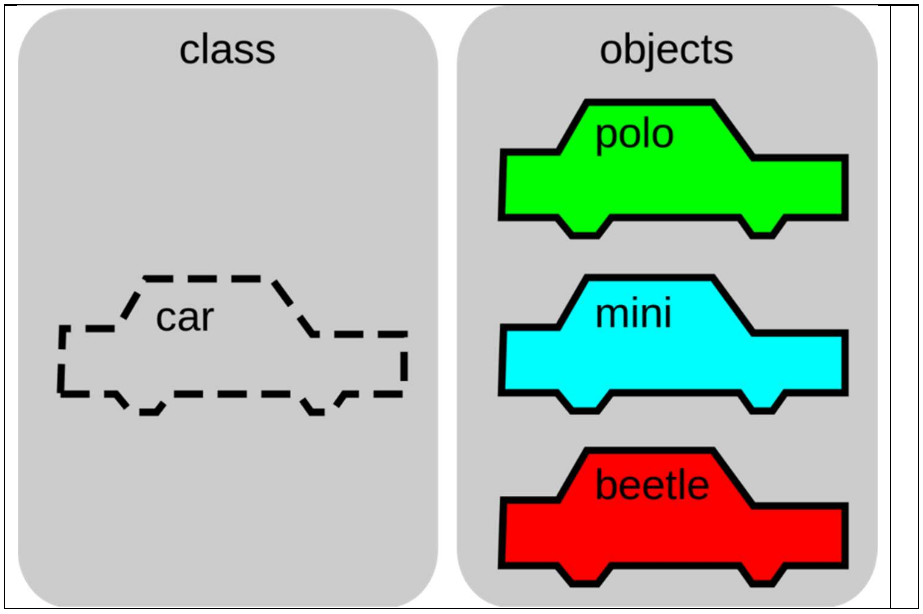
In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.



The outlines of this lecture

- **Adding a Method to the Vehicle Class**
- **Returning from a Method**
- **Returning a Value**
- **Using Parameters in the Method**
- **Adding a Parameterized Method to Vehicle**

Classes and Methods:

As explained, **instance variables** and **methods** are the constituents of classes. So far, the **Vehicle** class **contains data**, but no methods. Although data-only classes are perfectly valid, most classes will have methods. Methods are subroutines that manipulate the data defined by the class and, in many cases, provide access to that data.

In most cases, other parts of the program will interact with a class through its methods. A method contains one or more statements. In well-written Java code, **each method performs only one task**. **Each method has a name, and it is this name that is used to call the method.**

In general, you can give a method whatever name you please. However, remember that **main( )** is reserved for the method that begins execution of your program. Also, don’t use Java’s keywords for method names.

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

A method will have parentheses after its name. For example, if a method's name is **getval**, it will be written **getval( )** when its name is used in a sentence. This notation will help you distinguish variable names from method names.
The general form of a method is shown here:

```
type name( parameter-list ) {
// body of method
}
```

Here, **type** specifies the **type of data returned** by the method. This can be any valid type, including class types that you create. If the **method does not return** a value, its return type must be **void**. The name of the method is specified by **name**. This can be any legal identifier other than those already used by other items within the current scope. The **parameter-list** is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.
If the method has no parameters, the parameter list will be empty.

Adding a Method to the Vehicle Class

As just explained, the methods of a class typically manipulate and provide access to the data of the class. Recall that **main( )** in the preceding examples computed the **range of a vehicle** by multiplying its fuel consumption rate by its fuel capacity. While technically correct, this is not the best way to handle this computation. The calculation of a **vehicle's range is something that is best handled by the Vehicle class itself**. The reason for this conclusion is easy to understand: the range of a vehicle is dependent upon the capacity of the fuel tank and the rate of fuel consumption, and **both quantities are encapsulated by Vehicle**. By **adding a method** to **Vehicle** that **computes the range**, you are enhancing its object-oriented structure. To add a method to **Vehicle**, specify it within **Vehicle**'s declaration. For example, the following version of **Vehicle** contains a method called **range( )** that displays the range of the vehicle.

```
// Add range to Vehicle.
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon

// Display the range.
void range() {
System.out.println("Range is " + fuelcap * mpg);
}
}
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

```
class AddMeth {
public static void main(String args[]) {
Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
int range1, range2;

// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;
System.out.print("Minivan can carry " + minivan.passengers +". ");

minivan.range(); // display range of minivan

System.out.print("Sportscar can carry " + sportscar.passengers +". ");
sportscar.range(); // display range of sportscar.
}
}
```

This program generates the following output:

```
Minivan can carry 7. Range is 336
Sportscar can carry 2. Range is 168
```

Let's look at the key elements of this program, beginning with the **range( )** method itself.

The first line of **range( )** is

```
void range() {
```

The **range( )** method is contained within the **Vehicle** class. Notice that **fuelcap** and **mpg** are used directly, without the dot operator.

This line declares a method called **range** that has no parameters. Its return type is **void**. Thus, **range( )** does not return a value to the caller.

The body of **range( )** consists solely of this line:

```
System.out.println("Range is " + fuelcap * mpg);
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

This statement displays the range of the vehicle by multiplying **fuelcap** by **mpg**. Since each object of type **Vehicle** has its own copy of **fuelcap** and **mpg**, when **range( )** is called, the range computation uses the calling object's copies of those variables.
The **range( )** method ends when its closing curly brace is encountered. This causes program control to transfer back to the caller.
Next, look closely at this line of code from inside **main( )**:

minivan.range();

This statement invokes the **range( )** method on **minivan**. That is, it calls **range( )** relative to the **minivan** object, using the object's name followed by the dot operator. When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call. In this case, the call to **minivan.range( )** displays the range of the vehicle defined by **minivan**.
In similar fashion, the call to **sportscar.range( )** displays the range of the vehicle defined by **sportscar**. Each time **range( )** is invoked, it displays the range for the specified object.
There is something very important to notice inside the **range( )** method: the instance variables **fuelcap** and **mpg** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **fuelcap and mpg inside range( ) implicitly refer to the copies of those variables found in the object that invokes range( )**.

Returning from a Method

In general, there are **two conditions that cause a method to return**.
- **First**, as the **range( )** method in the preceding example shows, when the method's **closing curly brace** **}** is encountered.
- **Second** is when a **return** statement is executed. There are two forms of **return**:

- **one** for use in **void methods** (those that do not return a value) and,
- **Second** for **returning values**.

In a **void** method, you can cause the immediate termination of a method by using this form of **return**:

return ;

When this statement executes, program control returns to the caller, skipping any remaining code in the method. For **example**, consider this method:

```
void myMeth() {
int i;
for(i=0; i<10; i++) {
   if(i == 5) return; // stop at 5
System.out.println();
 }
}
```

Here, the **for** loop will only run from 0 to 5, because once **i** equals 5, the method returns. It is permissible to have multiple return statements in a method, especially when there are two or more routes out of it. For **example**:

```
void myMeth() {
// ...
if(done) return;
// ...
if(error) return;
}
```

Here, the method returns if it is done or if an error occurs. Be careful, however, because having too many exit points in a method can destructor your code; so, avoid using them casually. A well-designed method has well-defined exit points.
To review: a **void** method can **return in one of two ways**—**its closing curly brace** is reached, or a **return** statement is executed.

Returning a Value

You can use a return value to improve the implementation of **range( )**. Instead of displaying the range, a better approach is to have **range( )** compute the range and return this value. Among the advantages to this approach is that you can use the value for other calculations. The following example modifies **range( )** to return the range rather than displaying it.

```
// Use a return value.
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon

// Return the range.
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

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

class RetMeth {
public static void main(String args[]) {
Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
int range1, range2;

// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;

// get the ranges
range1 = minivan.range();
range2 = sportscar.range();

System.out.println("Minivan can carry " + minivan.passengers +" with range of " +
range1 + " Miles");

System.out.println("Sportscar can carry " + sportscar.passengers +" with range of " +
range2 + " miles");
}
}
```

The output is shown here:

```
Minivan can carry 7 with range of 336 Miles
Sportscar can carry 2 with range of 168 miles
```

In the program, notice that when **range( )** is called, it is put on the right side of an assignment statement. On the left is a variable that will receive the value returned by

range( ).

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

Thus, after
range1 = minivan.range();

executes, the range of the **minivan** object is stored in **range1**.

Notice that **range( )** now has a return type of **int**. This means that it will return an integer value to the caller. The return type of a method is important because the type of data returned by a method must be **compatible** with the return type specified by the method. Thus, if you want a method to return data of type **double**, its return type must be **type double**.

Specifically, there is no need for the range1 or range2 variables. A call to range( ) can be used in the println( ) statement directly, as shown here:

```
System.out.println("Minivan can carry " + minivan.passengers +" with range of " + minivan.range() + " Miles");
```

In this case, when **println( )** is executed, **minivan.range( )** is called automatically and its value will be passed to **println( )**. Furthermore, you can use a **call to range( ) whenever the range of a Vehicle object is needed**. For **example**, this statement compares the ranges of two vehicles:

```
if(v1.range() > v2.range()) System.out.println("v1 has greater range");
```

Using Parameters in the Method

It is possible to **pass one or more values to a method** when the method is called. As explained, a value passed to a method is called an **argument**. Inside the method, the variable that receives the argument is called a **parameter**. Parameters are declared inside the parentheses that follow the method's name. The parameter declaration syntax is the same as that used for variables.
A parameter is within the scope of its method, and aside from its special task of receiving an argument, it acts like any other local variable.

Here is a simple example that uses a parameter. Inside the **ChkNum class**, the method **isEven( )** returns **true** if the value that it is passed is even. It returns **false** otherwise. Therefore, **isEven( )** has a return type of **boolean**.

```
// A simple example that uses a parameter.
class ChkNum {
// return true if x is even
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

```
boolean isEven(int x) {
if((x%2) == 0) return true;
else return false;
}
}

class ParmDemo {
public static void main(String args[]) {

ChkNum e = new ChkNum();

if(e.isEven(10)) System.out.println("10 is even.");
if(e.isEven(9)) System.out.println("9 is even.");
if(e.isEven(8)) System.out.println("8 is even.");
}
}
```

Here is the output produced by the program:

```
10 is even.
8 is even.
```

In the program, **isEven( )** is called three times, and each time a different value is passed. Let's look at this process closely. First, notice how **isEven( )** is called. The argument is specified between the parentheses. When **isEven( )** is called the first time, it is passed the value 10. Thus, when **isEven( )** begins executing, the parameter **x** receives the value 10. In the second call, 9 is the argument, and **x**, then, has the value 9. In the third call, the argument is 8, which is the value that **x** receives. The point is that the value passed as an argument when **isEven( )** is called is the value received by its parameter, **x**.

Adding a Parameterized Method to Vehicle

You can use a parameterized method to add a new feature to the **Vehicle** class: the ability to **compute the amount of fuel needed for a given distance**. This new method is called **fuelneeded( )**. This method takes the number of miles that you want to drive and returns the number of gallons of gas required. The **fuelneeded( )** method is defined like this:

```
double fuelneeded(int miles) {
return (double) miles / mpg;
}
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

Notice that this method returns a value of type **double**. This is useful since the amount of **fuel needed for a given distance might not be an even number**.
The entire **Vehicle** class that includes **fuelneeded( )** is shown here:

```
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon

// Return the range.
int range() {
return mpg * fuelcap;
}

// Compute fuel needed for a given distance.

double fuelneeded(int miles) {
return (double) miles / mpg;
 }
}

class CompFuel {
public static void main(String args[]) {
Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
double gallons;
int dist = 252;

// assign values to fields in minivan

minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar

sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;

gallons = minivan.fuelneeded(dist);
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

```
System.out.println("To go " + dist + " miles minivan needs " +gallons + " gallons of fuel.");

gallons = sportscar.fuelneeded(dist);

System.out.println("To go " + dist + " miles sportscar needs " +gallons + " gallons of fuel.");
}
}
```

The output from the program is shown here:

```
To go 252 miles minivan needs 12.0 gallons of fuel.
To go 252 miles sportscar needs 21.0 gallons of fuel.
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

Example:

Calculator class doesn’t have variables, and have **two methods**, **add** (to add two integers’ numbers), and **great** (to display name). write java code **that represent the** Calculator class in addition to main program.

```
class Calculator {
   // Method to add two integers
   public int add(int a, int b) {
      return a + b;
   }

   // Method to display a message
   public void greet(String name) {
      System.out.println("Hello, " + name + "!");
   }
}

public class Main {
public static void main(String[] args) {

Calculator myCalc = new Calculator(); // Create an object
int sum = myCalc.add(5, 3); // Call the add method
System.out.println("Sum: " + sum); // Output: Sum: 8

myCalc.greet("Alice"); // Call the greet method
      // Output: Hello, Alice!
   }
}
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

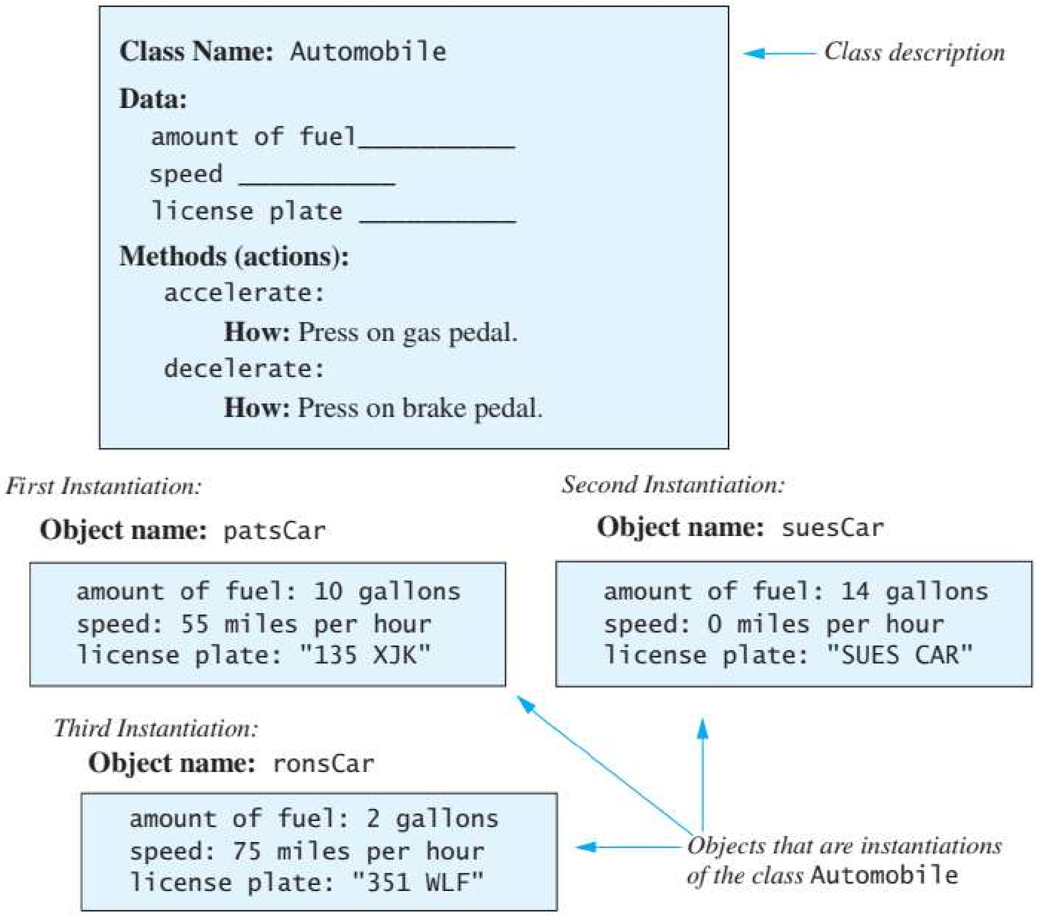
Example

Student class have **variables** (rollno and two marks), and have **one method**, **total** (without parameters), return the **float** computing the adding of two marks. write java code **that represent the** Student class in addition to main program.

```
class student{
int rollno;
float marks1,marks2;
float total(){
return(marks1+marks2); } }

class studentCreate{
public static void main(String args[]){
float totalmarks;

//one student with rollno=25, marks=81, marks2=85 is created
student student1=new student();
student1.rollno=25;
student1.marks1=81;
student1.marks2=85;

//second student with rollno=26, marks=80. marks2=95 is created
student student2=new student();
student2.rollno=26;
student2.marks1=80;
student2.marks2=95;

//calculate marks of student1 by calling total() of student class
totalmarks=student1.total();
System.out.println("total marks="+ totalmarks);

//claculate marks of student2 by calling total() of student class
totalmarks=student2.total();
System.out.println("total marks="+totalmarks);
}
}

/*Output:
total marks=166.0
total marks=175.0*/
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

Example



```
public void accelerate() {
    speed += 5;}
public void decelerate () {
    speed -=5; }
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

<u>Example</u>

Definition of a Dog Class

```
public class Dog
{
public String name;
public String breed;
public int age;

public void writeOutput()
{
System.out.println("Name: " + name);
System.out.println("Breed: " + breed);
System.out.println( "Age in calendar years: " + age);
System.out.println("Age in human years: " +getAgeInHumanYears());
System.out.println(); }

public int getAgeInHumanYears()
{
int humanAge = 0;

if (age <= 2)
{ humanAge = age * 11; }
else
{ humanAge = 22 + ((age-2) * 5); }
return humanAge;
}
}
```

Using the Dog Class and its Methods

```
public class DogDemo
{
public static void main(String[] args) {

Dog balto = new Dog();
balto.name = "Balto";
balto.age = 8;
balto.breed = "Siberian Husky";
balto.writeOutput();
```

In Object-Oriented Programming (OOP), a class serves as a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will possess.

```
Dog scooby = new Dog();
scooby.name = "Scooby";
scooby.age = 42;
scooby.breed = "Great Dane";

System.out.println( scooby.name + " is a " + scooby.breed + ".");
System.out.print( "He is " + scooby.age + " years old, or ");

int humanYears = scooby.getAgeInHumanYears();
System.out.println(humanYears + " in human years."); } }
```

Sample Screen Output

```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52
Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```