



# Java Programming

## Unit 2. Part 1.

Intro to Object-Oriented  
Programming

# Classes, methods, properties

- Java is an object-oriented language. Its constructs represent objects from the real world.
- Each Java program *has at least one class may have fields and/or* implements certain behavior.
- Classes in Java may have *methods* (similar to functions) and *properties* (a.k.a. *attributes* or *fields*).
- Starting from Java 8, behavior can be implemented using lambda expressions (covered in Unit 19).

# The Class Car

**Fields** represent some properties of a car, e.g. the number of doors or the body color.

`numberOfDoors` is a variable of type `int` – to store integers; `color` can hold a string of characters (text).

**Local variables** are declared inside methods, **Fields (a.k.a. member variables)** - outside

Single-line comments start with `//`  
Multi-line comments go between `/*` and `*/`

**Methods** describe what our car can do: stop and start the engine.

```
class Car{
    String color;
    int numberOfDoors;

    void startEngine() {
        // Some code goes here
    }
    void stopEngine () {
        int tempCounter=0;
        // Some code goes here
    }
}
```

This class has no `main()` method. What does it mean?

# How many cars can you create?

- A class definition is a blueprint from which you can create one or more instances of the class Car.

- These **two instances** are created with the **new** operator:

```
Car car1 = new Car();  
Car car2 = new Car();
```

Where do you write this code?

Let's do it in the `main()` method of another class

- Now the variables **car1** and **car2** represent these instances:

```
car1.color="blue";  
car2.color="red";
```

# The class TestCar with the method main()

```
class TestCar{

    public static void main(String[] args){

        Car car1 = new Car();    // creating one instance
        Car car2 = new Car();    // creating another instance

        car1.color="blue";
        car2.color="red";

        // Printing a message on the system console
        System.out.println("These cars have been painted ");
    }
}
```

# Walkthrough 1

- Create a new Eclipse Java project called **OOP**
- Write a class **Car** using the sample code from the previous slide.
- Write a class **TestCar** that creates two instances of the class **Car**, changes their colors and prints the message about it.
- Run the class **TestCar**. Observe the message in the View Console.

# Inheritance – James Bond Car

In object-oriented languages the term *inheritance* means an ability to define a new class based on an existing one.

```
class JamesBondCar extends Car{
```

```
    int currentSubmergeDepth;  
    boolean isGunOnBoard=true;  
    final String MANUFACTURER;
```

```
    void submerge() {  
        currentSubmergeDepth = 50;  
        // Some code goes here  
    }
```

```
    void surface() {  
        // Some code goes here  
    }  
}
```

```
class JamesBondCar extends Car{
```

```
// ...
```

```
}
```

The class JamesBondCar has everything that the class Car has *plus something else*.

In this example, it defines:

- three more attributes

- two more methods.

# Variable and constants

Java is statically typed language

```
int currentSubmergeDepth;           // an integer variable
boolean isGunOnBoard=true;          // a boolean variable
final String MANUFACTURER="GAZ";    // a final text variable (a.k.a. constant)
```

**First declare a variable, then use it.** You can assign and change the value of the variable many times:

```
currentSubmergeDepth = 25;
...
currentSubmergeDepth = 30;
```

You can assign the value to a final variable **only once** and can't change it afterward.

```
MANUFACTURER = "Toyota";
```

Read more on variable types in Lesson 3 of the textbook.



# Yet another example: class Tax

```
class Tax {  
    double grossIncome;  
    String state;  
    int dependents;  
  
    public double calcTax()  
    {  
        return 234.55;  
    }  
}
```

To calculate taxes, you can declare a class **Tax** that will have some properties to store the values, required for calculations.

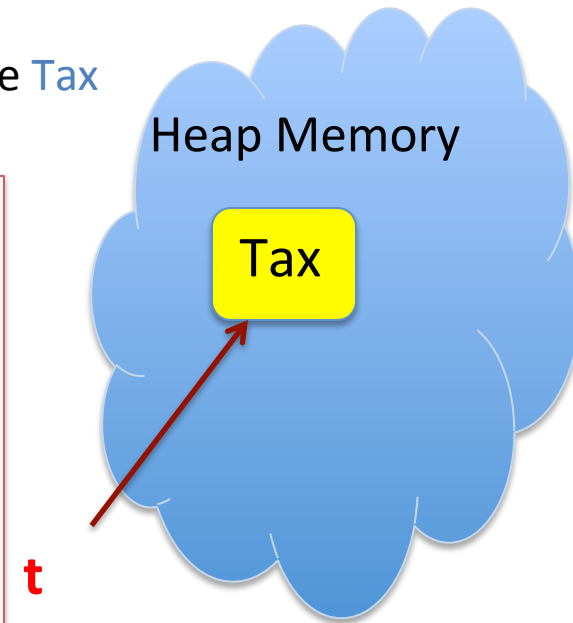
Then add methods that implement required *behavior*.

The keyword **double** in the method signature means that the method **calcTax()** will **return** the result of calculations as a double precision value.

# Testing the class Tax

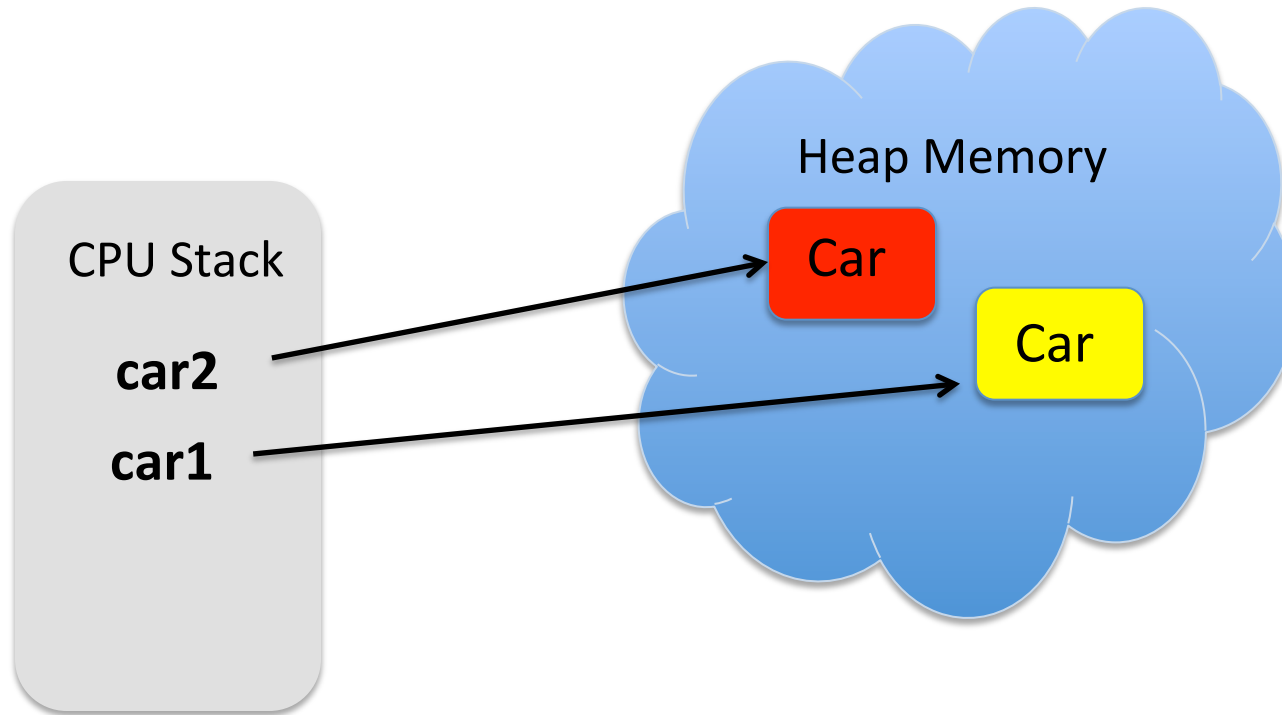
The class `TestText` has the `main()` method, which will instantiate `Tax` and calls its method `calcTax()`.

```
class TestTax{  
  
    public static void main(String[] args){  
  
        Tax    t = new Tax(); // creating an instance of  
Tax  
  
        t.grossIncome= 50000; // assigning the values  
        t.dependents= 2;  
        t.state= "NJ";  
  
        double yourTax = t.calcTax(); //calculating tax  
  
        // Printing the result  
        System.out.println("Your tax is " + yourTax);  
    }  
}
```



Note the use of the variable `t`, which knows the address of the instance of the class `Tax` in memory.

# Stack and Heap



# if-statement

Sometimes you need to change the flow of the code execution. You can do it with the if-statement:

```
if (totalOrderPrice <= 200){  
    System.out.println("You'll get a 20% discount");  
}  
else if (totalOrderPrice > 200 && totalOrderPrice <=300){  
    System.out.println("You'll get 25% discount");  
}  
else {  
    System.out.println("You'll get 30% discount");  
}
```

# Walkthrough 2

- Add two more classes to the OOP project: `Tax` and `TestTax`.
- Run the `TestTax` and observe that it always prints 234.55 as calculated text
- Modify the code of the `calcTax()` method to print the tax as 6% of of gross income if it was up to \$50000 and 8% otherwise.
- Run the `TestTax` program and see if the tax is properly calculated. Change the value of `grossIncome` and re-run the program.

# switch-statement

The `switch` statement is an alternative to `if`. The case label in the switch condition (`taxCode`) is evaluated and the program goes to one of the following `case` clauses:

```
int taxCode=someObject.getTaxCode(grossIncome);

switch (taxCode){
    case 0:
        System.out.println("Tax Exempt");
        break;
    case 1:
        System.out.println("Low Tax Bracket");
        break;
    case 2:
        System.out.println("High Tax Bracket");
        break;
    default:
        System.out.println("Wrong Tax Bracket");
}

//some other code goes here
```

Don't forget about the `break` statements to avoid the *fall through* situation.

As of Java 7 you can use the `String` type in the `switch` statement:

```
switch (taxCategory){
    case "rich":
        ...
        break;
    case "poor":
        ...
}
```

# Method Arguments

External data can be provided to a method in the form of *arguments* (a.k.a. *parameters*).

In the method signature declare the data type and the name of each argument.

For example, the method `calcLoanPayment()` has 3 arguments:

```
int calcLoanPayment(int amount, int numberOfMonths, String state){  
  
    // Your code goes here  
  
}
```

You can call this method passing the values for the payment calculations as arguments:

```
calcLoanPayment(20000, 60, "NY");
```

This next method invocation will cause the compilation error if the class has no method with 2 arguments:

```
calcLoanPayment(20000, 60);
```

# Another example of inheritance

The subclass `NJTax` defines a new method `adjustForStudents()`:

```
class NJTax extends Tax{  
  
    double adjustForStudents (double stateTax){  
        double adjustedTax = stateTax - 500;  
        return adjustedTax;  
    }  
}
```

The `TestTax` can instantiate `NJTax` and use methods and fields from both *super* and *subclasses*:

```
NJTax t= new NJTax();  
  
double yourTax = t.calcTax();  
double totalTax = t.adjustForStudents (yourTax);
```



# Method Overriding

If a subclass has the method with the same name and argument list, it will *override* (suppress) the corresponding method of its ancestor.

Method overriding comes handy in the following situations:

- The source code of the super class is not available, but you still need to change its functionality.
- The original version of the method is still valid in some cases, and you want to keep it as is.

# Walkthrough 3

- Add to the OOP project the class `NJTax` that will have the `calcTax()` method with the same signature as in `Tax`
- Add the code to `NJTax.calcTax()` method to print the tax as 10% of gross income if it was up to \$50000 and 13% otherwise.
- Run existing `TestTax` and observe that your changes didn't have any effect on the calculate tax. Why?
- Change the code of the `TestTax` to instantiate `NJTax` instead of `Tax`.
- Run the `TestTax` program again and observe that now the new percentage is properly applied. You are using the *overriden* version of the method `calcTax()`.

# Method Overloading

*Method overloading* means having a class with more than one method having the same name but different argument lists.

```
class Financial{
    ...
    int calcLoanPayment(int amount, int numberOfMonths){
        // by default, calculate for New York state
        calcLoanPayment(amount, numberOfMonths, "NY");
    }

    int calcLoanPayment(int amount, int numberOfMonths,
                        String state){

        // Your code for calculating loan payments goes here

    }
    ...
}
```

A method can be overloaded not only in the same class but in a descendant too.

# Constructors are special methods

When a program creates an instance of a class using `new`, Java invokes the class's *constructor* — a special method that is called only once :

```
Tax t = new Tax(40000, "CA",4);
```

- Constructors are called when the class is being instantiated.
- They must have the same name as the class.
- They can't return a value and you don't use `void` as a return type.

```
class Tax {  
    double grossIncome; // class variables  
    String state;  
    int dependents;  
  
    // Constructor  
    Tax (double gi, String st, int depen){  
        grossIncome = gi; // class variable initialization  
        state = st;  
        dependents=depen;  
    }  
}
```

# Homework

Study the lessons 3 and 4 from the textbook and do the assignments from the Try It sections of these lessons.

## Additional Reading

Java SE 8 documentation:

<http://docs.oracle.com/javase/8/>

Overriding vs. hiding:

<http://www.coderanch.com/how-to/java/OverridingVsHiding>

Eclipse Debugging tutorial by Lars Vogel:

<http://www.vogella.com/articles/EclipseDebugging/article.html>

Redmonk's rating of programming languages, Q2 2014:

<http://redmonk.com/sograzy/2014/06/13/language-rankings-6-14/>