap2 should throw an exception because `t` is empty.

```
@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws() {
    List<Integer> t = new ArrayList<Integer>();
    Test2E.swap2(t);
}

@Test(expected = IllegalArgumentException.class)
public void test_swap2_throws2() {
    List<Integer> t = new ArrayList<Integer>();
    t.add(10000);
    Test2E.swap2(t);
}
```

swap2 should throw an exception because `t` has only one element.

Choosing test cases

- typically, you use several test cases to test a method
  - the course notes uses the term *test vector* to refer to a collection of test cases
- it is usually impossible or impractical to test all possible sets of arguments

Choosing test cases

- when choosing tests cases, you should consider using
  - arguments that have typical (not unusual) values, and
  - arguments that test boundary cases
    - argument value around the minimum or maximum value allowed by the preconditions
    - argument value around a value where the behavior of the method changes

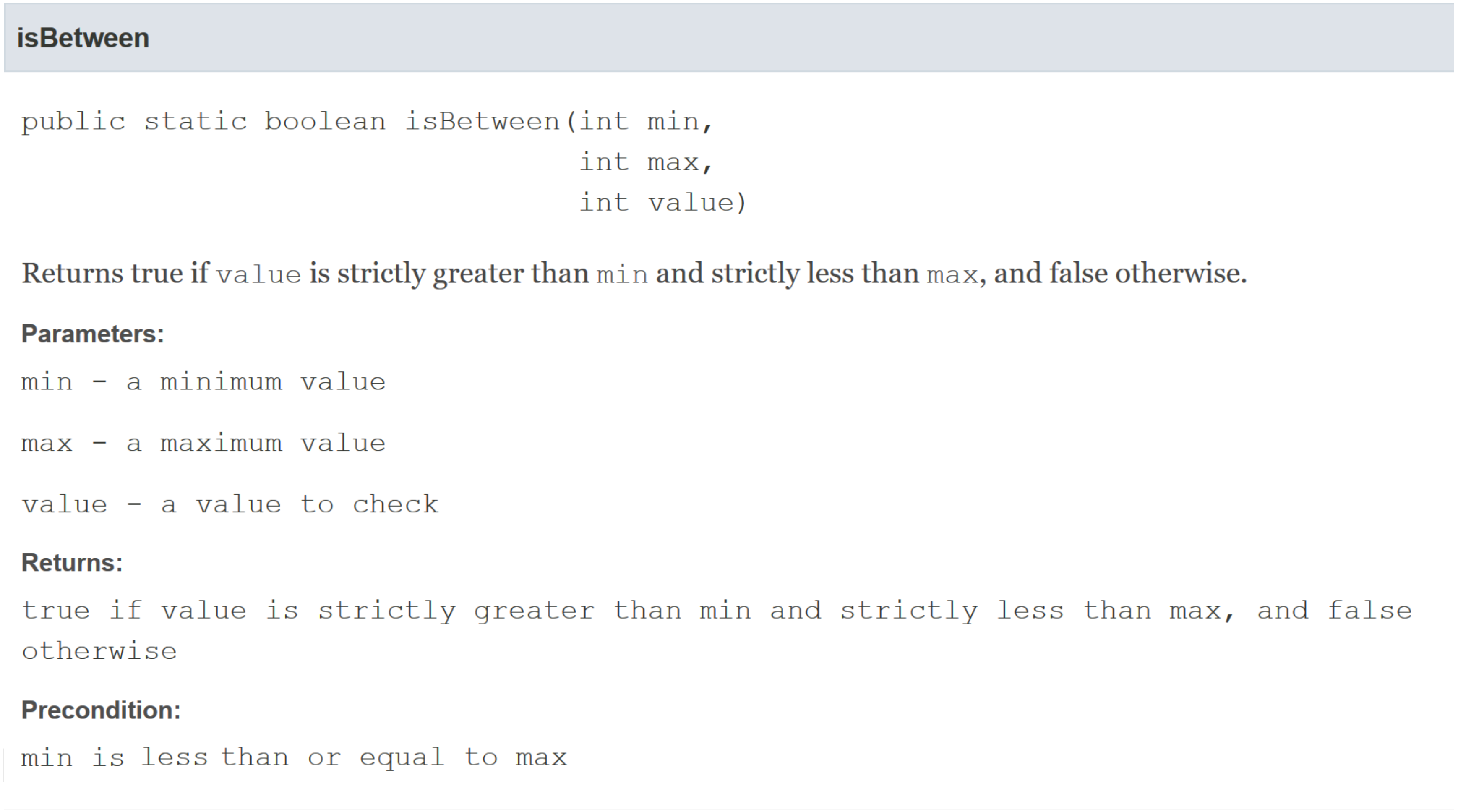
Example of a boundary case

- consider testing the method **`avg`**
- the method has no preconditions
- the boundary values of the arguments **`a`**, **`b`**, and **`c`** are **`Integer.MAX_VALUE`** and **`Integer.MIN_VALUE`**

```
@Test
public void test_avg_boundary() {
    int a = Integer.MAX_VALUE;
    int b = Integer.MAX_VALUE;
    int c = Integer.MAX_VALUE;
    double expected = Integer.MAX_VALUE;
    double actual = Test2E.avg(a, b, c);
    double delta = 1e-9;
    assertEquals(expected, actual, delta);
}
```

Example of a boundary case

- consider testing the method **`isBetween`**
- the method has a precondition that min <= max



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

Example of a boundary case

- boundary cases:
  - `value == min + 1`
    - expected return value: true
  - `value == min`
    - expected return value: false
  - `value == max`
    - expected return value: false
  - `value == max - 1`
    - expected return value: true
  - `min == max`
    - expected result: no exception thrown
  - `min == max - 1`
    - expected result: `IllegalArgumentException` thrown