

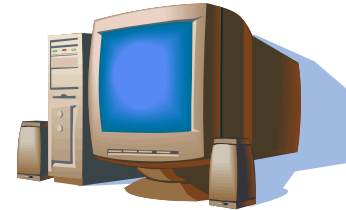


Object Oriented Programming with Java

03 - Introduction to Classes and Objects

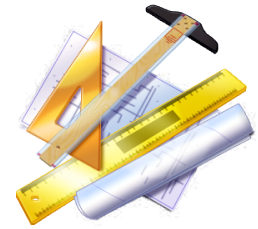
Object Oriented Programming

- Our lives are focussed around objects



- We use these objects in an abstract way for designing software
- So, objects are abstracted to computer code for reusability
- These abstract blueprints are *classes*

What is a class?



- Car analogy

- Engineers create the blueprints in order to manufacture a brand new car model
 - All the details like how the engine functions or how the gas emission is managed are in the blueprints
 - The car design will have details about certain behaviors (move, accelerate,..) and the materials being used
- Does a driver need to know all the details about the production and the functioning of the parts?
 - No! He just needs to drive the car.

What is an object?

- Objects are specific instances of classes
- Objects have a *state* and *behavior*



A car has a

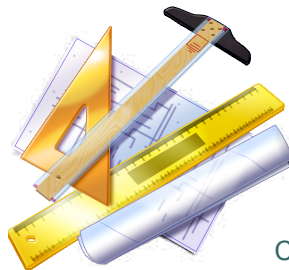
-color, speed, capacity

} state

-a pedal for
accelerating and
breaking

} behaviour

- Classes are the blueprints containing the details about the state and the behaviour of objects



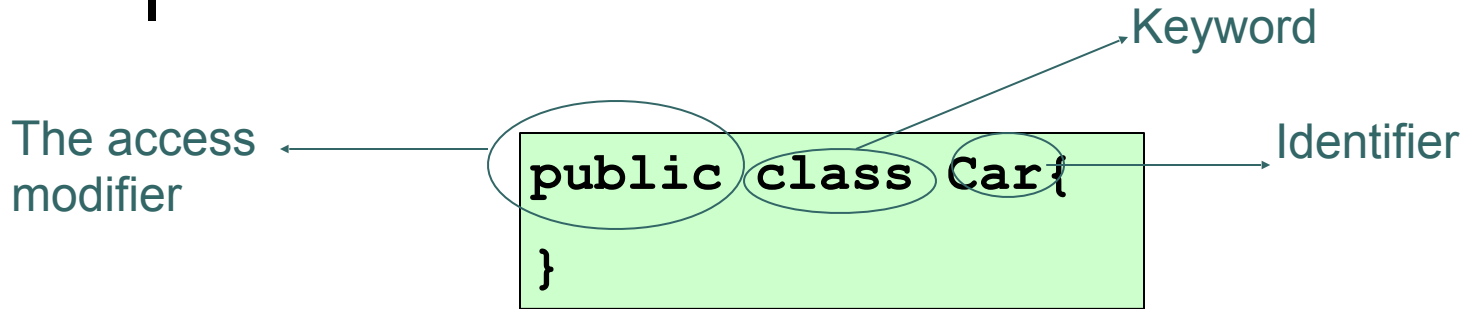
Car class:

-What happens in the engine to accelerate/break?

State & Behavior of an Object

- **State**: Classes contain one or more attributes
 - Specified by **instance variables**
 - Carried with the object as it is used
- **Behavior**: Classes provide one or more methods
 - **Method** represents task in a program
 - Describes the mechanisms that actually perform its tasks
 - Hides from its user the complex tasks that it performs
 - Method call tells method to perform its task

Class Declaration



State: *instance variables*

```
public class Car{
    String color = "red";
    int capacity = 5;
}
```

Behavior: *method*

```
public class Car{
    String color;
    int capacity;
    void moveForward() {}
}
```

The driver calls the `moveForward()` message in order to move the car.

OOP Rule - Abstraction

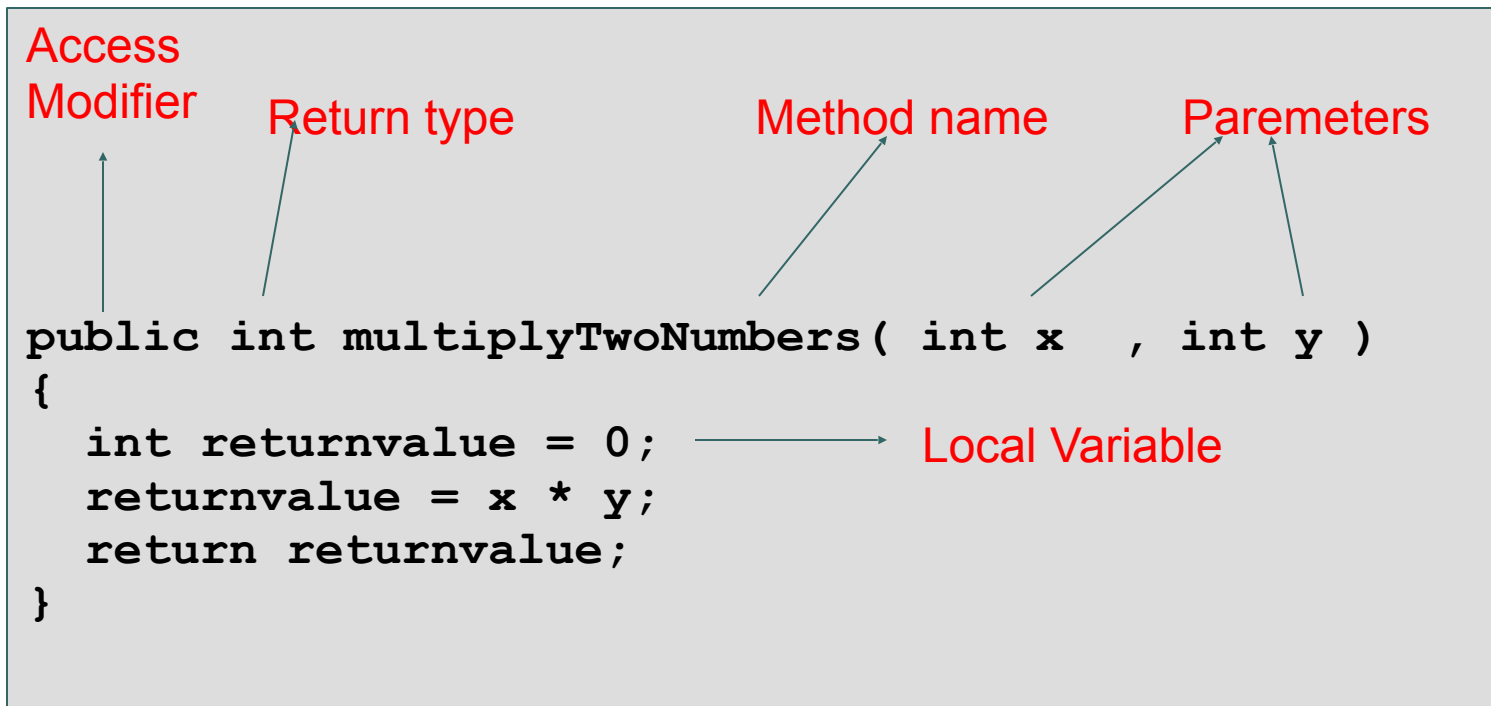
- Abstraction is simplifying complex reality by modeling classes appropriate to the problem
- Computer programmers use abstraction to understand and solve problems and communicate their solutions with the computer in some particular computer language
- Ex : Real life Car → Car Class

Methods

- Set of statements that performs a specific task
- Single unit, and named with an identifier
- Functions are usually called “Methods” in Java
- “main” method does the all job and calls the other functions
- A program can have only one “main” function as a starting point
- Methods are reusable, therefore they can be called from main method more than once or they can be called from other functions
- Every method can have its own variables that are declared and used in its scope but these **local variables** can not be accessed directly from outer scope of the methods

Methods

- Parameters are used to send data to a method within the method they behave like variables
- Return values are used to receive data from a method
- **Method declaration in Java:**



The diagram shows a Java method declaration with labels pointing to its components:

- Access Modifier** points to `public`
- Return type** points to `int`
- Method name** points to `multiplyTwoNumbers`
- Paremters** points to `(int x , int y)`
- Local Variable** points to `int returnvalue = 0;`

```
public int multiplyTwoNumbers( int x , int y )
{
    int returnvalue = 0;
    returnvalue = x * y;
    return returnvalue;
}
```

Methods

- If a method doesn't return any value instead of a return type we use the keyword *void* (*Ex: main method*)

```
public static void main ( String [ ] args )  
{  
    int result = multiplyTwoNumbers(28,1290);  
    System.out.println ( "The result is = " + result );  
}
```

Variable Length Argument Lists

- Unspecified number of arguments
- Use ellipsis (...) in method's parameter list
 - Can occur only once in parameter list
 - Must be placed at the end of parameter list
- Array whose elements are all of the same type

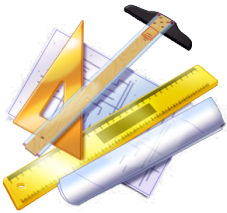
```
public double avarage(double... numbers) {  
    double total = 0.0;  
    for(double d: numbers) {  
        total += d;  
    }  
    return total/numbers.length;  
}
```

Basic Scope Rules

- Scope of a **parameter** declaration is the body of the method in which appears
- Scope of a local-variable declaration is from the point of declaration to the end of that block
- Scope of a **local-variable** declaration in the initialization section of a `for` header is the rest of the `for` header and the body of the `for` statement
- Scope of a method or **field of a class** is the entire body of the class

Instantiating an Object from a Class

Class: Car



Class name

Object: myCar



identifier or
reference

keyword

```
Car    myCar    =    new Car () ;  
myCar.accelerate () ;
```

} myCar Object

Method call using “.”

Constructors

- A method to initialize variables of a class
- Java requires a constructor for every class
- Java will provide a default no-argument constructor if none is provided
- Called when keyword `new` is followed by the class name and parentheses
- Constructors can take parameters but cannot return a value
- Constructors can be overloaded

```
public class Car{
    private String color;
    private int capacity;

    public Car(){
        color = "Blue";
        capacity = 5;
    }
}
```

constructor is a method with the same name of the class without a return value

Packages & import Statements

- Packages are directories containing group of classes
- Package declaration should be made before any statements
- `import` statements are used in order to instantiate and use methods of classes in other packages
- can use “*” to import all classes of a package

```
package com.myPackage;
import java.util.Scanner;
public class Test{
    public static void main(String args[]){
        Scanner input = new Scanner(System.in);
        String name = input.nextLine();
        System.out.println(name);
    }
}
```

1- declare package

2- import statements

3- class declaration

Packages & import Statements

- `java.lang` is implicitly imported into every program (contains `System`, `String`, etc.)
- Imports unnecessary if fully-qualified names are used (ex: *`java.util.ArrayList`*)

Access Modifiers - `public` and `private`

- `public/private` access modifiers can be used with methods and instance variables
- **Classes cannot be `private`**
- `private` variables and methods are accessible only to methods of the class in which they are declared
- `public` variables and methods are accessible to methods of other classes and objects

Access Modifiers - protected and default

- protected access modifier:
 - **Classes cannot be protected**
 - protected fields and methods can only be accessed from the same package
- default access modifier
 - If no access mod. is defined classes, fields and methods act like protected

OOP Rule - Encapsulation

- It is the hiding of the internal mechanisms and data structures (*state*) of a component
- Users of the component only need to know what the comp. does, and cannot make themselves dependent on the details of how it does it
- The purpose is to achieve potential for change
- Also protects the integrity of the object
- Ex: Driver only needs to know how to drive the car

Set - Get Methods

- Client of an object calls the class's public methods to manipulate the private fields of an object of the class

```
public class Car{  
    private String color;  
  
    public void setColor(String color){  
        this.color = color;  
    }  
    public String getColor(){  
        return this.color;  
    }  
}
```

Shadowing

- When a method, declares its own local variables with the same identifiers of some instance variables, the inner variable *shadows* the outer variable.
- To overcome shadowing we use *this* keyword to refer to the instance variables of the class

```
public class Car{
    private String color;

    public Car(String color){
        this.color = color;
    }
    public void setColor(String color){
        this.color = color;
    }
}
```

Primitive Types vs. Reference Types

- *Primitive types* (**boolean, byte, short, int, long, float, double**) can store only one value
- Primitive type instance variables are initialized by default (*byte, short, int, long, float, double to "0", boolean to "false"*)
- All other types except primitive are *reference types* (like **objects** we created)
- Reference types hold a reference pointing to the actual object in the memory
- Ref. type instance variables are initialized by default to *null*

Reference Type Assignments

- When reference of an object is assigned to a previously created reference, the two objects point to the same object in the memory

```
Student stu1 = new Student();
stu1.setName("Ahmet");
Student stu2 = stu1;
System.out.println(stu1.getName());
stu2.setName("Mehmet");
System.out.println(stu1.getName());
```

Output:
Ahmet
Mehmet

stu1 and stu2 refer to the same object in memory. So, whenever one changes the other is affected too

Argument Promotion and Casting

- Java will promote a method call argument to match its corresponding method parameter according to the promotion rules
- Values in an expression are promoted to the “highest” type in the expression (a temporary copy of the value is made)
- Converting values to lower types results in a compilation error, unless the programmer explicitly forces the conversion to occur
 - Place the desired data type in parentheses before the value
 - example: `(int) 4.5`

Argument Promotion and Casting

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

```
double d = 3 + 3.1233;
long l = 3 + 12000L;
```

3's are ints but implicitly converted to double and long respectively

```
int c = 12;
void someMethod() {
    doSomething(c);
}
void doSomething(double d){}
```

c is an int but passed to the method which takes a double argument. casting occurs implicitly

Pass-by-value / Pass-by-Reference

- Pass-by-value

- When a method is invoked with a *primitive* type parameter *the value* of the variable is copied and passed to the method, the original variable isn't effected

```
void someMethod(){  
    int x = 3;  
    changeValue(x);  
    System.out.println("x= " + x);  
}  
  
public void changeValue(int x){  
    x += 2;  
    System.out.println("x= " + x);  
}
```

Output:

```
x= 5  
x= 3
```

Pass-by-value / Pass-by-Reference

- Pass-by-reference

- When a method is invoked with a *reference* type parameter a copy of *the reference* of the object is passed to the method.
- So, both references point to the same object and the object's state can be modified through a method

```
void someMethod() {  
    Student s = new Student();  
    changeValue(s);  
    System.out.println("Name: " + stu.getName());  
}  
  
public void changeValue(Student stu) {  
    stu.setName("Ali");  
    System.out.println("Name: " + stu.getName());  
}
```

Output: Name: Ali
Name: Ali

Pass-by-value / Pass-by-Reference

- Passing arrays/array elements to methods
 - Arrays are reference types, so their reference is copied and send to the method as parameter
 - Array elements differ according to the type of the element.
 - primitive type → pass by value
 - reference type → pass by reference

static Methods

- Applies to the class as a whole instead of a specific object of the class
- Call a **static** method by using the method call:
ClassName.methodName (arguments)
- All methods of the **Math** class are **static**
 - example: **Math.sqrt (900.0)**
- **static** methods cannot call non-**static** methods/fields of the same class directly

```
public static int makeSum(int a, int b){  
    return a + b;  
}
```

static fields

- **static** fields (or class variables)
 - Are fields where one copy of the variable is shared among all objects of the class
- **Ex : Math.PI**
- Used when:
 - all objects of the class should share the same copy of this variable or
 - this variable should be accessible even when no objects of the class exist

Why main method is static

- Method `main`
 - `main` is declared `static` so it can be invoked without creating an object of the class containing `main`
 - Any class can contain a `main` method
 - The JVM invokes the `main` method belonging to the class specified by the first command-line argument to the `java` command

Constants

- Constants
 - Keyword **final**
 - Cannot be changed after initialization
 - Should be assigned a value at declaration or constructor
- **Math.PI** and **Math.E** are **final static** fields of the **Math** class

Method Overloading

- Multiple methods with the same name, but different types, number or order of parameters in their parameter lists
- Compiler decides which method is being called by matching the method call's argument list (**method signature**) to one of the overloaded methods' parameter lists
 - A method's name and number, type and order of its parameters form its signature
- Differences in return type are irrelevant in method overloading
 - Overloaded methods can have different return types
 - Methods with different return types but the same signature cause a compilation error

Lab: The Pen

- Create three objects:
 - Circle : has a radius (int) and color (String)
 - Rectangle: has a width and height (both int) and color (String)
 - Pen: Has two methods:
 - draw (Rectangle r) : writes out the area of “r”
 - draw (Circle c): writes out the area of “c”
 - changeColor (String color, Rectangle r)
 - changeColor (String color, Circle c) : changes color of shapes and writes them out
- From main method, instantiate a Pen and practice methods of it