LASSONDE
SCHOOL OF ENGINEERING

YORK
UNIVERSITÉ
UNIVERSITY

Advanced Object Oriented Programming

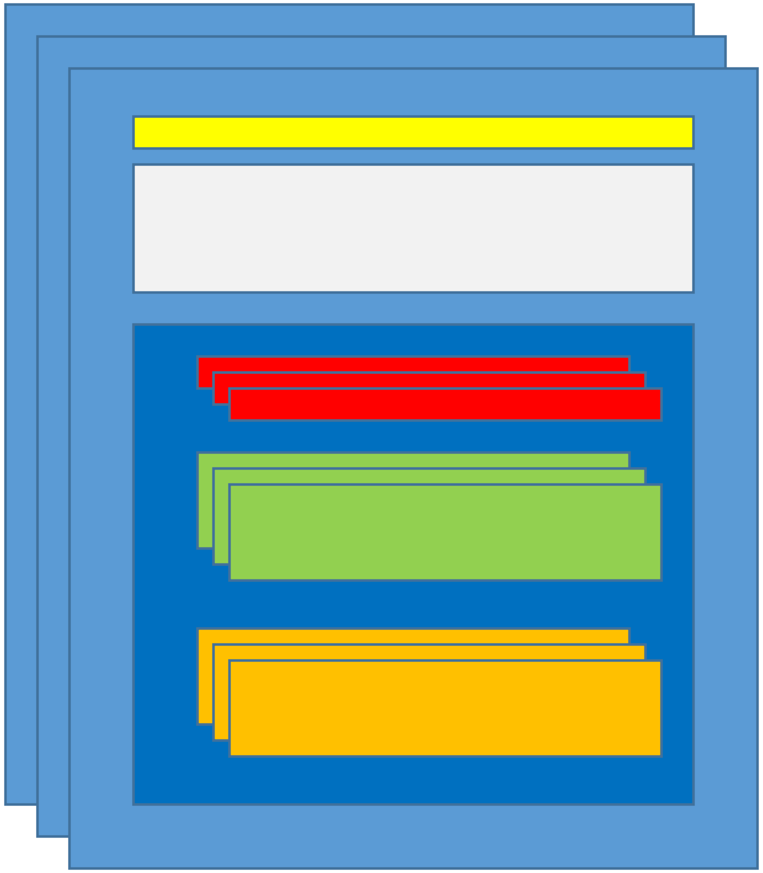
EECS2030

Section M

Organization of a Java Program

Packages, classes, fields, and methods

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

Packages

- Packages are used to organize Java classes into namespaces
- Packages are use to organize related classes and interfaces
  - e.g., all of the Java API classes are in the package named `java`

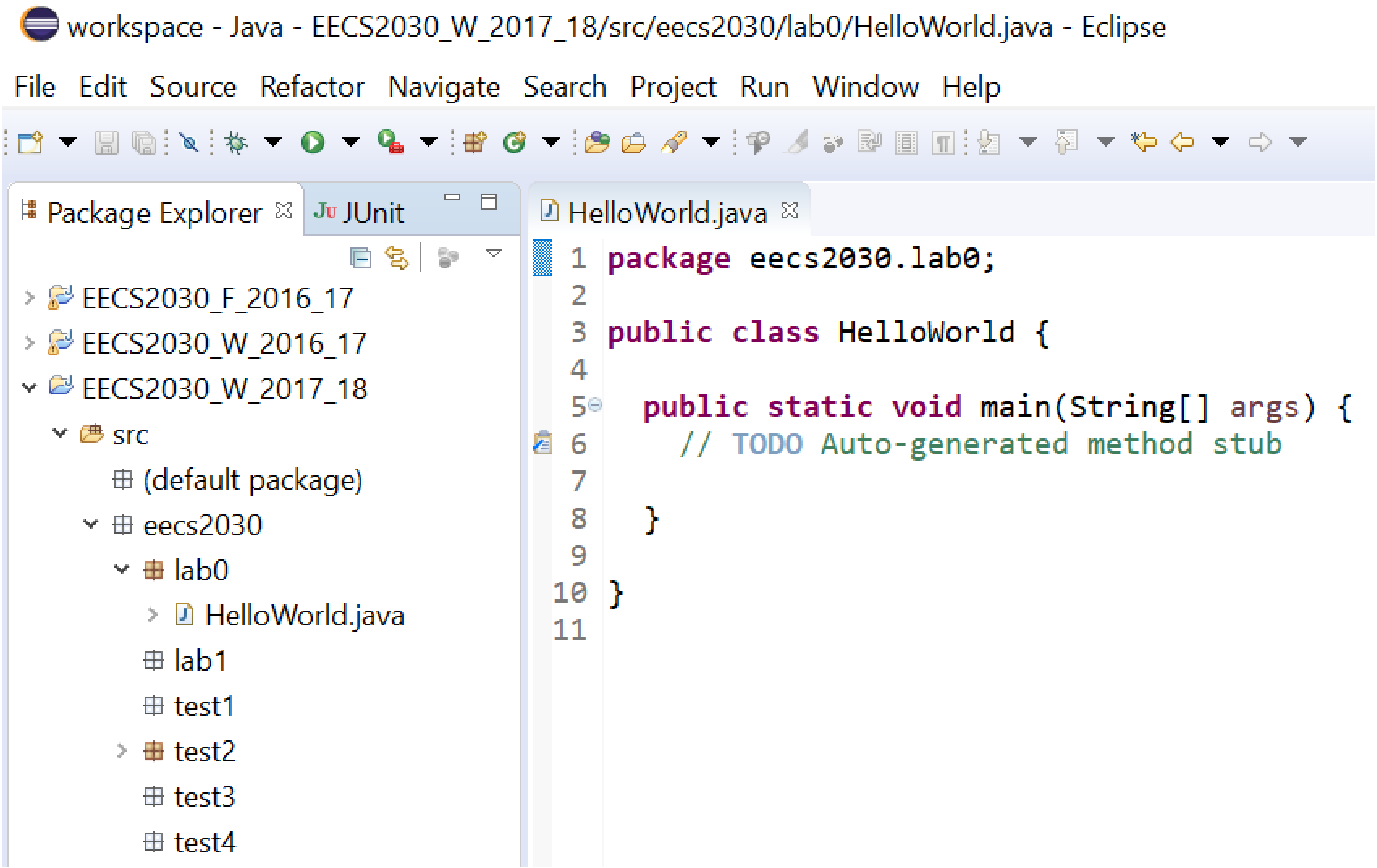
package ↔ directory (folder)

class ↔ file

General Overview of Java Packages API

- **`javax.swing`**: classes dealing with the development of GUIs.
- **`java.lang:`** **essential classes required by the Java language**.
- **`java.text`**: facilities for formatting text output.
- **`java.util`**: classes for storing/accessing collections of objects.
- **`java.net`**: for network communication.

Eclipse – Packages overview



```java
package eecs2030.lab0;

public class HelloWorld {

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

    }

}

```

project folder

project sources folder

eecs2030 folder

lab0 folder

To put a class into a package, one uses the "package" statement

https://docs.oracle.com/javase/specs/jls/se10/html/jls-7.html

The package statement

Syntax

```
package <top_pkg_name>[.<sub_pkg_name>]*;
```

Example

```
package java.lang;
public class String{
…}
```

- Statement **at the beginning** of the source file
- Only **one package declaration** per source file
- If **no package name** is declared → the class is placed into the ***default package***

The import statement

Syntax

```
import <pkg_name>[.<sub_pkg_name>]*.*;
```

Example

```
import java.util.List;
import java.io.*;
```

- Precedes all class declarations
- Tells the compiler ***where to find classes***

Importing a package

```
import packageName.*;   // all classes
import packageName.className;// one class
```

Notes on the import statement

- Import ONLY imports public classes from the specified package
  - Classes which are not public cannot be referenced from outside their package.
- There is no way to "*import all classes except one*"
  - import either imports a single class or all classes within the package
  - Note: importing has no runtime or performance implications. It is only importing a namespace so that the compiler can resolve class names.
- **Import statements must appear at the top of the file after the package statement and before any class or interface definitions.**

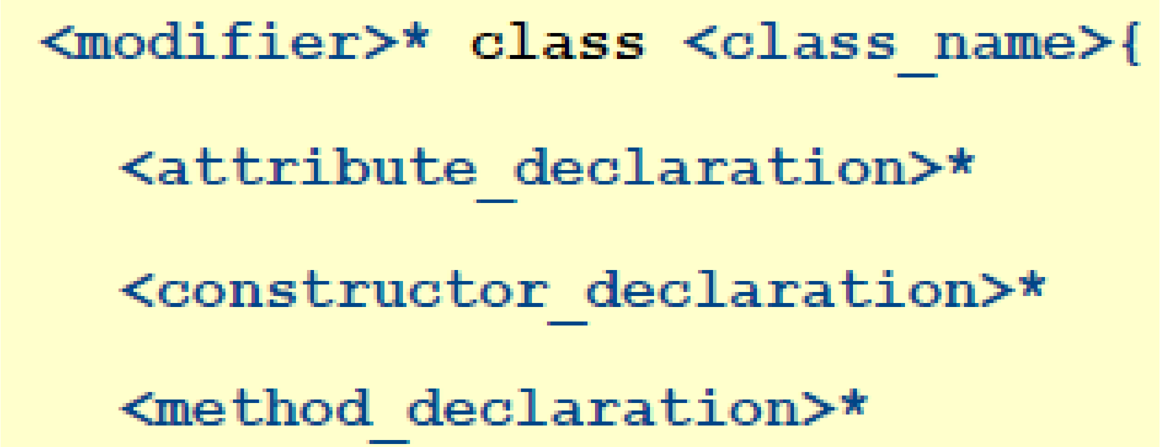
Objects in JAVA

Basics

In Java

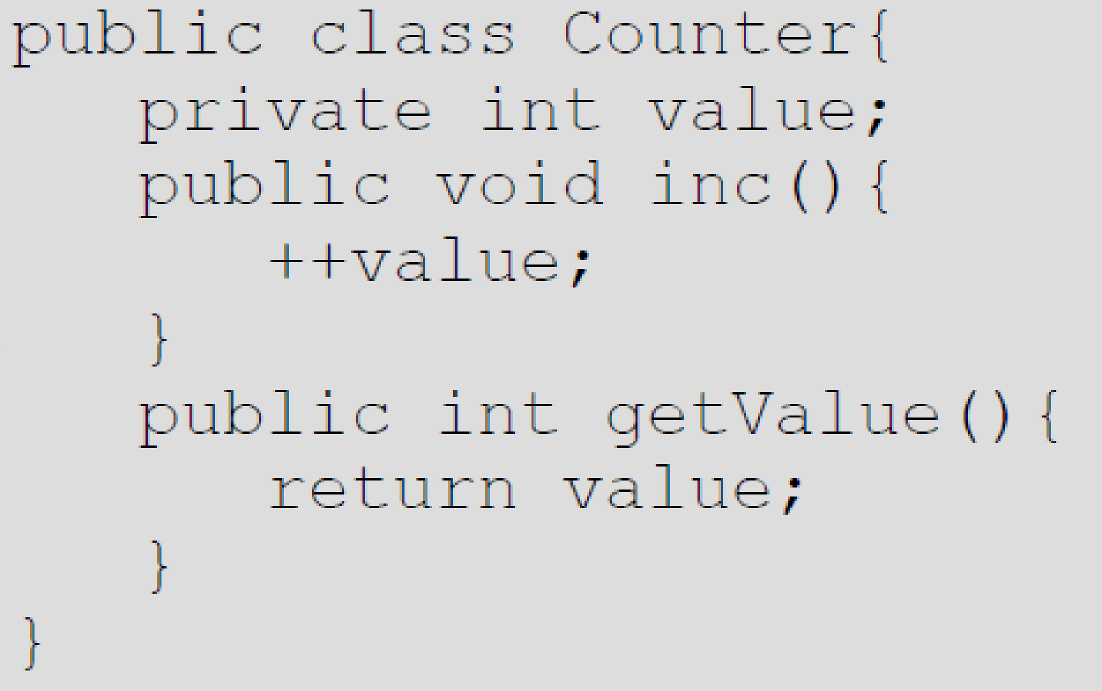
- **Class:** Is a **user-defined** type
  - Describes the *data* **(attributes)**
    - Also called **Variables, instance variables,** ***attributes,*** **fields.**
  - Defines the *behavior* (**methods**)
- Instances of a class are **objects**

Declaring Classes



```
<modifier>* class <class_name>{
    <attribute_declaration>*
    <constructor_declaration>*
    <method_declaration>*
}
```

Syntax



```
public class Counter{
    private int value;
    public void inc(){
        ++value;
    }
    public int getValue(){
        return value;
    }
}
```

Example

Overview

- An object can contain variables as well as methods.
- Variables and methods are called **members** of class.

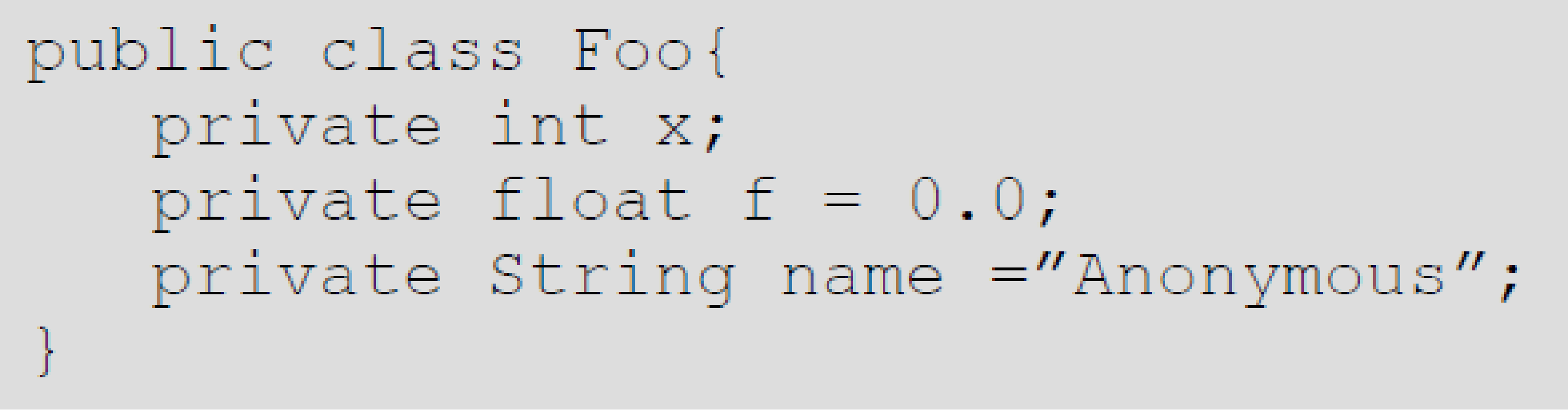
Note: ***Variable in an object is called a field*, *data*, *attributes or instance variables.***

Declaring Attributes/fields

Syntax

```
<modifier>* <type> <attribute_name>[= <initial_value>];
```

Example



```
public class Foo{
   private int x;
   private float f = 0.0;
   private String name =″Anonymous″;
}
```

- Generally, fields are defined as private so they can’t be seen from outside the class.
- May add getter methods (functions) and setter methods (procedures) to allow access to some or all fields.
- *We use constructors, to initialize fields of a new object during evaluation of a new-expression.*

Non-static classes

Utility class

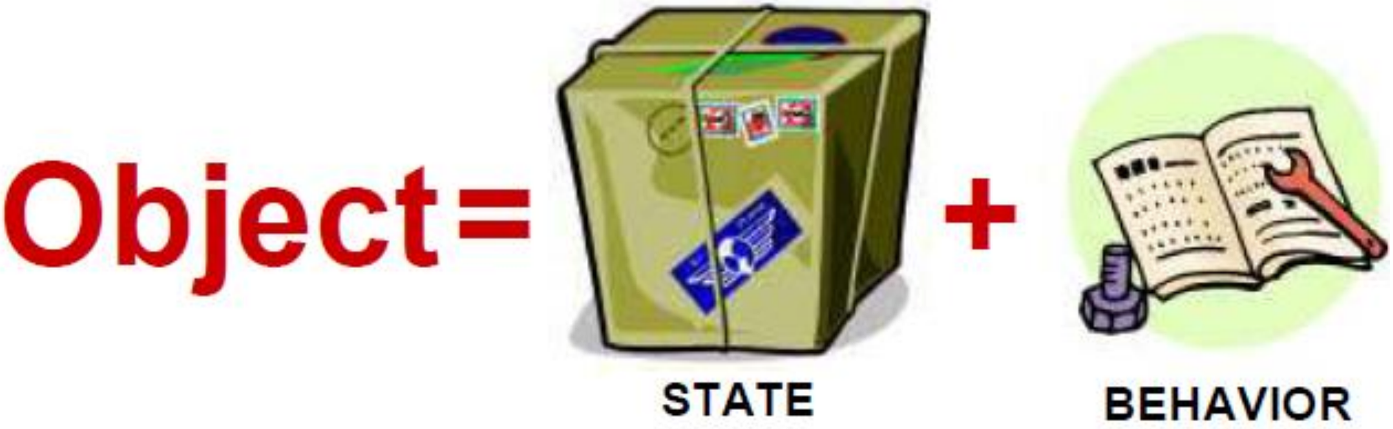
- **A utility class has features (fields and methods) that are all `static`.**
    - therefore, you do not need objects to use those features
        - a well implemented ***utility class*** should have **a single, empty private constructor** to prevent the creation of objects. (more detail later)

Non-static classes

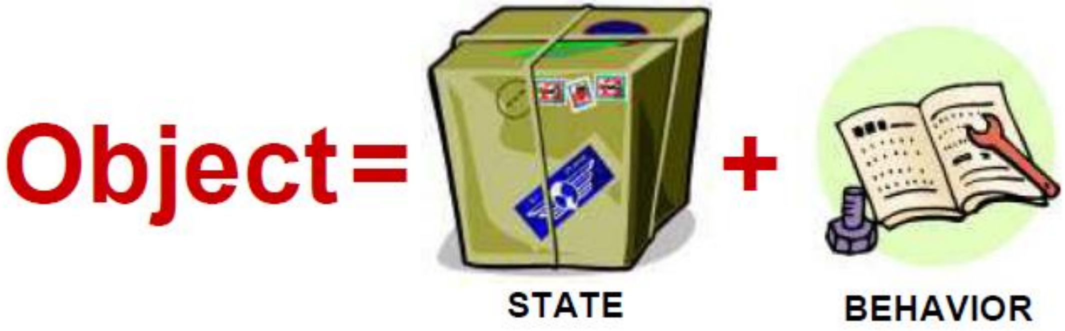
- **Most Java classes are *not* utility classes**
  - *they are intended to be used to create to objects*
  - each object has its ***own copy*** of all `non-static` fields
    - ***it is also useful to imagine that each object has its own copy of all non-static methods***

Why objects?

- Each object has its own copy of all ***non-static fields***
  - this allows objects to have their **own *state***
    - in Java the state of an object is the set of **current values of all of its non-static fields**



- Object-oriented programming in Java:
  - Use classes to define **templates**
  - Use objects to **instantiate** classes
  - At runtime, create objects and call methods on objects, to **simulate** interactions between real-life entities.



Implementing classes

- Many classes represent kinds of values
  - examples of values: name, date, colour, mathematical point or vector
  - Java examples: **`String`**, **`Date`**, **`Integer`**
- *When implementing a class you need to choose appropriate fields to represent the state of each object*

- Consider implementing a class that represents **2-dimensional points**
  - a possible implementation would have:
    - a field to represent the **x-coordinate** of the point
    - a field to represent the **y-coordinate** of the point

```java
/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every <code>SimplePoint2D</code> instance has a
 * public x and y coordinate that can be directly accessed
 * and modified.
 *
 * @author EECS2030 Winter 2016-17
 *
 */
public class SimplePoint2 {
    public float x;
    public float y;
}
```

public class: any client can use this class

public fields: any client can use these fields by name

Note: Client is any class with its main method using this class

Using `SimplePoint2`

- Even in its current form, we can use **`SimplePoint2`** to create and manipulate point objects

```java
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 + ")");
}
```

- Notice that printing a point is somewhat inconvenient
  - we have to manually compute a string representation of the point

Using SimplePoint2

- Initializing the coordinates of the point is somewhat inconvenient
  - we have to manually set the x and y coordinates
- We get unusual results when using equals

```java
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));
}
```

Encapsulation

- We can add ***features*** to **`SimplePoint2`** to make it easier to use
  - We can add **constructors** that *set the values of the fields* of a **`SimplePoint2`** object when it is created
  - We can add methods that *use the fields* of **`SimplePoint2`** to perform some sort of computation (***like compute a string representation of the point***)

- In object oriented programming the term ***encapsulation*** means **bundling data and methods that use the data into a single unit**
  - ***That involves enclosing an object with a kind of "protective bubble" so that it cannot be accessed or modified without proper permission.***

Constructors

- The purpose of a constructor is to initialize the state of an object
    - *it should set the values of all of the* ***non-static fields*** *to appropriate values*
- A constructor:
    - must have the same name as the class
    - never returns a value (not even void)
        - constructors are not methods
    - can have zero or more parameters

Implicit (Generated) Constructor

- Java allows to define a class **without any constructors** but it ***does not mean the class will not have any***.
- This class has no constructor but **Java compiler** will generate one implicitly and the creation of new class instances will be possible using **`new`** keyword.

```
 public class NoConstructor {


 }
public static void main(String[] args) {
  final NoConstructor noConstructorInstance = new NoConstructor();
}
```

Declaring Constructors

Syntax

```
[<modifier>]<class_name>( <argument>*){
      <statement>*
}
```

Example

```
public class Date
{
private int year, month, day;
public Date( int y, int m, int d) {
if( verify(y, m, d) ){
      year = y; month = m; day = d;
}
}
private boolean verify(int y, int m, int d){
//...
}}
```

Default constructor

- The default constructor has zero parameters
- The default constructor initializes the state of an object to some well defined state chosen by the implementer

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

  /**
   * The default constructor. Sets both the x and y coordinate
   * of the point to 0.0f.
   */
  public SimplePoint2() {
    this.x = 0.0f;
    this.y = 0.0f;
  }
```

Inside a constructor, the keyword **this** is a reference to the object that is currently being initialized.

- The **default constructor** takes no argument
- The **default constructor's** body is **empty**

```
public class Date {
  private int year, month, day;
  public Date( ){
  }
}
```

Constructors without Arguments

- The constructor without arguments (or ***no-arg constructor***) is the simplest constructors.
- This constructor will be called **once new instance of the class** is created using the **`new`** keyword.

```
 public class NoArgConstructor {
    public NoArgConstructor() {
       // Constructor body here
    }
 }
final NoArgConstructor noArgConstructor =
      new NoArgConstructor();
```

Constructors with Arguments

- The ***constructors with arguments*** are the **most** interesting and useful way to parameterize new class instances creation.
  - The following example defines a constructor with two arguments.
  - In this case, when class instance is being created using the **`new`** keyword, **both constructor arguments should be provided.**

```
public class ConstructorWithArguments {
    public ConstructorWithArguments(final String arg1,final
    String arg2) {
        // Constructor body here
    }

 final ConstructorWithArguments constructorWithArguments =
                new ConstructorWithArguments( "arg1", "arg2" );
```

Custom constructors

- A class can have multiple constructors but the **signatures** of the constructors must be unique
  - ***i.e., each constructor must have a unique list of parameter types***
- It would be convenient for clients if **`SimplePoint2`** had a constructor that let the client set the **`x`** and **`y`** coordinate of the point

```
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

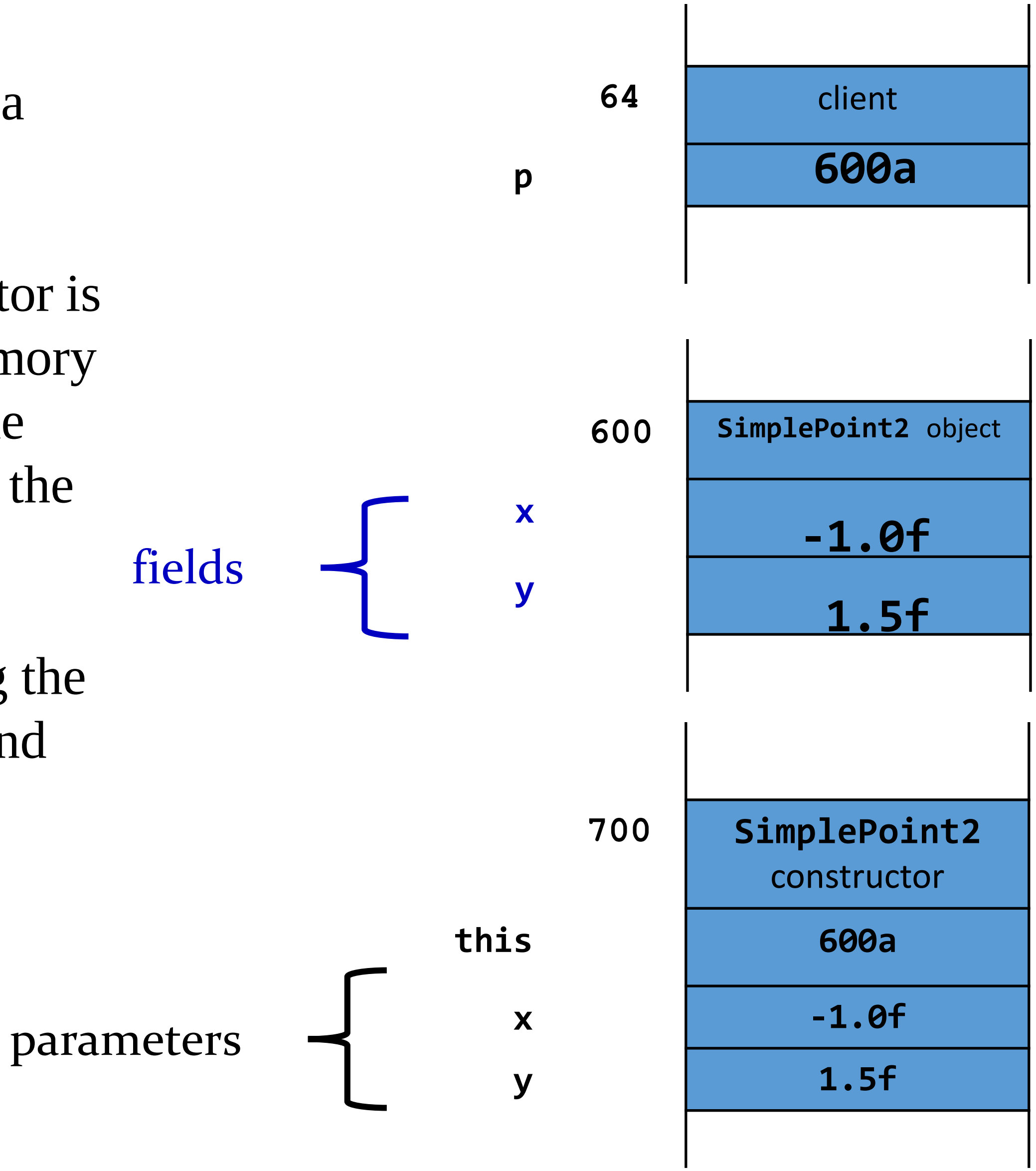
```
SimplePoint2 p = new SimplePoint2(-1.0f, 1.5f);
```

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



64
p
client
600a

600
SimplePoint2 object
fields
x
-1.0f
y
1.5f

700
SimplePoint2 constructor
this
600a
parameters
x
-1.0f
y
1.5f

`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***

References

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