JAVA

Lecture IV – Object Oriented Programming

EXAMPLE: VEHICLE

```
public class  Vehicle{
        int passengers;  // maximum number of passengers
        int fuelCap;     // fuel capacity in gallons
        int mpg;         // consumptions in miles per gallon
}
```

EXAMPLE: VEHICLE

```
public static void main(String[] args){
        Vehicle car = new Vehicle(); // create new objects
        Vehicle van = new Vehicle();
        car.mpg = 12;
        car.fuleCap = 14;
        van.mpg = 21;
        van.fuleCap = 16;
        car.mpg = van.mpg;   // what will happen?
        car.mpg = 10;        // what is the mpg value of van?
        van = car;           // what is the mpg value of car?
        van.mpg = 20;        // what is the mpg value of car?
}
```

EXAMPLE: VEHICLE

- We want to know the range of a vehicle:

```
public static void main(String[] args){
	Vehicle car = new Vehicle(); // create new objects
	Vehicle van = new Vehicle();
	car.mpg = 12;
	car.fuleCap = 14;
	van.mpg = 21;
	van.fuleCap = 16;
	int carRange = car.mpg * car.fuleCap;
	int vanRange = van.mpg * van.fuleCap;
}
```

- What is the problem here?

METHODS

- The user doesn't need to know how the range is calculated.
- Objects of the same class have similar ways to manipulate on its data.
- General Form

```
returnType methodName(parameters){
        statement;
        ……
}
```

- Parameters are local variables provided by the caller. Its scope is within the method.
- The return type can be any valid type of void.

EXAMPLE METHOD

- The following method can be added to the Vehicle class:

```
void range(){
        System.out.println("range: " + (fuleCap * mpg));
}
```

- If this method is called in the main method as follows:

```
car.range();
```

- "range: 168" will be displayed
- Variables and Methods are accessed/called through the **object** not the class

RETURNING A VALUE

- return *value*;

```
int range(){
        return fuleCap * mpg;
}
```

- If the main method

```
int range = car.range();
System.out.println("range: " + range);
```

- Return type must be consistent with the definition.
- Return is the end of a method. Two ways to end a void method:
  - 1) reach the end brace }
  - 2) return;

PARAMETERS

- What if we want to know the amount of fuel needed to travel a certain distance?
- Parameters: local variable are provided by the caller whose scope is the method body.

```
double fuleNeeded(int distance){
        return (double)distance / mpg;
}
```

- In the main method:

```
System.out.println( car.fuleNeeded(120));
// print 10
```

CONSTRUCTORS

- Initial variable values are set in the main method.
  - Difficult to manage
  - Error Prone
- Constructor is used to initialize an object when it is created.

```
Vehicle(){
        fuleCap = 14;
        mpg = 12;
        passages = 4;
}
```

- The name of a constructor must be consistent with its class name

```
Vehicle car = new Vehicle();
```

CONSTRUCTORS

- Problem: different objects may have different initial values.
- Constructor with parameters

```
Vehicle(int f, int m, in p){
        fuleCap = f;
        mpg = m;
        passages = p;
}
```

- In the main method:

```
Vehicle car = new Vehicle(14, 12, 4);
```

- Variables can be initialized in the constructor or in the variable declaration, what is the order?

EXAMPLE

```
class ThisTest{
	int a, b;
	public void setData(int a, int b){
		a = a;
		b = b;
	}
	public void showData(){
		System.out.println("a=" + a);
		System.out.println("b=" + b);
	}
	public static void main(String[] args){
		ThisTest tt = new ThisTest();
		tt.setData(10, 20);
		tt.showData();
	}
}
```

- What is the output?

KEYWORD *THIS*

- Problem: the value of a local variable is assigned to an instance variable of the same name.
- ***this***: an implicit argument that refers to the object on which the method is called.
- You don't have to create an object. this help you to clarify instance variables and local variables.

```
public void setData(int a, int b){
      this.a = a;
      this.b = b;
}
```

EXAMPLE

```
class ThisTest{
	int a, b;
	public void setData(int a, int b){
		this.a = a;
		this.b = b;
	}
	public void showData(){
		System.out.println("a=" + a);
		System.out.println("b=" + b);
	}
	public static void main(String[] args){
		ThisTest tt = new ThisTest();
		tt.setData(10, 20);
		tt.showData();
	}
}
```

- What is the output?

EXAMPLE

```
class ThisTest{
	int a, b;
	public static void main(String[] args){
		a = 10; //what will happen?
	}
}
```

EXAMPLE

```
class ThisTest{
      int a, b;
      public static void main(String[] args){
            a = 10; //what will happen?
      }
}
```

Syntax error!

Instance variables cannot be accessed by a static method!

EXAMPLE

```
class ThisTest{
	int a, b;
	public static void main(String[] args){
		ThisTest tt = new ThisTest();
		tt.a = 10;
	}
}
```

Instance variables could be only called through an object!

STATIC KEYWORD

- Typically, variables and methods are accessed through an object of its class.

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

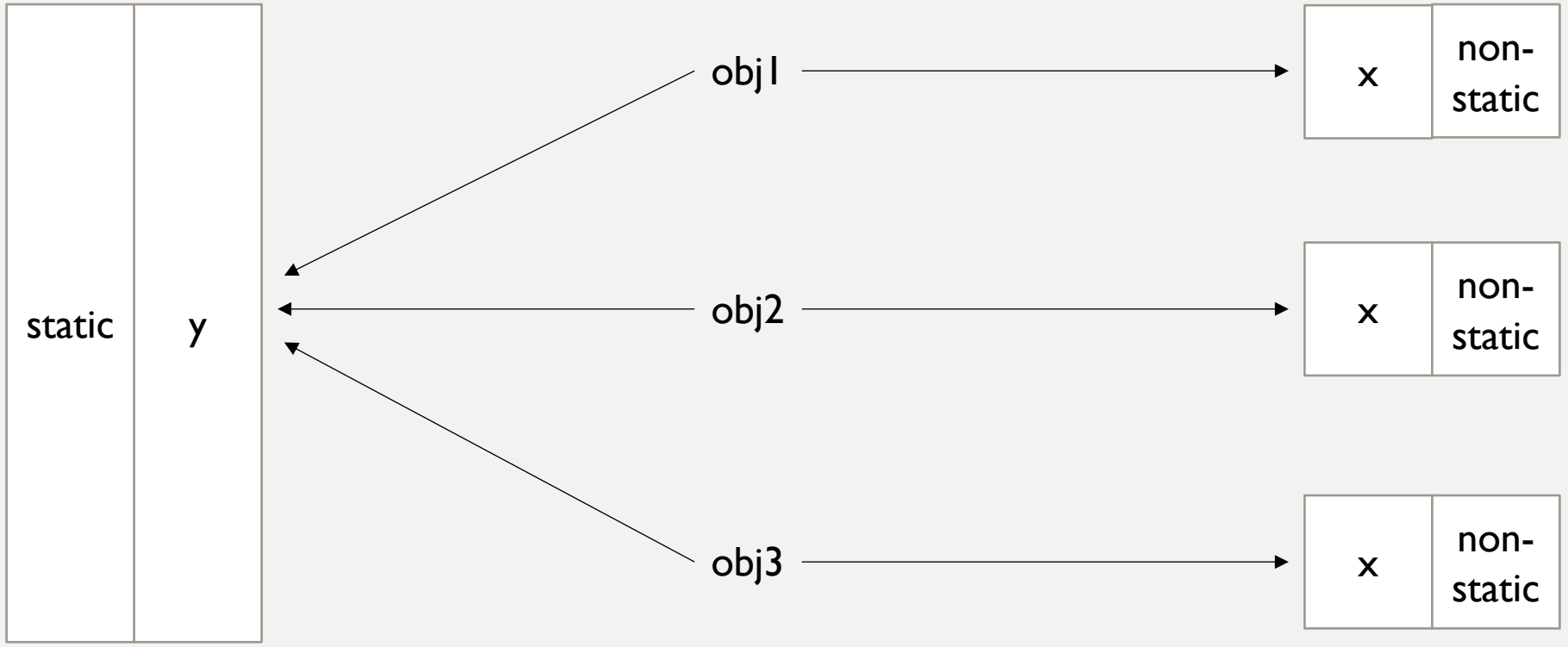
- Static: to define a class member that will be used independently of any objects of that class.
- Static variables: global variables for all objects of a particular class, i.e., all instance of the class share the same static variable.
- Static variables are independent of any specific object.
- General form:

```
ClassName.variable_name
```

EXAMPLE: STATIC VARIABLE

```
public class StaticDemo{
	int x;
	static int y;
	int sum(){
		return x + y;
	}
}
public class SDemo{
	public static void main(String[] args){
		StaticDemo obj1 = new StaticDemo();
		StaticDemo obj2 = new StaticDemo();
		obj1.x = 10;
		obj2.x = 20;
		StaticDemo.y = 19;
		System.out.println("sum obj1: " + obj1.sum());
		System.out.println("sum obj1: " + obj2.sum());
	}
}
```

STATIC KEYWORD



EXAMPLE: COUNTER

```
public class MyClass{
	static int count;
	MyClass(){
		count++; // the value of count increase
	}
}
public class CounterDemo{
	public static void main(String[] args){
		for(int i = 0; i < 3; i++){
			MyClass obj = new MyClass();
			System.out.println(“Class ” + MyClass.count;)
		}
	}
}
```

STATIC METHODS

- Similarly, a static method are independently of any objects, and thus can be called through its class name.

```
ClassName.MethodName();
```

- Static method is useful for creating utility methods that performs useful functions not related to a specific object.
- For example, the Math class in java… Math.sqrt(), Math.cos(), …
- Restrictions:
  - Only the static methods can be called directly
  - Can only directly access to the static data
  - Do not have **this** reference

EXAMPLE: STATIC METHODS

```
public class StaticError{
	int x = 3;
	static int y = 1024;
	public static void main(String[] args){
		int z = y / x;
	}
}
```

EXAMPLE: STATIC METHODS

```
public class StaticError{
	int x = 3;
	static int y = 1024;
	public static void main(String[] args){
		StaticError se = new StaticError();
		int z = y / se.x;
	}
}
```

RECAP-ENCAPSULATION

- Encapsulation:
  - Links data with the code that manipulates it.
  - Control the access of particular data.
- Vehicle Example:

```
public class Vehicle{
      int passengers;
      int fuelCap;
      int mpg;
      ……
}
```

We want to prevent misuse of the data, e.g., the code outside are not allowed to set these values.

ACCESS MODIFIERS

- Access modifiers: Controls the member access in Java
- 4 types of access modifiers: public, private, default, protected.
- We focus on public and private in this lecture:
- public: can be accessed by any other code in your program, e.g., other classes, or any method in other classes.
- private: can only be accessed by other members of its own class.
- Encapsulation: use private access modifier to protect data.

EXAMPLE: VEHICLE

```
public class Vehicle{
	private int passengers, fuelCap, mpg;
	public Vehicle(int p, int f, int m){
		this.passengers = p;
		this.fuelCap = f;
		this.mpg = m;
	}
}

Public class VehicleDemo{
	public static void main(String[] args){
		Vehicle car = new Vehicle(4, 14, 12);
		car.mpg = 20;
		System.out.println(car.mpg);
	}
}
```

EXAMPLE: VEHICLE

```
public class Vehicle{
	private int passengers, fuelCap, mpg;
	public Vehicle(int p, int f, int m){
		this.passengers = p;
		this.fuelCap = f;
		this.mpg = m;
	}
}

Public class VehicleDemo{
	public static void main(String[] args){
		Vehicle car = new Vehicle(4, 14, 12);
		car.mpg = 20; // fails as mpg is private
		System.out.println(car.mpg);
	}
}
```

ENCAPSULATION

The code outside can only interact with public method.

```
public class Vehicle{
	private int passengers, fuelCap, mpg;
	public Vehicle(int p, int f, int m){
		this.passengers = p;
		this.fuelCap = f;
		this.mpg = m;
	}
	public int getMpg(){
		return mpg;
	}
	private int range(){
		return mpg * fuleCap;
	}
}
```

ANOTHER EXAMPLE

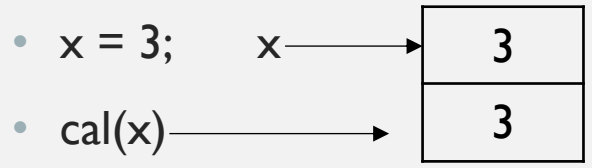
- Is it sufficient enough to use **private** to protect the data?

```
public class A{
        public String name;
        public A(String n){
                name = n;
        }
        public void printA(){
          System.out.println(name);
        }
}
public class B{
        private A a;
        public B(A a){
                this.a = a;
        }
        public A getA(){
                return this.a;
        }
}
```

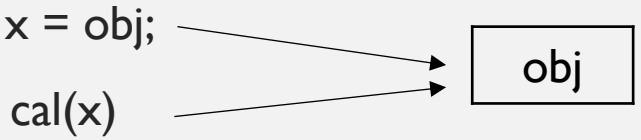
```
public class Test{
    public static void main(String[] args){
        A a1 = new A("A1");
        B b1 = new B(a1);

        A a2 = b1.getA()
        a2.printA();
        a2.name = "A2";
        b1.getA().printA();
    }
}
```

WHY

- Two ways of passing an argument to a subroutine:
  - *Call-by-value:* the value of an argument is copied into the subroutine (primitive types in Java)
  - *Call-by-reference*: a reference to an argument is passed to the subroutine (objects in Java)



call-by-value



call-by-reference

EXAMPLE: PRIMITIVE

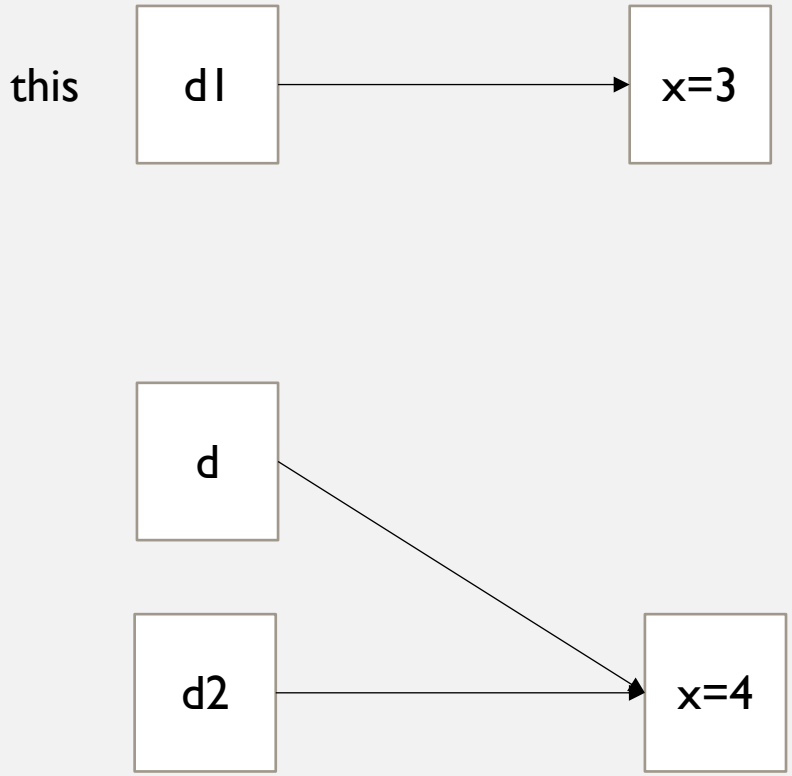
```
class TestPass{
	static void make3(int x){
		System.out.println(“(2)” + x);
		x = 3;
		System.out.println(“(3)” + x);
	}

	public static void main(String[] args){
		int x = 4;
		System.out.println(“(1)” + x);
		make3(x);
		System.out.println(“(4)” + x);
	}
}
```

EXAMPLE: OBJECTS

```
class Data{
	int x;
	void addTo(Data d){
		d.x = d.x + this.x;
		System.out.println("(1)" + d.x);
	}
}

class TestRef{
	public static void main(String[] args){
		Data d1 = new Data();
		Data d2 = new Data();
		d1.x = 3; d2.x = 4;
		d1.addTo(d2);
		System.out.println("(2)" + d2.x);
	}
}
```

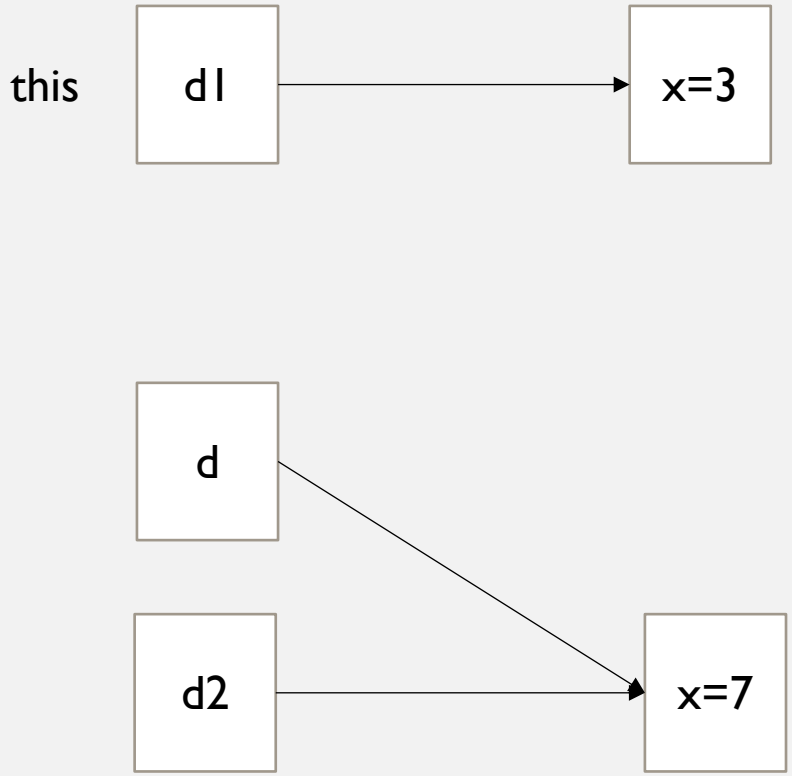
EXAMPLE: OBJECTS

```
class Data{
	int x;
	void addTo(Data d){
		d.x = d.x + this.x;
		System.out.println("(1)" + d.x);
	}
}

class TestRef{
	public static void main(String[] args){
		Data d1 = new Data();
		Data d2 = new Data();
		d1.x = 3; d2.x = 4;
		d1.addTo(d2);
		System.out.println("(2)" + d2.x);
	}
}
```



EXAMPLE: OBJECTS

```
class Data{
	int x;
	void addTo(Data d){
		d.x = d.x + this.x;
		System.out.println("(1)" + d.x);
	}
}

class TestRef{
	public static void main(String[] args){
		Data d1 = new Data();
		Data d2 = new Data();
		d1.x = 3; d2.x = 4;
		d1.addTo(d2);
		System.out.println("(2)" + d2.x);
	}
}
```

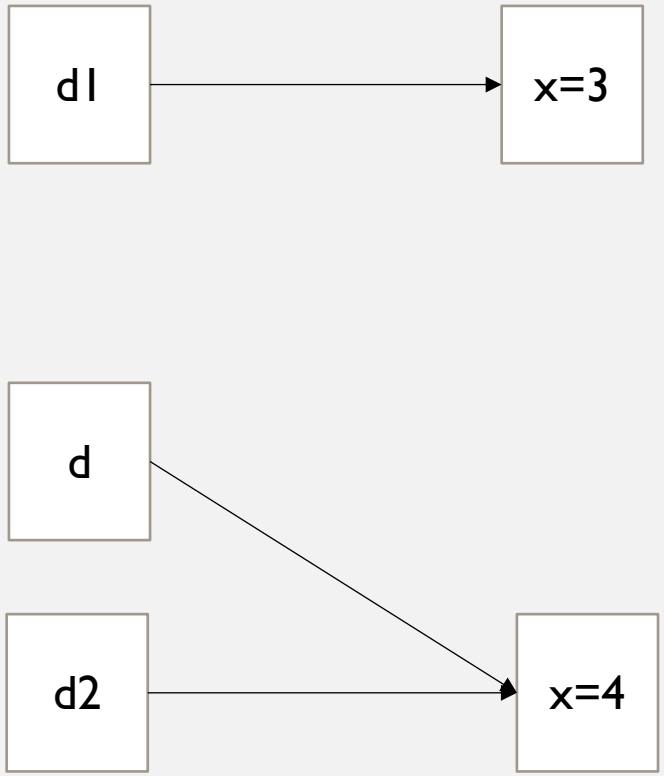


QUESTIONS

```
class Data{
        int x;
        void addTo(Data d){
                d = new Data();
                d.x = 10;
                System.out.println("(1)" + d.x);
        }
}

class TestRef{
        public static void main(String[] args){
                Data d1 = new Data(); Data d2 = new Data();
                d1.x = 3; d2.x = 4;
                d1.addTo(d2);
                System.out.println("(2)" + d2.x);
        }
}
```

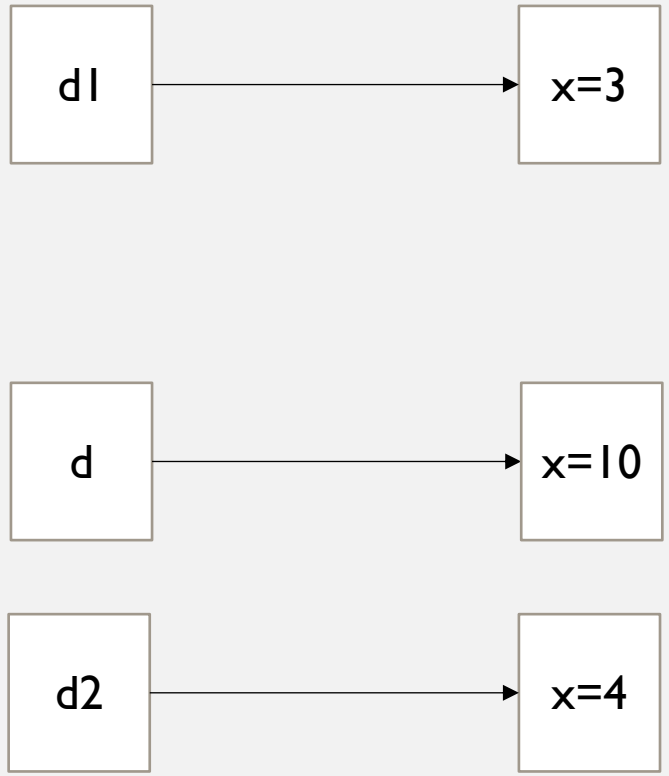
QUESTIONS

```
class Data{
        int x;
        void addTo(Data d){
                d = new Data();
                d.x = 10;
                System.out.println("(1)" + d.x);
        }
}

class TestRef{
        public static void main(String[] args){
                Data d1 = new Data(); Data d2 = new Data();
                d1.x = 3; d2.x = 4;
                d1.addTo(d2);
                System.out.println("(2)" + d2.x);
        }
}
```



QUESTIONS

```
class Data{
        int x;
        void addTo(Data d){
                d = new Data();
                d.x = 10;
                System.out.println("(1)" + d.x);
        }
}

class TestRef{
        public static void main(String[] args){
                Data d1 = new Data(); Data d2 = new Data();
                d1.x = 3; d2.x = 4;
                d1.addTo(d2);
                System.out.println("(2)" + d2.x);
        }
}
```



INNER CLASSES

- Nested classes:
  - Non-static nested classes: Inner classes
  - Static nested classes
- Group classes together to make code more readable and maintainable
- Scope: bounded by its outer class, all variables and methods of its outer class.
- General Form:

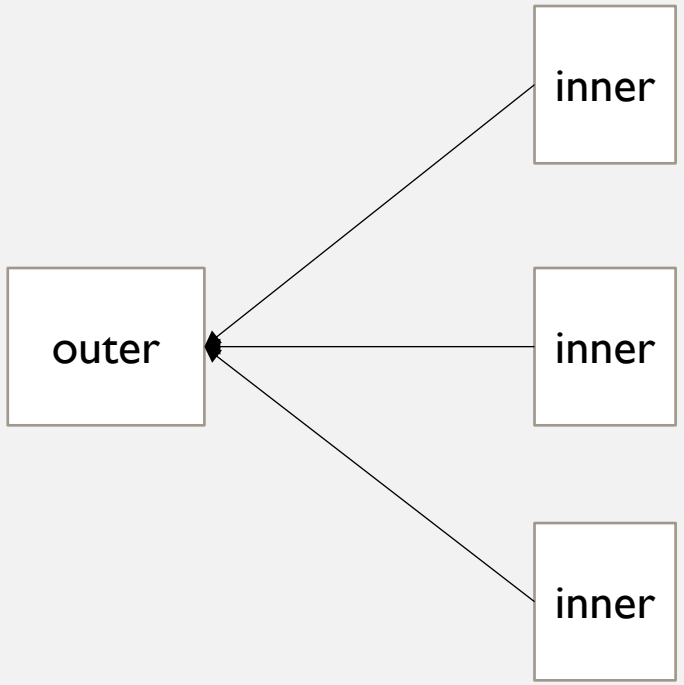
```
class Outer{
      class Inner{
      }
}
```

INNER CLASS

```
class Outer{
	int x = 10;
	class Inner{
		int y = 5;
		void printX(){
			System.out.println(x);
		}
	}

	int addition(){
		Inner inner = new Inner();
		return x + inner.y;
	}
}
```

INNER CLASS

```
class Outer{
	int x = 10;
	class Inner{
		int y = 5;
		void printX(){
			System.out.println(x);
		}
	}

	int addition(){
		Inner inner = new Inner();
		return x + inner.y;
	}
}
```



INNER CLASS

```
class Outer{
	int x = 10;
	class Inner{
		int y = 5;
	}
}

class Test{
	public static void main(String[] args){
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		System.out.println(”(”outer.x + “,” + inner.y +”)”);
	}
}
```

INNER CLASS

```
class Outer{
	int x = 10;
	private class Inner{
		int y = 5;
	}
}

class Test{
	public static void main(String[] args){
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		System.out.println("("outer.x + "," + inner.y +")");
	}
}
```

METHOD-LOCAL INNER CLASSES

- In Java, an inner class can be defined within a method.
- Its scope is within the method, like local variables

```
class LocalInner{
      void aMethod{
            class Inner{
            }
            Inner inner = new Inner();
      }
}
```

METHOD OVERLOAD

- Polymorphism: Redefine the way a class works by changing how it works or by changing the data.
  - Overload.
  - Override.
- Overload: two or more methods within the same class can share the same name, but with different parameter declarations.
- Restrictions:
  - The type and/or number of the parameters must differ.
  - The return types can be different, but not necessarily.

EXAMPLE: OVERLOAD

```
class Overload{
    void ovlDemo(){
       System.out.println("No parameters");
    }

    void ovlDemo(int a){
       System.out.println("One parameter:" + a);
    }

    int ovlDemo(int a, int b){
       System.out.println("Two parameters: " + a + " " + b);
       return a + b;
    }
}
```

OVERLOAD CONSTRUCTOR

- Constructors can also be overloaded.

```
class MyClass{
	int x;
	MyClass(){
		System.out.println(x);
	}
	MyClass(int i){
		x = i;
		System.out.println(x);
	}
	MyClass(int i, int j){
		x = i + j;
		System.out.println(x);
	}
}
```

OVERLOAD CONSTRUCTORS

- Overloading constructors allows to use one object to initialise another

```
class Summation{
	int sum;
	Summation(int num){
		sum = 0;
		for(int i = 1; i <= num; i++){
			sum += i;
		}
	}
	Summation(Summation ob){
		sum = ob.sum;
	}
}
```

RECURSION

- A method in Java can call itself.
- Recursion: the process of defining something in terms of itself.
  - Base case: a termination scenario that does not use recursion to produce an answer.
  - Recursive steps: reduce all successive cases towards the base case.

```
Void drawStars(int n){
        if(n == 1) // base case
                System.out.println("*");
        else{
                System.out.print("*");
                drawStars(n-1); //recursive step
        }
}
```

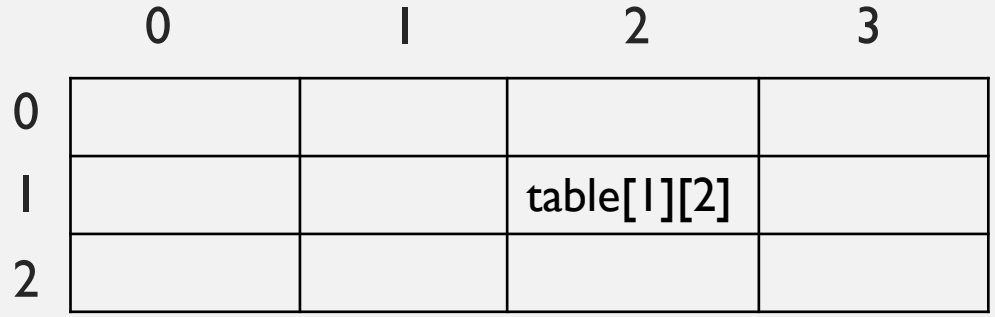
RECURSION VS ITERATION

```
void drawStars(int n){
      for(int i = 0; i < n; i++){
            System.out.print("*");
      }
      System.out.println();
}
```

- Each time you call a method, some overhead is added to your program.
- Typically, recursion is less efficient compared to iteration, but much clearer.
- You will learn more about efficiency in later courses, e.g., Algorithms…

MULTIDIMENSIONAL ARRAYS

- Multi-dimensional Arrays: an array of arrays.
- Two-dimensional array: An array of one-dimensional array.




```
int[][] table = new int[3][4];
for(int i = 0; i < 3; i++){
      for(int j = 0; j < 4; j++){
            table[i][j] = (j*4) + i + 1;
}
```

MULTIDIMENSIONAL ARRAYS

- Multi-dimensional Arrays: an array of arrays.
- Two-dimensional array: An array of one-dimensional array.

	0	1	2	3
0				
1			table[1][2]	
2				

```
int[][] table = new int[3][4];
for(int i = 0; i < 3; i++){
      for(int j = 0; j < 4; j++){
            table[i][j] = (j*4) + i + 1
}
```

1	2	3	4
5	6	7	8
9	10	11	12

IRREGULAR ARRAYS

- Similarly, we could assign the value of elements in two-dimensional arrays as follows:

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- What happens if …

```
int[][] table = {{1,2,3},{5,6,7,8},{9}};
```

IRREGULAR ARRAYS

- Similarly, we could assign the value of elements in two-dimensional arrays as follows:

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- What happens if …

```
int[][] table = {{1,2,3},{5,6,7,8},{9}};
```

1	2	3	
5	6	7	8
9			

- Java allows irregular arrays. You only need to specify the memory for the first dimension.
- `int[][] table = new int[3][];`

IRREGULAR ARRAYS

- What is the length of a 2D array?

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
int l = table.length;
```

IRREGULAR ARRAYS

- What is the length of a 2D array?

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
int l = table.length;
```

- 3.

- What does `table[0].length` means?

IRREGULAR ARRAYS

- What is the length of a 2D array?

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
int l = table.length;
```

- 3.

- What does `table[0].length` means?
- The number of element in `{1,2,3,4}`

ARRAYLIST

- What is the length of a 2D array?

```
int[][] table = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
int l = table.length;
```

- 3.

- What does `table[0].length` means?
- The number of element in `{1,2,3,4}`

ARRAYLIST

- What is the problem of array?
- Cannot change its size
- We are asked to pick up all the numbers that is greater than 10, but we don't know how many integers are there.
- Use java. util.ArrayList, can be seen as dynamic array
- ArrayList provides a wide range of useful methods to manipulate a collection of elements.

```
ArrayList<Type> name = new ArrayList<>();
```

ARRAYLIST

- Useful ArrayList methods:

```
boolean add(E e); // add an element to the tail
void add(int index, E e); // add an element at a specified position
void clear();// remove all elements
boolean contains(Object o); // check if it contains a specified element
E get(index i);// get an element at a specified position
E remove(index i);// remove an element at a specified position
boolean remove(Object o);// remove the first occurrence of an object
int size();// return the size of the arraylist
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--

EXAMPLE:ARRAYLIST

```
class ArrayListDemo{
      public static void main(String[] args){
            ArrayList<Character> a1 = new ArrayList<>();
            a1.add('A');
            a1.add('B');
            a1.add(1, 'C');
            a1.remove(2);
            …
      }
}
```