



# Classes

## Lecture 4

**Department of Computer Engineering**  
**INHA University**  
**Dr. Tamer ABUHMED**



# Outline



- Class Constructor
- Overloaded Constructors
- Class *Set* and *Get* Methods
- Class Composition
- Class static members
  - Creating static variables and methods.
  - Importing static variables and methods from other class.
- Creating Packages
- Access Packages

# Class Constructor



- Every class must have at least one constructor.
- If you do not provide any constructors in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked.
  - The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, false for **Boolean** values and **null** for references).
- If your class declares constructors, the compiler will not create a default constructor.
  - In this case, you must declare a no-argument constructor if default initialization is required.
  - Like a default constructor, a no-argument constructor is invoked with empty parentheses.

# Referring to the Current Object's Members with the `this` Reference



- Every object can access a reference to itself with keyword `this`.
- When a non-static method is called for a particular object, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods.
  - Enables the class's code to know which object should be manipulated.
  - Can also use keyword `this` explicitly in a non-static method's body.
- Can use the `this` reference implicitly and explicitly.

# Class Constructor Overloading

```
public class StudentData {
```

Class

```
private int studentID;
```

```
private String studentName;
```

```
private int studentAge;
```

```
StudentData() {
```

Constructor

```
// Default constructor
```

```
studentID = 100;
```

```
studentName = "New Student";
```

```
studentAge = 18;
```

```
}
```

```
StudentData(int num1,String str,int num2)  
{
```

Constructor

```
// Parameterized constructor
```

```
studentID = num1;
```

```
studentName = str;
```

```
studentAge = num2;
```

```
}
```

```
// Getter and setter methods
```

```
public int getStudentID() {
```

```
return studentID;
```

```
}
```

```
public void setStudentID(int studentID) {
```

```
this.studentID = studentID;
```

```
}
```

```
public String getStudentName() {
```

```
return studentName;
```

```
}
```

```
public void setStudentName(String  
studentName) {
```

```
this.studentName = studentName;
```

```
}
```

```
public int getStudentAge() {
```

```
return studentAge;
```

```
}
```

```
public void setStudentAge(int studentAge)  
{
```

```
this.studentAge = studentAge;
```

```
}
```

```
}
```

```
class TestOverloading {
```

Class

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

```
// This object creation would call the  
default constructor
```

```
StudentData myobj = new StudentData();
```

```
System.out.println("Student Name is: " +  
myobj.getStudentName());
```

```
System.out.println("Student Age is: " +  
myobj.getStudentAge());
```

```
System.out.println("Student ID is: " +  
myobj.getStudentID());
```

```
StudentData myobj2 = new StudentData(555,  
"Chaitanya", 25);
```

```
System.out.println("Student Name is: " +  
myobj2.getStudentName());
```

```
System.out.println("Student Age is: " +  
myobj2.getStudentAge());
```

```
System.out.println("Student ID is: " +  
myobj2.getStudentID());
```

```
}
```

```
}
```

# Class Constructor Overloading



## Output

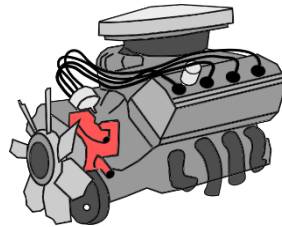
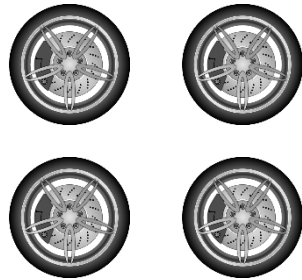
```
Student Name is: New Student
Student Age is: 18
Student ID is: 100
Student Name is: Chaitanya
Student Age is: 25
Student ID is: 555
```

- Notes regarding class `StudentData`'s *set* and *get* methods
  - Classes often provide public methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) private instance variables.
  - *Set* methods are also commonly called **mutator methods**, because they typically change an object's state—i.e., modify the values of instance variables.
  - *Set* methods must include *Validity and Checking before modifying any* variables of the class.
  - *Get* methods are also commonly called **accessor methods** or **query methods**.

# Composition



- A class can have references to objects of other classes as members.
- This is called **composition** and is sometimes referred to as a **has-a relationship**.
- Example: An Car object needs to have an engine and wheels, so it's reasonable to include one references to engine object and four references for wheels objects.



# Composition Example (1/2)

```
public class Engine {  
    private int engineCapacity;  
    private int engineSerialNumber;
```

Class

```
    public Engine(int engineCapacity, int engineSerialNumber) {  
        this.engineCapacity = engineCapacity;  
        this.engineSerialNumber = engineSerialNumber;  
    }
```

Constructor

```
    public int getEngineCapacity() {  
        return engineCapacity;  
    }
```

```
    public int getEngineSerialNumber() {  
        return engineSerialNumber;  
    }  
}
```

```
class Car {  
    private String make;  
    private int year;  
    private Engine engine;
```

Class

Reference from  
class Engine

```
    public Car(String make, int year, int engineCapacity, int  
engineSerialNumber) {  
        this.make=make;  
        this.year=year;  
        engine = new Engine(engineCapacity, engineSerialNumber);
```

Constructor

Initializing  
Engine object

```
    }  
    public String getMake() {  
        return make;  
    }  
    public int getYear() {  
        return year;  
    }  
    public int getEngineSerialNumber() {  
        return engine.getEngineSerialNumber();  
    }  
    public int getEngineCapacity() {  
        return engine.getEngineCapacity();  
    }  
}
```



# Composition Example (2/2)

```
class TestCar
{
    public static void main(String args[])
    {
        Car car = new Car("BMW",2014,2500,235215363);
        System.out.println("This car model is "+ car.getMake());
        System.out.println("with a Production Year "+ car.getYear());
        System.out.println("This car engine Power is "+ car.getEngineCapacity());
        System.out.println("This car engine Serial Number is "+ car.getEngineSerialNumber());
    }
}
```

## Output

```
This car model is BMW
with a Production Year 2014
This car engine Power is 2500
This car engine Serial Number is 235215363
```

# static Class Member Method



- A **static method** **cannot** access **non-static class members**, because a static method can be called even when no objects of the class have been instantiated.
  - static method -> static field  
static은 특정한 객체에서만 접근할 수 있음  
non static에는 접근불가
- For the same reason, the this reference cannot be used in a static method.
- The **this** reference must refer to a **specific object** of the class, and when a static method is called, there might **not be any objects of its class in memory.**
- If a static variable is not initialized, the compiler assigns it a default value—in this case 0, the default value for type `int`.

# Example: static Class Members (1/2)

```
class Student{

    int registrationNumber;

    String name;

    static String college = "IT&Engineering";

    Student(int num, String nam){

        registrationNumber = num;

        name = nam;

    }

    void printInformation(){

        System.out.println("This is "+this.name+",
        with ID #"+ this. registrationNumber +", and
        collage name: "+ this.college);

    }

    static void change () {

        // this.name = "Math" error: cannot use this

        college = "Engineering";

    }

    static void print(){

        System.out.println(college);

    }

}
```

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

{

    Student.print();

    Student.change(); class의 모든 college를 change

    Student tome = new Student(152442, "Tome");

    tome.printInformation();

}

}
```

## Output

```
IT&Engineering
This is Tome, with ID #152442, and
collage name: Engineering
```

Student a = new Student();  
Student.print(); 가능  
a가 아닐경우도 가능  
객체 선언 안해도 non static method 호출 가능  
but, 객체가 안만들어짐  
but method안에 static 변수 들어가면 X  
생성자로 초기화 안되어있기 때문에

# static Class Member Variables



- A *static variable* is a variable that belongs to the class as a whole, and not just to one object
  - There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
- All objects of the class can read and change a static variable
- Although a static method cannot access an instance variable, a static method can access a static variable
- A static variable is declared like an instance variable, with the addition of the modifier **static**

```
private static int myStaticVariable;
```

static은 하나의 메모리를 공유

private 변수라도 class안에 main method가 있으면 접근 가능  
if main method가 밖의 클래스에 있으면, private 변수 접근 불가

# Static Variables



- Static variables can be declared and initialized at the same time

```
private static int myStaticVariable = 0;
```

- If not explicitly initialized, a static variable will be automatically initialized to a default value
  - **boolean** static variables are initialized to **false**
  - Other primitive types static variables are initialized to the zero of their type
  - Class type static variables are initialized to **null**
- It is always preferable to explicitly initialize static variables rather than rely on the default initialization

# Example: static Class Members (2/2)

```
class Static {  
  
    static int staticCount ;  
  
    private int NonStaticCount = 0;  
  
    public Static(){  
        NonStaticCount = NonStaticCount+1;  
    }  
  
    public int getCount() {  
        return NonStaticCount;  
    }  
  
    public void setCount(int count1) {  
        this.NonStaticCount = NonStaticCount;  
    }  
  
    public static int countStaticPosition() {  
        staticCount = staticCount+1;  
        return staticCount;  
    }  
}
```

```
public static void main(String args[])  
  
    {  
        Static p = new Static();  
  
        System.out.println("static count position is " +  
            Static.staticCount);  
  
        System.out.println("count position is " +  
            p.getCount());  
  
        System.out.println("static count position is " +  
            Static.countStaticPosition());  
    }  
}
```

## Output

```
static count position is 0  
count position is 1  
static count position is 1
```



# static Import

- A `static import` declaration enables you to import the `static` members of a class or interface so you can access them via their unqualified names in your class—the class name and a dot ( `.` ) are not required to use an imported `static` member.
- Two forms
  - One that imports a particular `static` member (which is known as `single static import`)
  - One that imports all `static` members of a class (which is known as `static import on demand`)
- The following syntax imports a particular `static` member:  
`import static packageName.ClassName.staticMemberName;`
- The following syntax imports all `static` members of a class:  
`import static packageName.ClassName. *;`



# The Math Class

- The **Math** class provides a number of standard mathematical methods
  - It is found in the **java.lang** package, so it does not require an **import** statement
  - All of its methods and data are static, therefore they are invoked with the class name **Math** instead of a calling object
  - The **Math** class has two predefined constants, **E** ( $e$ , the base of the natural logarithm system) and **PI** ( $\pi$ , 3.1415 . . . )  

```
area = Math.PI * radius * radius;
```





# Example: static Import

```
// Static import of Math class methods.  
import static java.lang.Math.*;  
  
public class StaticImportTest  
{  
    public static void main( String[] args )  
    {  
        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );  
        System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );  
        System.out.printf( "E = %f\n", E );  
        System.out.printf( "PI = %f\n", PI );  
    } // end main  
} // end class StaticImportTest
```

```
sqrt( 900.0 ) = 30.0  
ceil( -9.8 ) = -9.0  
log( E ) = 1.0  
cos( 0.0 ) = 1.0
```

# Creating Packages



- Each class in the Java API belongs to a package that contains a **group of related classes**.
- Packages are defined once, but can be imported into many programs.
- Packages help programmers **manage the complexity** of application components.
- Packages facilitate **software reuse** by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps **prevent class-name conflicts**.



# The `package` Statement

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each class file:

```
package package_name;
```

- Only the `.class` files must be in the directory or folder, the `.java` files are optional
- Only blank lines and comments may precede the package statement
- If there are both import and package statements, the package statement must precede any import statements

# The Package `java.lang`



- The package `java.lang` contains the classes that are fundamental to Java programming
  - It is imported automatically, so no import statement is needed
  - Classes made available by `java.lang` include `Math`, `String`, and the wrapper classes

# Creating Packages

```
package com.companyName.departmentName.carProject;
```

```
public class Engine {  
    private int engineCapacity;  
    private int engineSerialNumber;
```

Package name



```
    public Engine(int engineCapacity, int engineSerialNumber) {  
        this.engineCapacity = engineCapacity;  
        this.engineSerialNumber = engineSerialNumber;  
    }
```

```
    public int getEngineCapacity() {  
        return engineCapacity;  
    }
```

```
    public int getEngineSerialNumber() {  
        return engineSerialNumber;  
    }  
}
```

```
package com.companyName.departmentName.carProject;
```

```
class Car {  
    private String make;  
    private int year;  
    private Engine engine;
```

Package name



```
    public Car(String make, int year, int engineCapacity, int  
        engineSerialNumber) {  
        this.make=make;  
        this.year=year;  
        engine = new Engine(engineCapacity, engineSerialNumber);
```

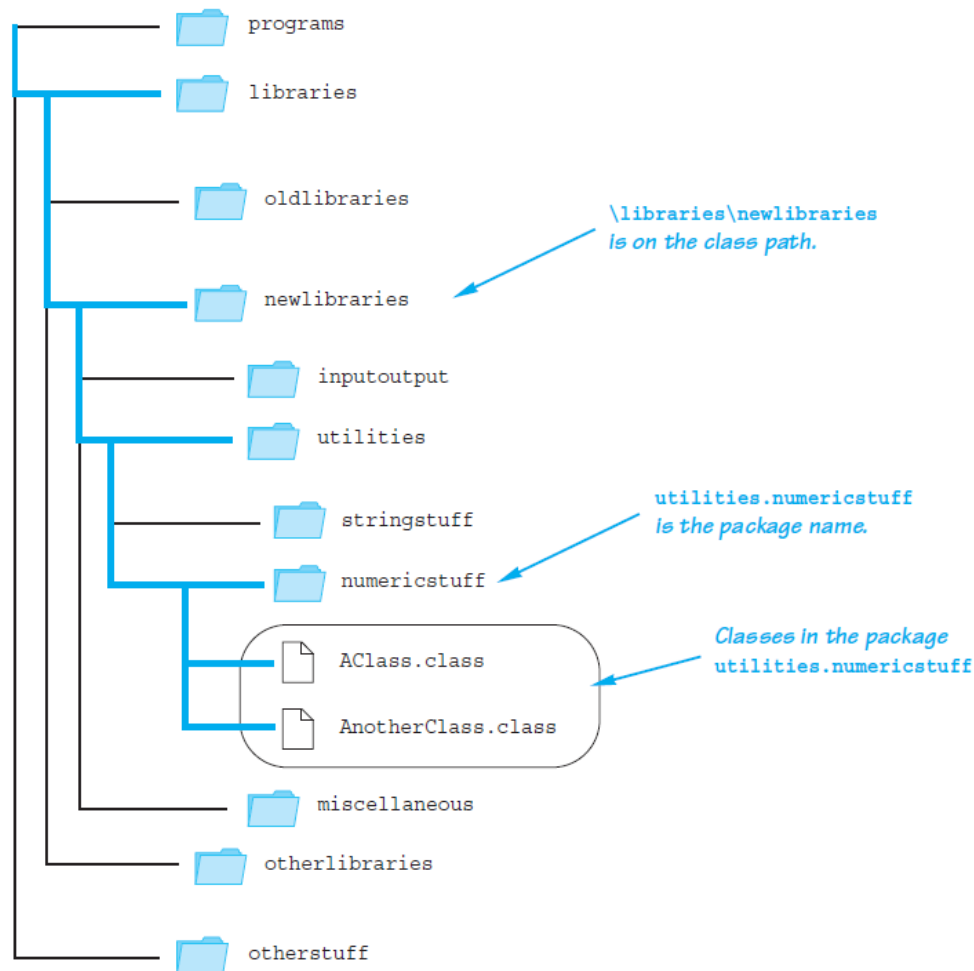
```
    }  
    public String getMake() {  
        return make;  
    }  
    public int getYear() {  
        return year;  
    }  
    public int getEngineSerialNumber() {  
        return engine.getEngineSerialNumber();  
    }  
    public int getEngineCapacity() {  
        return engine.getEngineCapacity();  
    }  
}
```

# Package Names and Directories



- A package name is the path name for the directory or subdirectories that contain the package classes
- Java needs two things to find the directory for a package: the name of the package and the value of the **CLASSPATH** variable
  - The **CLASSPATH** environment variable is similar to the **PATH** variable, and is set in the same way for a given operating system
  - The **CLASSPATH** variable is set equal to the list of directories (including the current directory, ".") in which Java will look for packages on a particular computer
  - Java searches this list of directories in order, and uses the first directory on the list in which the package is found

# A Package Name



# Subdirectories Are Not Automatically Imported



- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- The import statement:  
`import utilities.numericstuff.*;`  
imports the `utilities.numericstuff` package only
- The import statements:  
`import utilities.numericstuff.*;`  
`import utilities.numericstuff.statistical.*;`  
import both the `utilities.numericstuff` and `utilities.numericstuff.statistical` packages





# The Default Package

- All the classes in the current directory belong to an unnamed package called the *default package*
- As long as the current directory ( `.` ) is part of the **CLASSPATH** variable, all the classes in the default package are automatically available to a program

# Not Including the Current Directory in Your Class Path



- If the **CLASSPATH** variable is set, the current directory must be included as one of the alternatives
  - Otherwise, Java may not even be able to find the **.class** files for the program itself
- If the **CLASSPATH** variable is not set, then all the class files for a program must be put in the current directory

# Specifying a Class Path When You Compile



- The class path can be manually specified when a class is compiled
  - Just add `-classpath` followed by the desired class path
  - This will compile the class, overriding any previous `CLASSPATH` setting
- You should use the `-classpath` option again when the class is run

```
C:> set CLASSPATH= java Directory
```

For example, suppose you want the Java runtime to find a class named `Cool.class` in the package `utility.myapp`. If the path to that directory is `C:\java\MyClasses\utility\myapp`, you would set the class path so that it contains `C:\java\MyClasses`.

To run that app, you could use the following JVM command:

```
C:> java -classpath C:\java\MyClasses utility.myapp.Cool
```

# Name Clashes



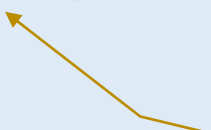
- In addition to keeping class libraries organized, packages provide a way to deal with *name clashes*: a situation in which two classes have the same name
  - Different programmers writing different packages may use the same name for one or more of their classes
  - This ambiguity can be resolved by using the *fully qualified name* (i.e., precede the class name by its package name) to distinguish between each class

**`package_name.ClassName`**
- If the fully qualified name is used, it is no longer necessary to import the class (because it includes the package name already)

# Access Packages

```
import com.companyName.departmentName.carProject.*;
class TestCar
{
    public static void main(String args[])
    {
        Car car1 = new Car("BMW",2014,2500,235215363);
        com.companyName.departmentName.carProject Car2 = new Car("KIA",2014,2200,54561444561);

        System.out.println("This car model is "+ car1.getMake());
        System.out.println("with a Production Year "+ car1.getYear());
        System.out.println("This car engine Power is "+ car1.getEngineCapacity());
        System.out.println("This car engine Serial Number is "+ car1.getEngineSerialNumber());
    }
}
```



Access Package

## Output

```
This car model is BMW
with a Production Year 2014
This car engine Power is 2500
This car engine Serial Number is 235215363
```

# Summary



- Class Constructor
- Overloaded Constructors
- Class *Set* and *Get* Methods
- Class Composition
- Class static members
  - Creating static variables and methods.
  - Importing static variables and methods from other class.
- Creating Packages
- Access Packages