

ET0736

Lesson 3

Class and Object

Topics

- Defining a class and creating Object
- Constructor
- ***this*** keyword
- Array of Objects and sorting with comparators
- Passing Objects to methods
- More on String class
- Static variable and method
- Getter and Setter methods
- Package in Java
- Java Wrapper class

Class

Object-oriented programming (OOP) languages are designed to overcome problems in using procedural programming.

- 1) The basic unit of OOP is a **class**.
- 2) A class has a **class name**.
- 3) A class has **data fields** (or some texts refer as **attributes** or **properties**)
- 4) A class has **methods** (or some texts refer as behaviours or actions).
- 5) A **class** encapsulates (or hides) both the **data fields** and **methods**.
- 6) The difference between **data fields** and **method** is:
 - **Data fields** are for storing data
 - **Methods**, on the other hand, contains statements and code to perform certain actions
- 7) Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes.

Class

We have used classes before, but mainly provided by JDK.

Class	Example of method	Example of data field
Scanner	Scanner.nextInt()	--
String	String.length()	--
Math	Math.random()	Math.PI



ゆりこ Yuriko
1923232
3.97



あおやま Aoyama
1919191
3.88

```
class SpStudent
{
    String name;
    String ID;
    double GPA;
}
```



なおき Naoki
1908080
4.0

Define a class

Where to define a custom class?

```
public class Project {  
    public static void main(String[] args) {  
  
    }  
} // end of project class  
  
// start your custom class from here  
  
class SpStudent  
{  
    String name;  
    String ID;  
    double GPA;  
}
```

SpStudent
String name; String ID; double GPA;
<no method yet>

Create Object and initialise attributes

```
public class Project {  
    public static void main(String[] args) {  
        SpStudent s1 = new SpStudent();  
        s1.name="Aoyama";  
        s1.ID = "1919191";  
        s1.GPA = 3.88;  
    }  
}  
  
class SpStudent  
{  
    String name;  
    String ID;  
    double GPA;  
}
```

s1 is an
instance/object of
SpStudent class

Class with attributes and method

```

public class Project {
    public static void main(String[] args) {
        SpStudent s1 = new SpStudent();
        s1.name="Aoyama";
        s1.ID = "1919191";
        s1.updateGPA(1, 3.88);
        System.out.println ("GPA after year 1 : "+s1.GPA);
        s1.updateGPA(2, 3.4);
        System.out.println ("GPA after year 2 : "+s1.GPA);
        s1.updateGPA(3, 4.0);
        System.out.println ("GPA after year 3 : "+s1.GPA);
    }
}

class SpStudent
{
    String name;
    String ID;
    double GPA=0.0;

    void updateGPA(int currentYear, double gpa){
        switch (currentYear) {
            case 1: GPA = gpa; break;
            case 2 : GPA = (GPA +gpa)/2; break;
            case 3: GPA = (GPA*2/3) + (gpa/3);
        }
    }
}

```

SpStudent
String name; String ID; double GPA;
void updateGPA(int currentYear, double gpa)

Class constructor

A **constructor** :

- is a special kind of **method**
- is defined inside the class itself
- must have the same name as the **class**.
- do not have a return type – not even **void**.
- called (invoked) automatically when an object is created via the **new** keyword.
- is usually used to 'automate' the initialising of variables of the object

Class with Constructor

```
public class Project {
    public static void main(String[] args) {
        SpStudent s1 = new SpStudent("Aoyama", "1919191");
        s1.updateGPA(1, 3.88);
        System.out.println ("GPA after year 1 : "+s1.GPA);
        s1.updateGPA(2, 3.4);
        System.out.println ("GPA after year 2 : "+s1.GPA);
        s1.updateGPA(3, 4.0);
        System.out.println ("GPA after year 3 : "+s1.GPA);
    } }
```

```
class SpStudent
{
```

```
    String name;
    String ID;
    double GPA;
```

```
    SpStudent (String n, String id) {
        GPA = 0.0;
        name = n;
        ID = id;
    }
```

```
    void updateGPA(int currentYear, double gpa){
        switch (currentYear) {
            case 1: GPA = gpa; break;
            case 2 : GPA = (GPA +gpa)/2; break;
            case 3: GPA = (GPA*2/3) + (gpa/3);
        } } }
```

s1 is created with the newly added constructor

SpStudent constructor

Another example

```
public class TestBank {  
  
    public static void main(String[] args) {  
  
        BankAC b1 = new BankAC("112233", "Jane Fonda", 1000.34);  
  
        b1.deposit(100);  
        System.out.println (b1.balance);  
    }  
}  
  
class BankAC {  
  
    String acNo;  
    String name;  
    double balance;  
  
    public BankAC(String ac, String n, double b){  
        acNo = ac;  
        name = n;  
        balance = b;  
    }  
  
    public void deposit(double amt){  
        balance += amt;  
    }  
  
    public boolean withdraw(double amt){  
        if (balance >= amt){  
            balance -= amt;  
            return true;  
        }  
        else return false;  
    }  
}  
} // end class
```

Overloading Constructors

```
class Circle {  
    private double radius;  
  
    public Circle(){                // constructor 1  
        radius = 5.0;  
    }  
    public Circle(double radius){  // constructor 2  
        this.radius = radius;  
    }  
  
    public double findArea(){  
        return radius * radius * 3.14159;  
    }  
    public double getRadius() {  
        return radius;  
    }  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
} // end class Circle  
  
public class TestCircle {  
    public static void main(String[] args) {  
        Circle c1 = new Circle (17.5);  
        Circle c2 = new Circle (); // default radius = 5.0  
    }  
}
```

overloading
constructors

(as long as the signatures
are unique)

this keyword

If parameter names are the same as the attribute names, precede attribute names with “**this.**” , to differentiate the internal attributes and the incoming input parameters.

```
class BankAC {  
  
    String acNo;  
    String name;  
    double balance;  
  
    public BankAC(String acNo, String name, double balance){  
  
        this.acNo = acNo;  
        this.name = name;  
        this.balance = balance;  
  
    }  
} // end class
```

Array of Objects

Example:

```
Car[] cars = {  
    new Car ("SGX1234","Toyota",4.3),  
    new Car ("SNG8888","Honda",1.9)  
};
```

```
class Car {  
    String license;  
    String brand;  
    double age;  
    public Car(String license, String  
brand, double age) {  
        this.license = license;  
        this.brand = brand;  
        this.age = age;  
    }  
}
```

Sorting Array of Objects

Using

Arrays.sort(*array*, *comparator*)

and needs a helper class implementing **Comparator** interface

```
class SortByBrand implements Comparator<Car> {  
    @Override  
    public int compare(Car a, Car b) {  
  
        // sort by car brand  
  
        return (int)(a.brand.compareTo(b.brand));  
    }  
}
```

Sorting Array of Objects

```
class SortByBrand implements Comparator<Car> {  
  
    @Override  
    public int compare(Car a, Car b) {  
        // sort by car brand  
        return (int)(a.brand.compareTo(b.brand));  
    }  
}
```

< E > indicates that the container is a generic class and is able to hold different types other than **Car** object

Mandatory implementation of method **compare(E, E)**

The **compareTo()** is a String class method, which compares two strings lexicographically. It returns a *negative* if the string < the other string. (0 for equal or positive for >).

Sorting Array of Objects

Sorting **Car** objects by **brands**:

```
Car[] cars = {  
    new Car ("SGX1234","Toyota",4.3),  
    new Car ("SNG8888","Honda",1.9)  
};
```

```
Arrays.sort (cars, new SortByBrand());
```

The **Arrays.sort()** uses the returned results from **SortByBrand()** to determine if the objects should be swapped.

```
class SortByBrand implements Comparator<Car> {  
    @Override  
    public int compare(Car a, Car b) {  
        return (int)(a.brand.compareTo(b.brand));  
    }  
}
```

```
class Car {  
    String license;  
    String brand;  
    double age;  
    public Car(String license, String brand,  
        double age) {  
        this.license = license;  
        this.brand = brand;  
        this.age = age;  
    }  
}
```

Sorting Array of Objects

Sorting **Car** objects by **age**:

```
Car[] cars = {  
    new Car ("SGX1234","Toyota",4.3),  
    new Car ("SNG8888","Honda",1.9)  
};
```

```
Arrays.sort (cars, new SortByAge());
```

```
class SortByAge implements Comparator<Car> {  
    @Override  
    public int compare(Car a, Car b) {  
        return (int)(a.age - b.age);  
    }  
}
```

```
class Car {  
    String license;  
    String brand;  
    double age;  
    public Car(String license, String brand,  
        double age) {  
        this.license = license;  
        this.brand = brand;  
        this.age = age;  
    }  
}
```

Object Reference

- **Object types** variables are not **primitive types** of variables.
- The **name** of an object variable represents the '**reference**' to that object.

```
c1    // refers to memory location site storing object c1  
c2    // refers to memory location site storing object c2
```

- The statement below merely makes c2 points to the site storage of object c1. It is not a **copy** operation from one object to another. Object previously referenced by c2 is no longer accessible. It is now referred to as **garbage**

```
c2 = c1
```

- To clone or create the exact same copy of an object with all its fields initialised with the contents of the corresponding fields of this object, the class must implement **java.lang.Cloneable** interface, which will be discussed in later lesson.

Passing Object to Method

Objects are passed by reference into methods.

```
class Animal {
    int LegCount;
    int age;
    Animal(int LC, int a) {
        LegCount=LC; age=a;
    }
}

public class TestPassByRef {
    public static void main(String[] args) {
        Animal caterpillar = new Animal(16,1);
        chopLeg (caterpillar,2);
        System.out.println ("legs left:"+ caterpillar.LegCount );
    }

    static void chopLeg( Animal x, int y) {
        x.LegCount= x.LegCount-y;
    }
}
```

More on String class

String is not a primitive data type but rather an object.

Hence, String is passed into method by reference.

```
class TestString {  
    public static void main(String[] args) {  
        String s = "I like orange";  
        ABC(s);  
        System.out.println ("After ABC() called, s = " +s);  
    }  
  
    public static void ABC(String a) {  
        a += " and apple";    // new string object is created as old literal is immutable  
        System.out.println ("Inside ABC() , s = " +a);  
    }  
}
```

Output:

```
Inside ABC() , s = I like orange and apple  
After ABC() called, s = I like orange
```

Although the string is passed by reference, the original String object remains unchanged,.

More on String class

To modify the original String object, use mutable classes like ***StringBuilder*** or ***StringBuffer*** instead

```
class TestString {  
    public static void main(String[] args) {  
        StringBuilder s = new StringBuilder ("I like orange" );  
        ABC(s);  
        System.out.println ("After ABC() called, s = " +s);  
    }  
  
    public static void ABC(StringBuilder a) {  
        a += " and apple";    // new string object is created as old literal is immutable  
        System.out.println ("Inside ABC() , s = " +a);  
    }  
}
```

Output:

```
Inside ABC() , s = I like orange and apple  
After ABC() called, s = I like orange and apple
```

Static variable

- is attached to a ***class***
- *shared* by all instances (objects) of the ***class***
- has only '*one copy*' of the variable.
- is declared with the ***static*** keyword (see code below)
- can be assessed by ***Class_name.variable_name*** (e.g. Circle.count)
- can also be assessed by ***Object_name.variable_name*** (e.g. x.count, z.count etc)

Static variable

```
class Circle {
    String Color;
    double Radius;

    static int count=0;    // class variable

    Circle (String c, double r){
        Color = c; Radius = r;
        count++;
    }
}

public class TestClassVariable {
    public static void main(String[] args) {
        Circle x = new Circle("red",1.0);
        System.out.println ("count=" + Circle.count);
        Circle y = new Circle("red",5.0);
        Circle z = new Circle("blue",10.0);
        System.out.println ("count=" + y.count);
    }
}
```

The variable **count** is defined as **static**.

The 3 objects **x**, **y** and **z** (of class **Circle**) share the same common copy of **count**.

Output:
count=1
count=3

Static method

- is attached to a class (not any specific object)
- is similar to class variable, a ***static*** modifier is added in front of a method
- can be assessed directly by ***Class_name.method_name***
- can also be assessed by ***Object_name.method_name***
- can call only other static methods and static variables defined the class
- If you try to use a non-static method and variable defined in this class then the compiler will warn that non-static variable or method cannot be referenced from a static context.

Static method

```
class SPApartmentRental {  
    private double rental;  
    private static int UnitsLeft=10;  
  
    SPApartmentRental(double r) {  
        UnitsLeft--;  
        rental = r;  
    }  
  
    static int getUnitsLeft() {  
        return UnitsLeft;  
    }  
}  
  
public class TestStaticMethod {  
    public static void main(String[] args) {  
        System.out.println(SPApartmentRental.getUnitsLeft());  
        SPApartmentRental s1 = new SPApartmentRental(1800.50);  
        System.out.println(s1.getUnitsLeft());  
    }  
}
```

Output:

10

9

Encapsulation

The 4 characteristics commonly associated with OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation is about hiding implementation details of object within itself and only allow access to necessary data/method from outside the class. It can be done by :

- Employing various access modifiers
- Organising programs in packages

Access Modifiers – public vs Private

public - The variable or method in the class is accessible and available to ALL the other objects in the entire program.

private - The variable or method in the class is accessible and available within this class only. Indirect access to ***private*** attributes in a class (from outside) can be achieved by special methods called ***setter*** and ***getter*** methods provided by the class.

UML convention

- + **public**
- **private**
- # **protected**
- ~ **default**

Getter method

```
class SpStudent
{
    String name;
    String ID;
    private double GPA=0.0;

    public void updateGPA(int currentYear, double gpa){
        switch (currentYear) {
            case 1: GPA = gpa; break;
            case 2 : GPA = (GPA +gpa)/2; break;
            case 3: GPA = (GPA*2/3) + (gpa/3);
        }
    }

    public double getGPA() { return(GPA); }
}
```

radius is private

**Getter method
for GPA**

```
public class Project {
    public static void main(String[] args) {
        SpStudent s1 = new SpStudent();
        s1.name="Aoyama";
        s1.ID = "1919191";
        s1.updateGPA(1, 3.88);
        System.out.println ("GPA after year 1 : " + s1.getGPA());
    }
}
```

Changes in main()

Getter and Setter Methods

```
class Circle
{
    private double radius;

    public Circle() { radius=7.0; }
    public Circle(double r) { radius=r; }

    Public double getRadius() { return radius; }
    public setRadius(double r) { radius = r; }

    public double getArea(){
        return radius * radius * 22.0/7.0;
    }
}
```

**Getter and setter
methods for
radius**

```
public class Project {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        c1.setRadius(10);
        System.out.println ("Area of circle with radius " +
            c1.getRadius() + " is " + c1.getAread());
    }
}
```

Package in Java

Encapsulation can also be achieved by using packages in Java.

Package is used for better organising the various classes in a program. With package, better accessibility control can also be implemented.

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

* Default – no modifier

UML convention

+ **public**
- **private**
protected
~ **default**

Access Modifiers

To set up the example:

- Create a new **Java Project** in IDE.
- Before creating any classes, create 2 new **Packages** (right-click at **src**).

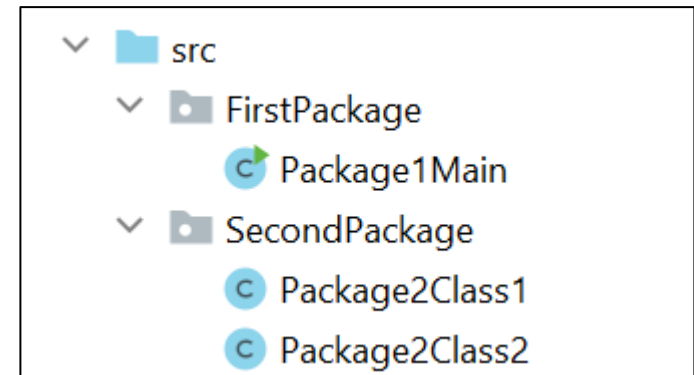
FirstPackage

SecondPackage

- Create a class with **main()** in the **FirstPackage**.
- Create two separate classes under **SecondPackage**:

Package2Class1

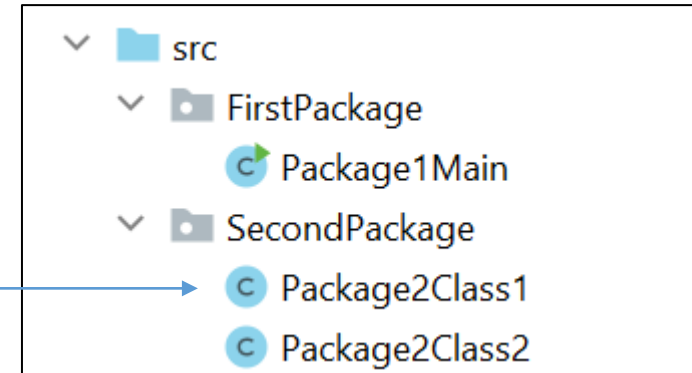
Package2Class2



Access Modifiers

- Begin the coding in class ***Package2Class1***.
- Create 4 variables with different access level (i.e. ***a1***, ***a2***, ***a3*** and ***a4***).

```
package SecondPackage;  
public class Package2Class1  
{  
    public int a1;  
    protected int a2;  
    int a3;  
    private int a4;  
}
```



Access Modifiers

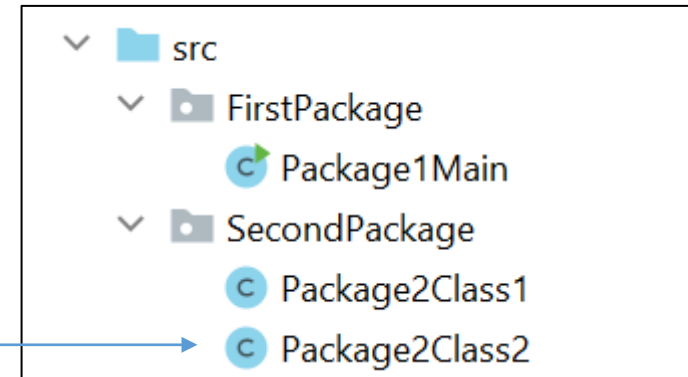
- Next, in class **Package2Class2**, set up some code to experiment the access to the variables data of **Package2Class1** object

```
package SecondPackage;
public class Package2Class2 {
    Package2Class1 x = new
    Package2Class1();

    public void TryToReach() {
        x.
    }
}
```

Autocomplete suggestions for `x.`:

- f a1
- f a2
- f a3
- m getClass()
- try



At this point, conclusion about the 2 classes within the **same package** can be drawn. Only **private** variable **a4** is not accessible.

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

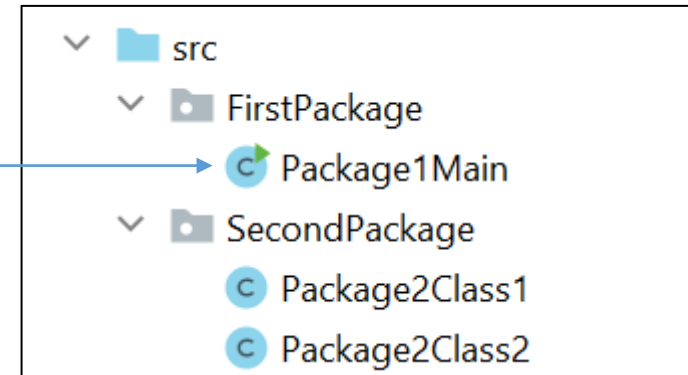
Access Modifiers

- Next, code the ***main()***, which is in a different package.
- Set up some code to experiment the access to the variables of ***Package2Class1*** object.

```
package FirstPackage;
import SecondPackage.Package2Class1;
public class Package1Main {
    public static void main(String[]
args) {
        Package2Class1 x = new
Package2Class1();
        x.
    }
}
```

f a1

- m** equals(Object obj)
- m** hashCode()
- m** toString()



main() can only access the ***public*** variable (i.e. ***a1***) defined in another package

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Wrapper Classes in Java

- The ***wrapper class*** in Java provides the mechanism to convert ***primitive*** into ***object*** and ***object*** into ***primitive***.
- All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The conversion of a primitive type into an object is known as **autoboxing** and vice-versa **unboxing**.

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

    byte b=14;
    short s=55;
    int i=666;
    long l=77777;
    float f=88.88F;
    double d=99.90D;
    char c='z';
    boolean bb=true;
```

```
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=bb;
```

Since J2SE 5.0, conversion
to wrapper class is
automatically done

Cont...

```
//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);
}
}
```

Autounboxing

Since J2SE 5.0, conversion
to primitive is
automatically done

Cont...

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

        byte b=14;
        short s=55;
        int i=666;
        long l=77777;
        float f=88.88F;
        double d=99.90D;
        char c='z';
        boolean bb=true;

        //Autoboxing: Converting
        // primitives into objects
        Byte byteobj=b;
        Short shortobj=s;
        Integer intobj=i;
        Long longobj=l;
        Float floatobj=f;
        Double doubleobj=d;
        Character charobj=c;
        Boolean boolobj=bb;
    }
}
```

//Unboxing: Converting Objects to Primitives

```
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;
```

//Printing primitives

```
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);

}
}
```

Integer (wrapper class)

- The ***Integer*** class wraps a value of the primitive type ***int*** in an ***object***.
- An object of type ***Integer*** contains a single field whose type is ***int***.
- In addition, this class provides ***several methods*** for converting an ***int*** to a ***String*** and a ***String*** to an ***int***, as well as other constants and methods useful when dealing with an int.
- Some of the methods of Integer:

Modifier and Type	Method	Description
int	<code>compareTo(Integer anotherInteger)</code>	Compares two Integer objects numerically.
static int	<code>parseInt(String s)</code>	Parses the string argument as a signed decimal integer.
static int	<code>parseInt(String s, int radix)</code>	Parses the string argument as a signed integer in the radix specified by the second argument.
String	<code>toString()</code>	Returns a String object representing this Integer's value.
static Integer	<code>valueOf(int i)</code>	Returns an Integer instance representing the specified int value.
static Integer	<code>valueOf(String s)</code>	Returns an Integer object holding the value of the specified String.

Integer (wrapper class)

Covert String to int

```
public class Test {  
    public static void main(String[] args) {  
        String s ="33";  
        int i = Integer.parseInt(s);  
        i++;  
        System.out.println(i);    // 34  
    }  
}
```

Covert String to Integer

```
public class Test {  
    public static void main(String[] args) {  
        String s ="33";  
        Integer i = Integer.valueOf(s);  
        i++;  
        System.out.println(i);    // 34  
    }  
}
```


“has-a” Relationships

- Two objects is related by “has-a” relationship
- It is also called composition
- Example below: a Traveller object “has a” Car object

```
public class Test {
    public static void main(String[] args) {
        Traveller a = new Traveller();
        a.travel("AMK");
    }
}
```

```
class Traveller {
    Car myCar();
    Traveller () {
        myCar = new Car("SHA1234");
    }

    public void travel(String dest) {
        myCar.move(dest);
    }
}
```

This implementation causes a close coupling between Traveller and Car.

Any changes made in the constructor of class Car will need to update the class Traveller as well.

```
class Car {
    String licensePlate;
    Car(String n) {
        licensePlate = n;
    }

    public void move(String dest) {
        System.out.println ("Going to " +
            dest + "by car");
    }
}
```

Dependency Inversion Principle (DIP)

- Dependency refers to the coupling between the different classes
- The main idea is that the high level classes should rely on **abstractions** rather than **concrete implementations** of the lower classes.
- By following DIP, classes are loosely coupled
- Dependency inversion makes code less coupled, more maintainable, more reusable and easier to test.
- Two ways of implementing DIP
 - By setter method
 - By constructor

DIP by Setter method

```
public static void main(String[] args) {
    Car theCar = new Car("SHA1234");
    Traveller a = new Traveller();
    a.setCar(theCar);
    a.travel("AMK");
}
```

Creation of Car object is outside the Traveller class
This decouples the lifecycle of Car objects from the Traveller class

```
class Traveller {
    Car myCar;
    public void setCar(Car c) {
        myCar = c;
    }
    public void travel(String dest) {
        myCar.move(dest);
    }
}
```

Car object is passed in via setter
Any changes made to the constructor of class Car will not affect the Class Traveller

```
class Car {
    String licensePlate;
    Car(String n) {
        licensePlate = n;
    }


    public void move(String dest) {
        System.out.println ("Going to " + dest + "by car");
    }
}
```

(no change)

By Constructor

```
public static void main(String[] args) {
    Car theCar = new Car("SHA1234");
    Traveller a = new Traveller(theCar);
    a.travel("AMK");
}
```

Creation of Car object is still outside the Traveller class



```
class Traveller {
    Car myCar;

    Traveller(Car c) {
        myCar = c;
    }
    Public void travel(String dest) {
        myCar.move(dest);
    }
}
```

Car object is passed in via constructor



```
class Car {
    String licensePlate;
    Car(String n) {
        licensePlate = n;
    }

    public void move(String dest) {
        System.out.println ("Going to " + dest + "by car");
    }
}
```

(no change)