

# **Unit 5: Writing Classes**

## **Variables, Scope and Semantics**

Adapted from:

- 1) Building Java Programs: A Back to Basics Approach  
by Stuart Reges and Marty Stepp
- 2) Runestone CSAwesome Curriculum

# Static Variables

```
private static type name;
```

or,

```
private static type name = value;
```

– Example:

```
private static int theAnswer = 42;
```

**static variable:** Stored in the class instead of each object.

- A "shared" global field that all objects can access and modify.
- Like a class constant, except that its value can be changed.

# Final Static fields

```
public static final type name;
```

or,

```
public static final type name = value;
```

– Example:

```
public static final int NUMOFMONTHS = 12;
```

## Final static variable:

- A class constant whose value cannot be changed. Usually public.
- ALL CAPS by convention.

# Instance Variables

```
private type name;
```

or,

```
private type name = value;
```

– Example:

```
private int id = 243342;
```

**instance variable:** Stored in an object instead of the class.

– each object has its own copy of the instance variable.

# BankAccount

```
public class BankAccount {  
    // static count of how many accounts are created  
    // (only one count shared for the whole class)  
    private static int objectCount = 0;  
  
    // instance variables (replicated for each object)  
    private String name;  
    private int id;  
  
    public BankAccount(String n) {  
        name = n;  
        objectCount++;    // advance the id, and  
        id = objectCount; // give number to account  
    }  
    // clients can call this to find out # accounts created  
    public static int getNumAccounts() {  
        return objectCount;  
    }  
    ...  
    public int getID() {    // return this account's id  
        return id;  
    }  
}
```

# Static vs Instance Call

A static method is called through the name of the class.

An instance method is called through the name of an object.

```
public class Main {  
    public static void main(String[] args) {  
        BankAccount a = new BankAccount("Jim Smith");  
  
        //getID is instance  
        // uses object name + dot notation to call  
        System.out.println(a.getID());  
  
        //getNumAccounts is static  
        // uses class name + dot notation to call  
        System.out.println(BankAccount.getNumAccounts());  
    }  
}
```

# Error: Static Access

```
public class BankAccount {  
  
    private static int objectCount = 0;  
  
    private String name;  
    private int id;  
  
    public BankAccount(String n) {  
        name = n;  
        objectCount++;  
        id = objectCount;  
    }  
  
    public static int getNumAccounts() {  
        System.out.println(name);  
        System.out.println(this.id);  
        return objectCount;  
    }  
  
    ...  
}
```

**Error! static method does not have access to any particular object's variables!**

**(No implicit this parameter)**

# Scope

- **scope:** The part of a program where a variable exists.
  - From its declaration to the end of the { } braces
    - A variable declared in a `for` loop exists only in that loop.
    - A variable declared in a method exists only in that method.

```
public static void example() {  
    int x = 3;  
    for (int i = 1; i <= 10; i++) {  
        System.out.println(x);  
    }  
    // i no longer exists here  
} // x ceases to exist here
```

i's scope

x's scope



# Scope implications

- Variables without overlapping scope can have same name.

```
for (int i = 1; i <= 100; i++) {  
    System.out.print("A");  
}  
for (int i = 1; i <= 100; i++) {    // OK  
    System.out.print("BB");  
}  
int i = 5;                        // OK: outside of loop's scope
```

- A variable can't be declared twice or used out of its scope.

```
for (int i = 1; i <= 100 * line; i++) {  
    int i = 2;                        // ERROR: overlapping scope  
    System.out.print("/");  
}  
i = 4;                              // ERROR: outside scope
```

# Example

```
if( x <= 3)
{
    int y = 2;
    ...
}
```

```
y = 5; // error since y does not exist outside
       // the if block
```

# Example

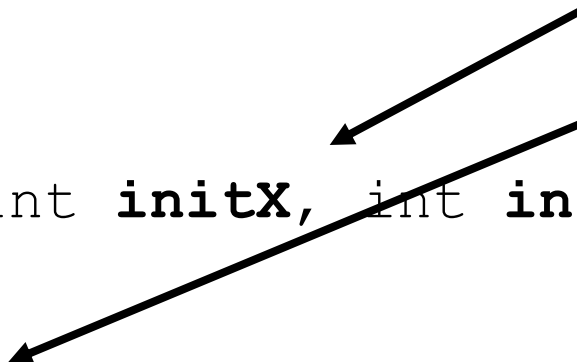
```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initX, int initY) {  
        x = initX;  
        y = initY;  
    }  
  
    public void setX() {  
        return x;  
    }  
}
```

The scope of the instance variables **x** and **y** is the entire class.

# Example

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initX, int initY) {  
        x = initX;  
        y = initY;  
    }  
  
    public void setX() {  
        // initX and initY don't exist here.  
        return x;  
    }  
}
```

The scope of the parameter **initX** and **initY** is **ONLY** the constructor.



# The `this` keyword

**this** : Within a non-static method or a constructor, the keyword **this** is a reference to the current object—the object whose method or constructor is being called.

- Refer to a field: `this.field`
- Call a method: `this.method (parameters) ;`
- One constructor can call another: `this (parameters) ;`


Note: "this" can be omitted if it is clear which variable is being referenced. The keyword "this" is helpful to fix the shadowing problem. We discuss this next.

# Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

```
public class Point {  
    private int x;  
    private int y;
```

The instance  
variables **x** and **y**  
are global  
variables.



```
    ...
```

```
    public Point(int x, int y) {  
  
    }
```

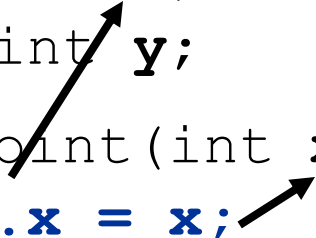
The parameter  
variables **x** and **y**  
are local variables.



- In most of the class, **x** and **y** refer to the instance variables.
- In the constructor, **x** and **y** refer to the method's parameters.

# Fixing shadowing

```
public class Point {  
    private int x;  
    private int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

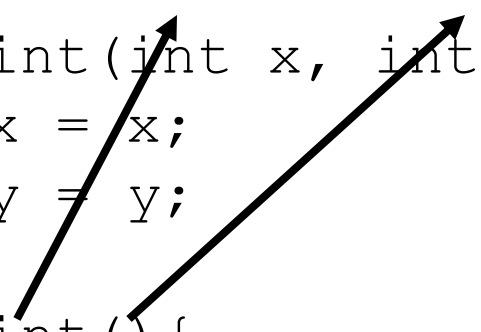


- Inside the constructor(or any method with shadowing):
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

# Constructors

One constructor can call another using the **this** keyword. This helps us reuse another constructor's code.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Point() {  
        this(0, 0);  
    }  
}
```





# Point Revisited


```
public class Point {  
    private int x;  
    private int y;  
    public Point(){  
        this(0, 0);  
    }  
    public Point(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
    public double distanceToPoint(Point other){  
        int dx = this.x - other.x;  
        int dy = this.y - other.y;  
        return Math.sqrt(dx * dx + dy * dy);  
    }  
    public double distanceToOrigin(){  
        Point origin = new Point();  
        return this.distanceToPoint(origin);  
    }  
}
```

# MyComplex

We wrote the MyComplex class in the last lecture's assignment.

```
public class MyComplex {  
    private double real;  
    private double img;  
    ...  
    // Add complex number other to this object.  
    public void add(MyComplex other) {  
        this.real += other.real;  
        this.img += other.img;  
    }  
    ...  
}
```

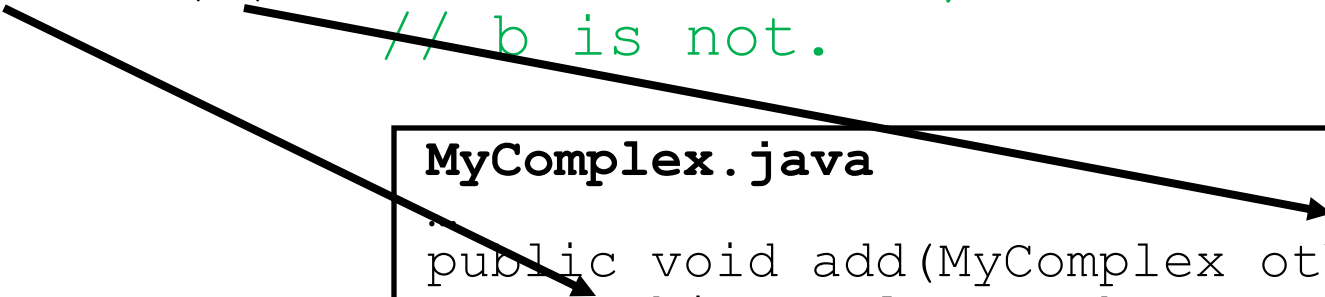
**Note that the use of  
this here is optional.**



# MyComplex

**this** is a reference to the current object—the object whose method is being called.

```
public class Main {  
    public static void main(String[] args) {  
        MyComplex a = new MyComplex(2, 5);  
        MyComplex b = new MyComplex(-1, 3);  
        a.add(b); // add b onto a, a is changed  
                  // b is not.  
    }  
}
```



**MyComplex.java**

```
public void add(MyComplex other) {  
    this.real += other.real;  
    this.img += other.img;  
}
```

# MyComplex

In the MyComplex Lab, we distinguish between the instance method and the static method `addNew`.

```
public class MyComplex {
    private double real;
    private double img;
    ...
    public void add(MyComplex other) {
        this.real = this.real + other.real;
        this.img = this.real + other.img;
    }
    public static MyComplex addNew(MyComplex a,
                                   MyComplex b) {
        double real = a.real + b.real;
        double img = a.img + b.img;
        return new MyComplex(real, img);
    }
}
```

# MyComplex

Let's use the "this" keyword to rewrite add using addNew.

```
public class MyComplex {  
    private double real;  
    private double img;  
    ...  
    public void add(MyComplex other) {  
        MyComplex temp = addNew(this, other);  
        real = temp.real;  
        img = temp.img;  
    }  
    public static MyComplex addNew(MyComplex a,  
                                   MyComplex b) {  
        double real = a.real + b.real;  
        double img = a.img + b.img;  
        return new MyComplex(real, img);  
    }  
}
```

# Swapping values

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
  
    // swap a with b?  
    a = b;  
    b = a;  
  
    System.out.println(a + " " + b);  
}
```

– What is wrong with this code? What is its output?

- The red code should be replaced with:

```
int temp = a;  
a = b;  
b = temp;
```

# A swap method?

- Does the following swap method work? Why or why not?

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
  
    // swap a with b?  
    swap(a, b);  
  
    System.out.println(a + " " + b);  
    // 7 35 (unchanged)  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

# Value semantics

- **value semantics:** Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;  
int y = x;      // x = 5, y = 5  
y = 17;          // x = 5, y = 17  
x = 8;           // x = 8, y = 17
```



# Reference semantics (objects)

- **reference semantics:** Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*(aliases).
  - Modifying the value of one variable *will* affect others.

```
Sprite a = new Sprite(10.0, 20.0);  
Sprite b = a; // refers to the same Sprite object as a  
b.center_x = 50.0;  
System.out.println(a.center_x); // 50.0
```

# Objects as parameters

Custom objects(except String) use reference semantics. Why?

- *efficiency*. Copying large objects slows down a program.
- *sharing*. It's useful to share an object's data among methods.

When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.

- If the parameter is modified, it *will* affect the original object.

# Value Semantics

The primitive types `int`, `double`, `boolean` all use value semantics.

Example:

```
public static void triple(int number) {  
    number = number * 3;  
}  
  
public static void main(String[] args) {  
  
    int x = 2;  
    triple(x);  
    System.out.println(x); // x is unchanged!  
  
}
```

# Value Semantics

**String uses value semantics like primitive types. It's the only object class that uses value semantics.**

Example:

```
public static void repeat(String str) {  
    str = str + str;  
}  
  
public static void main(String[] args) {  
  
    String str = "hi";  
    repeat(str);  
    System.out.println(str); // "hi"  
}
```

# Reference Semantics

In the example below, a and b both reference the same object. They are aliases. Modifying one will modify the other.

Example:

```
public static void moveRight(Sprite b) {  
    b.center_x += 5.0;  
}  
  
public static void main(String[] args) {  
  
    Sprite a = new Sprite(100.0, 200.0);  
    moveRight(a);  
    System.out.println(a.center_x);  
    // 105.0  
}
```

# Summary of Java classes

- A class is used for any of the following in a large program:
  - a *program* : Has a main and perhaps other static methods.
    - example: `GuessingGame`, `Birthday`, `MadLibs`,
    - does not usually declare any static fields (except `final`)
  - an *object class* : Defines a new type of objects.
    - example: `Point`, `BankAccount`, `Date`, `Car`, `TetrisPiece`
    - declares object fields, constructor(s), and methods
    - might declare static fields or methods, but these are less of a focus
    - should be encapsulated (all fields and static fields `private`)
  - a *module* : Utility code implemented as static methods.
    - example: `Math`

# Lab 1

- Create a class called BankAccount(BankAccount.java) below that keeps track of the account holder's name(public), the account number(private), and the balance(private) in the account. Create a static variable that keep tracks of the number of BankAccount objects. Make sure you use the appropriate data types for these.
- Write 2 constructors for the class that initialize the instance variables to default values and to given parameters. For the parameters, use the same variable names as your instance variables. Use the **this** keyword to distinguish between the instance variables and the parameter variables.

# Lab 1

- Write a `toString()` method for the class. Use the **this** keyword to return the instance variables. This method should return the string containing the name and balance of the account.
- Write a `withdraw(amount)` and `deposit(amount)` for the class. Withdraw should subtract the amount from the balance as long as there is enough money in the account (the balance is larger than the amount). Deposit should add the amount to the balance. Use the **this** keyword to refer to the balance.
- Write the method that returns the number of BankAccount objects.
- Test your class below with a main method (in `Main.java`) that creates a Bank Account object and calls its deposit and withdraw methods and prints out the object to test its `toString()` method.



# References

1) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum:

<https://runestone.academy/runestone/books/published/csawesome/index.html>

For more tutorials/lecture notes in Java, Python, game programming, artificial intelligence with neural networks:

<https://longbaonguyen.github.io>