

# **Inheritance And Polymorphism II**

# Instance and Class Variables

- Objects created from the same class , have their own distinct copies of *instance variables*. Each object has its own values for these variables, stored in different memory locations.
- Sometimes, we need to have variables that are common to all objects – variables associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.
- We use the *static* keyword to create fields and methods that belong to the class, rather than to an instance of the class.

# Instance and Class Methods

- The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, **without the need for creating an instance of the class**.
- A common use for static methods is to access static fields.
- ✓ Instance methods can access instance variables and instance methods **directly**.
- ✓ Instance methods can access class variables and class methods **directly**.
- ✓ Class methods can access class variables and class methods directly.
- ✓ Class methods **cannot access instance** variables or instance methods directly-they **must use an object reference**.
- ✓ Also, class methods **cannot use the “this”** keyword as there is no instance for this to refer to.

# Static Variable

- Also known as class variables
- It is a variable which belongs to the class and not to object (instance)
- These variables will be initialized first, before the initialization of any instance variables
- A single copy to be shared by all instances (Objects) of the class
- A static variable can be accessed directly by the class name and doesn't need any object .
- Class/static variables are declared using the static keyword in a class.
- These are declared outside a class and stored in static memory.
- Class variables are mostly used for constants.
- This variable is created when the program starts and gets destroyed when the programs stops.

# Static Method

- Also known as class methods.
- Belongs to the class rather than any object of the class.
- Does not require that any objects of the class have been instantiated.
- Static methods are implicitly final.
- A static method in a superclass **can be shadowed** by another static method in a subclass, as long as the original method was not declared final.
- Override a static method with a non-static method is **not allowed**.
- A static method **can not be changed** by an instance method in a subclass.
- Can **access only** static data. It **can not access** instance variables
- Can **call only** other static methods and **can't** call a non-static method from it
- A static method **can't refer to "this" or "super"** keywords in anyway
- main method is static , since it must be accessible for an application to run, before any instantiation takes place.

## Benefits :

- ✓ Documentation: makes reading and debugging easier.
- ✓ Efficiency: A compiler will usually produce slightly more efficient code because no implicit object parameter has to be passed to the method.

## המטרה : לדעת מהו מספר האובייקטים שנוצר.

תוכנית לא נכונה  
לבדיקת מספר האובייקטים הנוצרים

```
public class Person {  
    private String name;  
    public int counter;  
  
    public Person(String name) {  
        this.name=name;  
        counter++;  
    }  
  
    public String toString() {  
        return counter + ":" + name;  
    }  
}
```

Non-static variable counter can't be referenced  
from a static context

```
public class Tester {  
    public static void main(String[ ] args) {  
        Person p1=new Person("a");  
        System.out.println(p1); //1:a  
        Person p2=new Person("b");  
        System.out.println(p2); //1:b  
        Person p3=new Person("c");  
        System.out.println(p3); //1:c  
        System.out.println(Person.counter);  
    }  
}
```

פעולה לא חוקית לא ניתן  
לגשת למשתנה מופע  
ישירות (לא דרך  
אובייקט)

משתנים סטטיים מקבלים  
אתחול ברירת מחדל כמו משתני  
מחלקה לכן counter=0

המטרה : לדעת מהו מספר  
האובייקטים שנוצרו.

תוכנית נכונה  
לבדיקת מספר האובייקטים הנוצרים

```
public class Person{  
    private String name;  
    public static int counter;  
  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
  
    public String toString(){  
        return counter+": "+name;  
    }  
}
```


```
public class Tester {  
    public static void main(String[] args){  
        Person p1=new Person("a");  
        System.out.println(p1); //1:a  
        Person p2=new Person("b");  
        System.out.println(p2); //2:b  
        Person p3=new Person("c");  
        System.out.println(p3); //3:c  
        System.out.println(Person.counter)  
    }  
}
```

פעולה חוקית יודפס 3  
Counter הוא משתנה מחלקה ולא משתנה  
מופע. כלומר הוא מקושר למחלקה ולא  
לאובייקט. לכן ניתן לגשת אליו ישירות

בדומה למשתנה מופע , פניה למשתנה  
מחלקה עם הרשאת private היא לא  
חוקית

```
public class Person{  
    private String name;  
    private static int counter;  
  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
  
    public String toString(){  
        return counter+": "+name;  
    }  
}
```

```
public class Tester {  
    public static void main(String[ ] args) {  
        Person p1=new Person("a");  
        System.out.println(p1.counter);  
        System.out.println(Person.counter);  
    }  
}
```



counter has private access in Person



ע"י הגדרת שיטה עם הרשאת `public`  
ניתן לפנות למשתנה עם הרשאת `private`

```
public class Person{  
    private String name;  
    private static int counter;  
  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
    public int getCounter(){  
        return counter;  
    }  
    public String toString(){  
        return counter+": "+name;  
    }  
}
```

Instance method can access static variables  
Only if object exists (need the "this" pointer)!!!

המטרה : לגשת למשתנה מחלקה  
שהוא עם הרשאת `private`

```
public class Tester {  
    public static void main(String[ ] args){  
        Person p1=new Person("a");  
        System.out.println(p1); //1:a  
        System.out.println(p1.getCounter());//1  
        System.out.println(Person.getCounter());  
    }  
}
```

פעולה לא חוקית  
לא ניתן לגשת לשיטת מופע ישירות (רק דרך אובייקט)  
מה עושים? הבא ונעבור לעמוד הבא

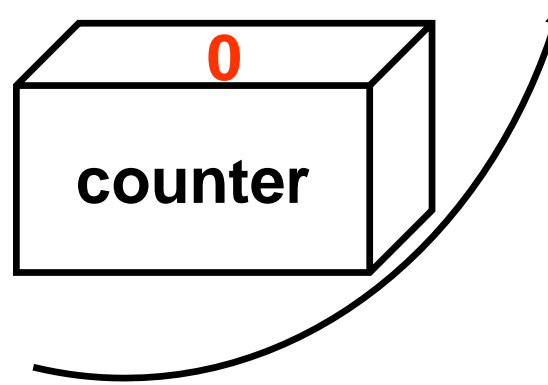
ע"י הגדרת שיטה **static** ניתן  
לפנות למשתנה **static** ללא צורך באובייקט

```
public class Person{
    private String name;
    private static int counter;
    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}
```

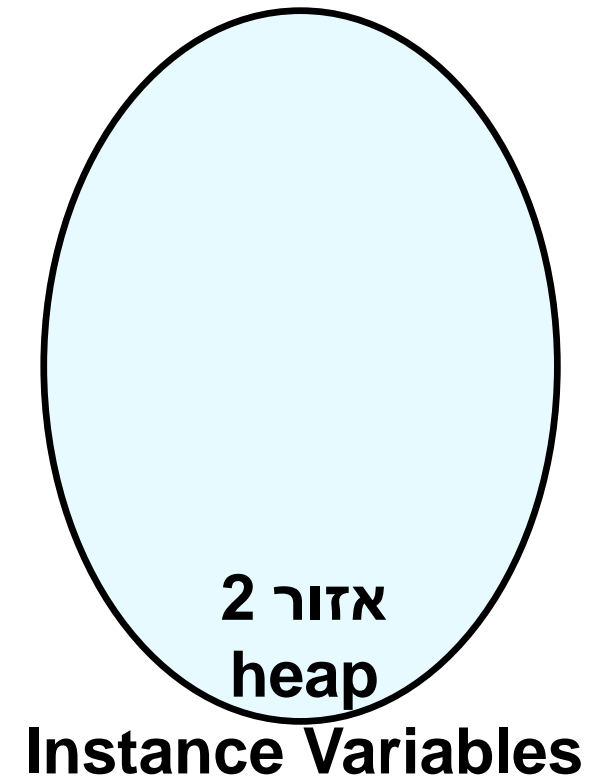
המטרה : לגשת למשתנה מחלקה  
שהוא עם הרשאת **private** ללא  
שימוש באובייקט

```
public class Tester {
    public static void main(String[ ] args){
        System.out.println(Person.getCounter());//0
        Person p1=new Person("a");
        System.out.println(p1); //1:a
        System.out.println(p1.getCounter());//1
        System.out.println(Person.getCounter()); //1
    }
}
```

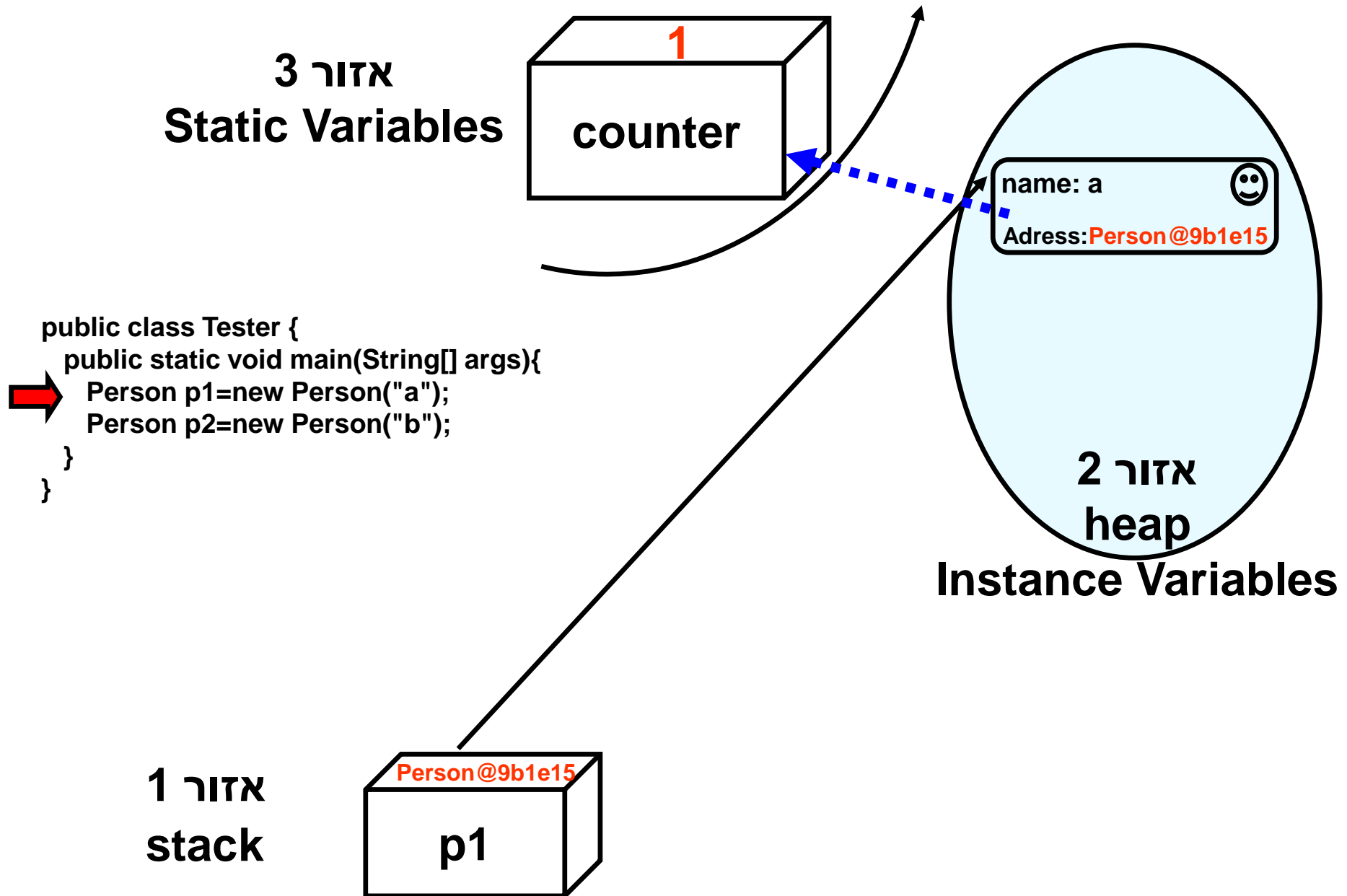
אזור 3  
Static Variables

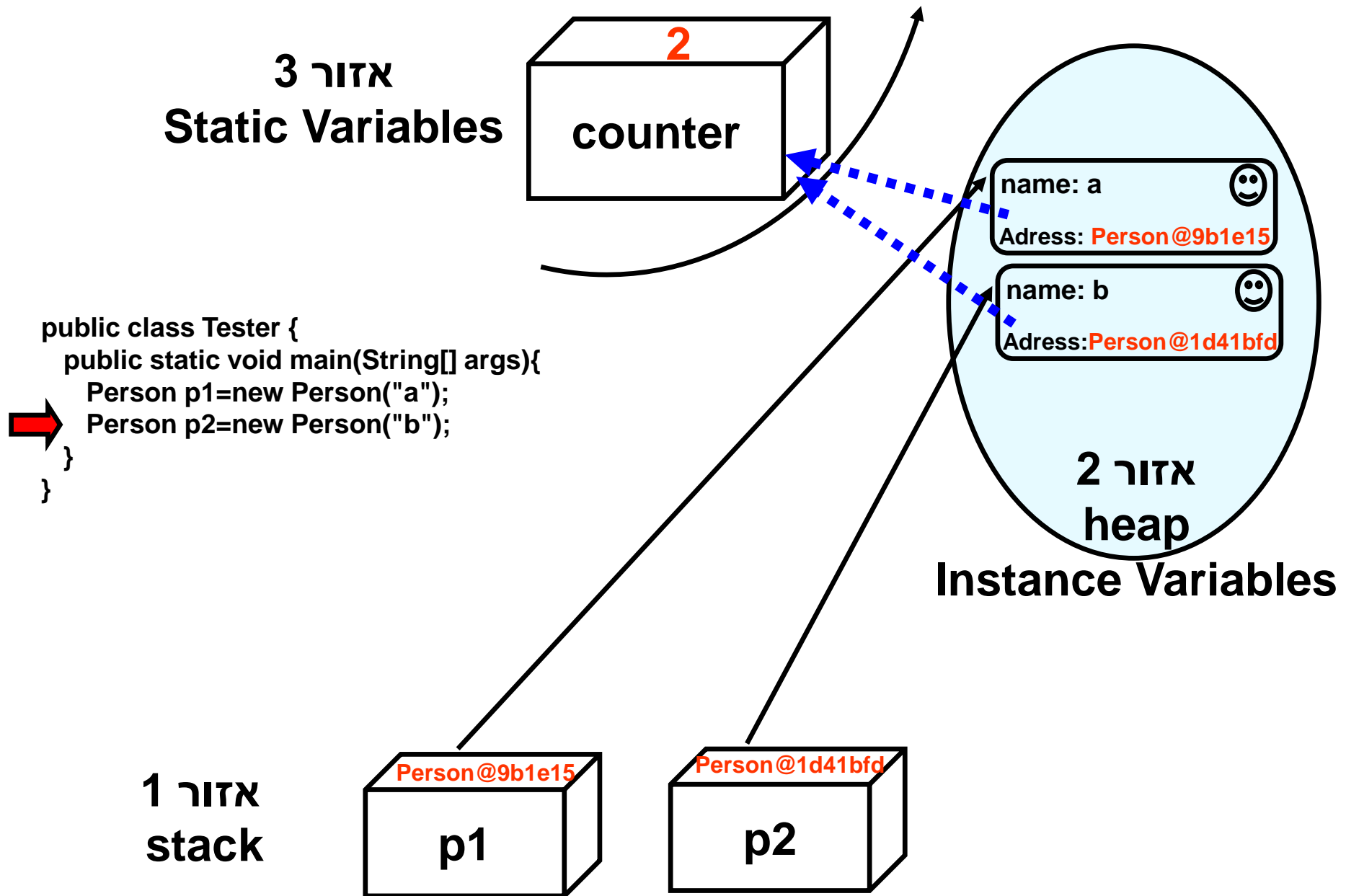


```
public class Tester {  
    public static void main(String[] args){  
        Person p1=new Person("a");  
        Person p2=new Person("b");  
    }  
}
```



אזור 1  
stack





```

public class Person{
    private String name;
    private static int counter=0;

    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        System.out.print(name);
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}

```

Static method  
 Can access only static data  
 It can not access instance variables directly

getCounter היא שיטה סטטית (מחלקה) ואילו המשתנה name הוא משתנה מופע (אובייקט). השיטה סטטית לא ניתן לגשת לשיטות או משתני מופע. ולכן מתקבלת שגיאת קומפילציה

```
public class Person{  
    private String name;  
    private static int counter=0;  
  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
    ✓ public static int getCounter(){  
        return counter;  
    }  
    public String toString(){  
        return counter+": "+name;  
    }  
}
```

getCounter היא שיטה  
סטטית (מחלקה) וגם המשתנה  
counter. לכן זה תקין

**Static method**  
**can access instance variables** indirectly

```
public class Person{  
    private String name;  
    private static int counter=0;  
  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
    public static int getCounter(Person p) {  
        System.out.print( p.name );  
        return counter;  
    }  
    public String toString() {  
        return counter + ":" + name;  
    }  
}
```

```
public class Tester {  
    public static void main(String[ ] args) {  
  
        Person p = new Person("avi");  
        int x = Person.getCounter(p);  
        System.out.println(x); // x = 1  
    }  
}
```

שימו לב ניתן לגשת למשתנה  
או שיטת מופע בדרך לא ישירה  
מתוך שיטה סטטית



```

public class Person{
    protected String name;
    protected static int counter=0;

    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}

```

```

public class Tester {
    public static void main(String[ ] args){
        System.out.println(Leader.getCounter());
        Leader L1=new Leader("Barak",10);
        System.out.println(L1);
        Leader L2=new Leader("Perse",40);
        System.out.println(L2);
    }
}

```

Output:

```

0
Leader 1:Barak with seniority 10
Leader 2:Perse with seniority 40

```

```

public class Leader extends Person{
    private int seniority;
    public Leader(String name , int seniority) {
        super(name);
        this.seniority = seniority;
    }
    public String toString(){
        return "Leader " +super.toString() + " with seniority " + seniority;
    }
}

```

אם במחלקת הבסיס  
הוגדרה משתנה/שיטה  
סטטית אז במחלקה  
הנגזרת הם נשארים  
סטטיים וההתייחסות  
אליהם בהתאם

```

public class Person{
    protected String name;
    protected static int counter=0;
    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}

```

```

public class Tester {
    public static void main(String[ ] args){
        Leader L1=new Leader("Barak",10);
        L1.showCounter();
    }
}

```

Output:

1

ניתן לפנות משיטת מופע  
לשיטה סטטית ( הזמן שבו  
נוצר האובייקט הוא מאוחר  
יותר - השיטה הסטטית  
קיימת לפני שהאובייקט נוצר)

```

public class Leader extends Person {
    private int seniority;
    public Leader(String name,int seniority) {
        super(name); this.seniority=seniority;
    }
    public void showCounter() {
        System.out.println(getCounter());
    }
    public String toString(){
        return "Leader " +super.toString() + " with seniority " + seniority;
    }
}

```

showCounter שיטת מופע  
getCounter שיטת מחלקה

Override a static method with a non-static method is not allowed

```
public class Person{
    protected String name;
    protected static int counter=0;
    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}
```

לא ניתן לדרוס שיטת סטטית ע"י שיטת מופע.  
שאלה : האם ניתן לדרוס שיטה סטטית?  
תשובה הבא ונעבור לעמוד הבא

```
public class Leader extends Person{
    private int seniority;
    public Leader(String name,int seniority){
        super(name); this.seniority=seniority;
    }
    public int getCounter(){
        return counter;
    }
    public String toString(){
        return "Leader "+super.toString()+" with seniority "+seniority;
    }
}
```

```
public class Person{
    protected String name;
    protected static int counter=0;
    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}
```

לא ניתן לדרוס שיטה סטטית בכלל  
שיטות שניתן לדרוס הן שיטות מופע בלבד.  
הערה אם מבטלים את @Override מתבצע  
red-define ולא override.

```
public class Leader extends Person{
    private int seniority;
    public Leader(String name,int seniority){
        super(name); this.seniority=seniority;
    }
    @Override
    public static int getCounter(){
        return counter ;
    }
    public String toString(){
        return "Leader "+super.toString()+" with seniority "+seniority;
    }
}
```

```

public class Person{
    protected String name;
    protected static int counter=0;
    public Person(String name){
        this.name=name;
        counter++;
    }
    public static int getCounter(){
        return counter;
    }
    public String toString(){
        return counter+": "+name;
    }
}

```

## Overriding and Hiding

זה חוקי. זהו hiding (red-define) ולא override.

```

public class Leader extends Person{
    private int seniority;
    public Leader(String name,int seniority){
        super(name); this.seniority=seniority;
    }

    public static int getCounter(){
        return counter ;
    }
    public String toString(){
        return "Leader "+super.toString()+" with seniority "+seniority;
    }
}

```

```
public class Person{
    protected String name;
    public Person(String name){
        this.name=name;
    }
    public void whoAml(){
        System.out.println("I am a Person");
    }
}c
```

```
public class Leader extends Person {
    private int seniority;
    public Leader(String name,int seniority) {
        super(name);
        this.seniority=seniority;
    }
    public static void whoAml(){
        System.out.println("I am a Leader");
    }
}
```

```
public class StrongLeader extends Leader{
    private int strength;
    public StrongLeader(String name , int seniority , int strength) {
        super(name,seniority);
        this.strength=strength;
    }
    public static void whoAml() {
        System.out.println("I am a StrongLeader");
    }
}
```

כנ"ל שיטה סטטית לא יכולה לדרוס  
שיטה לא סטטית

ניתן לגשת לשיטה סטטית משיטה סטטית  
אחרת וגם משיטת מופע

```
public class A {  
    protected static int count=0;  
  
    public A() {  
        count++;  
    }  
  
    public static int getCountFromA(){  
        return count;  
    }  
}
```

```
public class B extends A {  
  
    public static int getCountFromB(){  
        return getCountFromA();  
    }  
}
```


```
public class A {  
    protected static int count=0;  
  
    public A() {  
        count++;  
    }  
  
    public static int getCountFromA(){  
        return count;  
    }  
}
```

```
public class B extends A {  
  
    public int getCountFromB(){  
        return getCountFromA();  
    }  
}
```

לא ניתן לגשת למשתנה\שיטה ע"י שימוש ב  
super מתוך שיטה סטטית

```
public class A {  
    protected static int count=0;  
  
    public A() {  
        count++;  
    }  
  
    public static int getCountFromA(){  
        return count;  
    }  
}
```

```
public class B extends A {  
  
    public static int getCountFromB(){  
        return super.getCountFromA();  
    }  
}
```

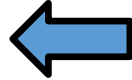


```
public class A {  
    protected static int count=0;  
  
    public A() {  
        count++;  
    }  
  
    public static int getCountFromA(){  
        return count;  
    }  
}
```

```
public class B extends A {  
  
    public int getCountFromB(){  
        return super.getCountFromA();  
    }  
}
```



20



```
public class Tester {  
    public static void main(String[] args){  
        B b=new B();  
        A a = b;  
        System.out.println(a.getValue());  
    }  
}
```

```
public class A {  
    protected static int x=10;  
    public A(){  
    }  
    public int getValue(){  
        return x;  
    }  
}
```

```
public class B extends A{  
    public B(){  
    }  
    public int getValue(){  
        return x*2;  
    }  
}
```

```
public class A {  
    protected static int x=10;  
    public A(){  
    }  
    public static int getValue(){  
        return x;  
    }  
}
```

```
public class B extends A{  
    public B(){  
    }  
    public static int getValue(){  
        return x*2;  
    }  
}
```

```
public class Tester {  
    public static void main(String[] args){  
        B b=new B();  
        A a = b;  
        System.out.println(a.getValue());  
    }  
}
```

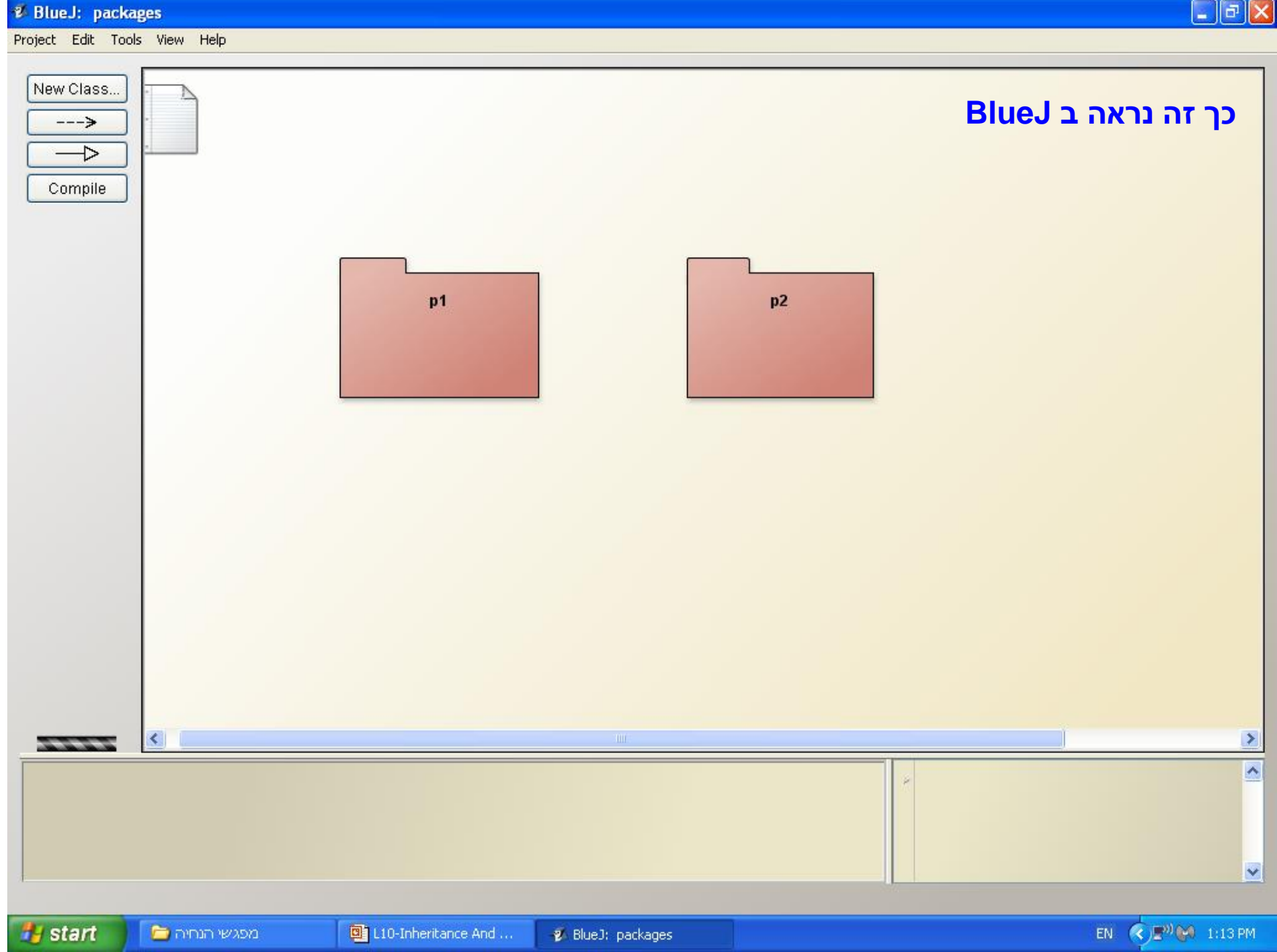


10

# Packages

## חבילות

- חבילה היא שם שניתן לאוסף משותף של מחלקות
- כל מחלקה שייכת לחבילה כלשהי
- ניתן בכל מחלקה ליצור עצמים משאר המחלקות שבחבילה ולהתייחס לשאר המחלקות ללא צורך לבצע יבוא של המחלקה -import
- להגדרת חבילה נשתמש במילה השמורה package בראש הקובץ
- כדי להתייחס למחלקה יש שתי אפשרויות
  - ✓ שימוש בשם המחלקה המלא
  - ✓ יבוא import המחלקה בשמה המלא
  - ✓ יבוא של כלל המחלקות בחבילה



ניצור שתי מחלקות A בחבילה p1 ו B בחבילה p2 כפי שמתואר למטה

```
package p1;  
public class A{  
    private int a;  
  
    public A(int a){  
        this.a=a;  
    }  
  
    public int getA(){  
        return a;  
    }  
}
```

```
package p2;  
public class B{  
    private int b;  
  
    public B(int b){  
        this.b=b;  
    }  
  
    public int getB(){  
        return b;  
    }  
}
```

```

package p1;
public class A{
    private int a;

    public A(int a){
        this.a=a;
    }

    public int getA(){
        return a;
    }
}

```

```

package p2;
public class B{
    private int b;

    public B(int b){
        this.b=b;
    }

    public int getB(){
        return b;
    }
}

```

נוסף את הטסטר  
 לחבילה p1. כיוון ש A  
 נמצא גם הוא בחבילה  
 p1 אז ניתן להגדיר  
 אובייקטים ממנו ללא  
 צורך ביבוא כפי  
 שמתואר

```

package p1;
public class Tester {
    public static void main(){
        A a1=new A(12);
        System.out.println(a1.getA()); //12
    }
}

```

```

package p1;
public class A{
    private int a;

    public A(int a){
        this.a=a;
    }

    public int getA(){
        return a;
    }
}

```

```

package p2;
public class B{
    private int b;

    public B(int b){
        this.b=b;
    }

    public int getB(){
        return b;
    }
}

```

כדי שנוכל להשתמש  
 במחלקה B מתוך  
 הטסטר שהוגדר  
 בחבילה p1 חובה  
 לייבא את המחלקה B  
 כפי שמתואר

```


package p1;
import p2.B;
public class Tester {
    public static void main(){
        B b1=new B(28);
        System.out.println(b1.getB());//28
    }
}

```

# Modifiers

- Modifiers: determine whether other classes can use a particular field or invoke a particular method.
- Two groups of modifiers
  1. Java Visibility Modifiers: public , protected , private , no-modifier.
  2. Others: final , static , abstract...
- Remember
  - ✓ Instance data should be defined with private visibility
  - ✓ Public variables violate encapsulation

# Visibility Modifiers

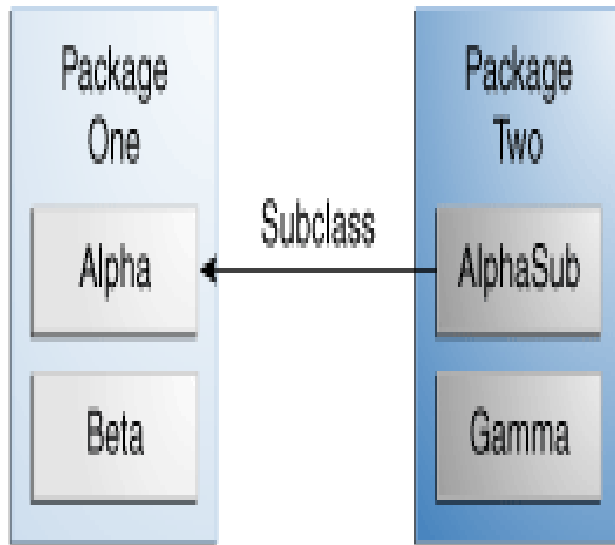


Modifier	<b>from Class</b>	<b>From Package</b>	<b>From Subclass</b>	<b>World</b> From Other packages
<b>public</b>	Y	Y	Y	Y
<b>protected</b>	Y	Y	Y	N
<b>no modifier</b>	Y	Y	N	N
<b>private</b>	Y	N	N	N

<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>



# A Visibility Example



<b>↓ Modifier of Alpha</b>	<b>Alpha</b>	<b>Beta</b>	<b>AlphaSub</b>	<b>Gamma</b>
<b>public</b>	Y	Y	Y	Y
<b>protected</b>	Y	Y	Y	N
<b>no modifier</b>	Y	Y	N	N
<b>private</b>	Y	N	N	N

<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

# Inheritance

## הורשה

- מנגנון שמאפשר להגדיר את המשותף שבין מספר מחלקות במחלקה אחת (מחלקת הבסיס)
- המחלקה הנגזרת יורשת את כל המשתנים והמתודות ממחלקת הבסיס
- ניתן להגדיר משתנים חדשים במחלקה הנגזרת
- ניתן להגדיר מתודות חדשות במחלקה הנגזרת
- ניתן לתת משמעות חדשה למתודות במחלקה הנגזרת וזאת ע"י הגדרתם מחדש במחלקה הנגזרת (מנגנון הדריסה – [Override](#)). במקרה כזה כאשר אנו פונים למתודה - נבחרת הגרסה העדכנית ביותר של המתודה
- המתודות של המחלקה הנגזרת קוראות למתודות של מחלקת הבסיס כדי שאלו תטפלנה באתחול משתנים ממחלקת הבסיס. כמו כן מתודות אלו מטפלות בעצמן במשתנים שהוגדרו במחלקה הנגזרת – דבר המאפשר את השימוש החוזר בקוד.

# יתרונות ההורשה

- דימוי של יחסים בין עצמים בעולם האמיתי
- מאפשר שימוש חוזר בקוד (reuse)
- מאפשר הרחבה של מבנה המחלקה מבלי לפגוע בתכונות הישנות שלה
- נוחות וקלות בביצוע שינויים
- חסכון בקוד (אין צורך לכתוב תכונות זהות לכל מחלקה ומחלקה)
- הגבלת גישה לנתונים (בטחון)
- מאפשר את קיומו של מנגנון הפולימורפיזם (ממשק אחיד למחלקות נגזרות)



# OOP

## תכונות עיקריות

### ▪ Encapsulation (ריכוזיות)

- ✓ מאפשר הסתרת המבנה הפנימי של המחלקה
- ✓ מאפשר בקרה נוחה על האובייקט (ממשק נוח)

### ▪ Inheritance (ירושה)

- ✓ מאפשר שימוש חוזר בקוד (reuse)
- ✓ מאפשר הרחבה של מבנה המחלקה מבלי לפגוע בתכונות הישנות שלה

### ▪ Polymorphism (רב צורתיות)

- ✓ מאפשר התייחסות לעצמים שונים בתור דברים דומים

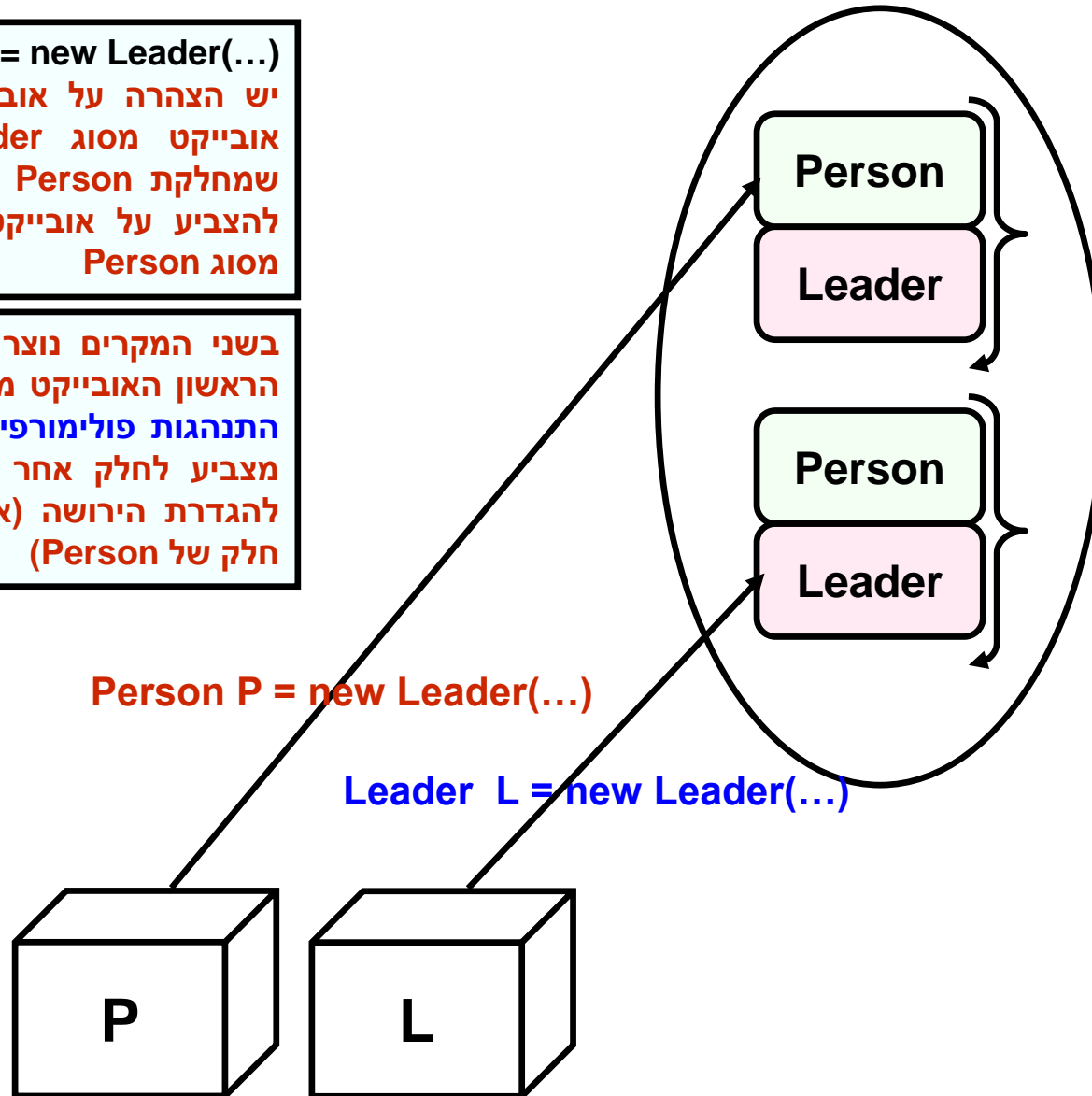
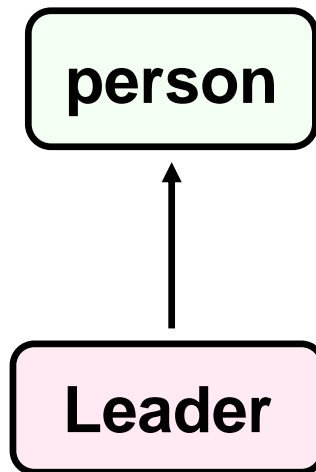
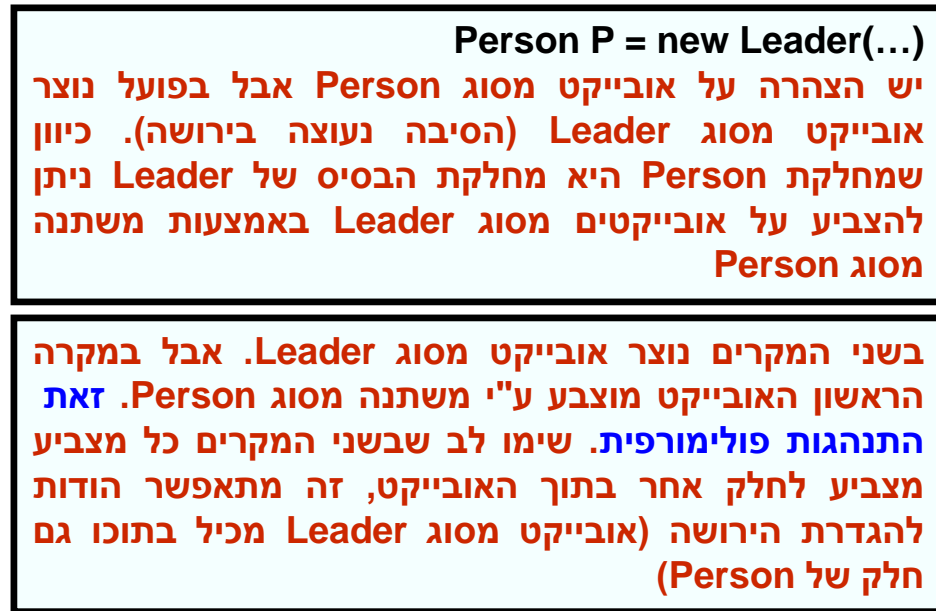
# Polymorphism

## רב-צורתיות

- מאפשר התייחסות לאובייקטים מטיפוסים שונים כאובייקטים דומים
- מאפשר כתיבת הוראות מופשטות יותר
- מאפשר הפניית מצביעים של מחלקת הבסיס (Baseclass) לאובייקטים של מחלקות הנגזרות ממנה ע"י שימוש בהמרה (casting). במקרה הזה ההמרה היא אוטומטית. כלומר מצביע של עצם הבסיס יכול להצביע בהתאם גם על עצם של מחלקה שנגזרה ממחלקת הבסיס.
- השיטה מופעלת על האובייקט הנכון (בזמן ריצה) ולא מתוך מחלקת הבסיס אשר ממנה האובייקטים ירשו את השיטה.
- התוכנית דוחה את ההחלטה על בחירת השיטה מזמן ההידור לזמן הריצה. כלומר התוכנית יודעת להחליט (בזמן הריצה) לאיזה גרסה של השיטה לפנות – וזאת בהתאם לסוג העצם שאליו מתייחס המצביע.
- בגוה כל שיטה מוגדרת כוירטואליות (בניגוד ל ++C)

Person P = new Leader(...)

Leader L = new Leader(...)



# עקרונות Polymorphism

קיימים שני עקרונות בעלי חשיבות גבוהה בפולימורפיזם

## ■מצביע בירושה

- ✓ מצביע ממחלקת בסיס יכול בפועל לשמש כמצביע לעצם ממחלקת נגזרת
- ✓ תכונה זו נקראת **up casting**

## ■מתודות וירטואליות

- ✓ בניגוד לשפות תכנות אחרות (כמו ++C) כל המתודות ב Java מוגדרות כוירטואליות
- דבר שמאפשר קריאה לגרסת המתודה בהתאם לטיפוס העצם שהמצביע מצביע עליו
- ✓ מתודות וירטואליות הן מנגנון שבו חמתאפשר להמהדר לדוחות את ההחלטה על גרסת המתודה מזמן ההידור לזמן **הריצה של התכנית**. המהדר משתמש בטכניקת קישור מאוחר (Late Binding) בכדי לדעת לאיזו גרסת מתודה לקרוא.

# Up and Down Casting

REMEMBER Up casting is done automatically, while down casting must be manually done by the programmer.

## Up Casting

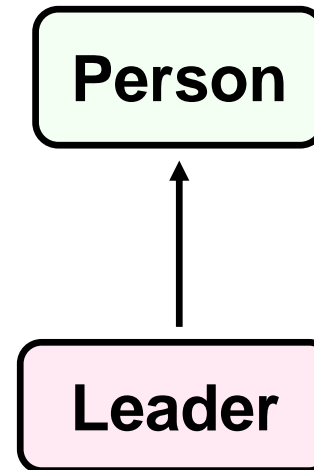
```
Person p1
Leader L1= new Leader(...)
p1= L1
או
Person p1 = new Leader(...)
```

מצביע ממחלקת הבסיס Person יכול להצביע על אובייקט ממחלקה נגזרת Leader ללא צורך בביצוע casting (ה casting מתבצע באופן אוטומטי).

## Down Casting

```
Person p1
Leader L1= new Leader(...)
p1= L1
Leader L2 = (Leader)p1
```

מצביע ממחלקה נגזרת ממחלקה כלשהי לא יכול להצביע על אובייקט ממחלקות גבוהות יותר אלא אם כן מתבצעת הממרה מפורשת למשל מצביע מסוג Leader יכול להצביע על אובייקט מסוג Person רק אחרי ביצוע casting מפורש לאובייקט שבתוך המצביע שמסוג Person כלומר מתבצעת המרת טיפוס אובייקט לטיפוס האמתי שלו





```

public class Person{
    protected String name;
    public Person(String name){
        this.name=name;
    }
    public void whoAml(){
        System.out.println("I am person");
    }
    public String toString(){
        return name;
    }
}

```

```

public class Tester {
    public static void main(String[ ] args){
        Leader L1=new Leader(...);//L1=Leader@12d294f
        L1.whoAml(); //I am a leader

        Person p1=L1; //p1=Leader@12d294f
        p1.whoAml(); // I am a leader

        Leader L2=(Leader)p1; //L2=Leader@12d294f
        L2.whoAml(); // I am a leader
    }
}

```

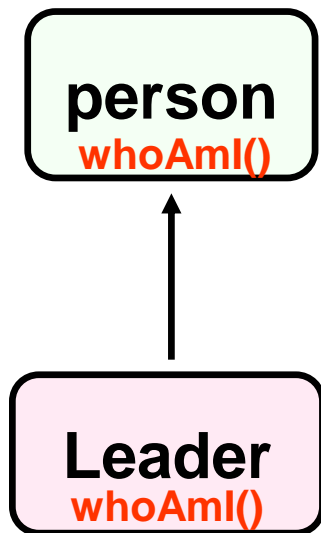
```

public class Leader extends Person {
    private int seniority;
    public Leader(String name,int seniority) {
        super(name);
        this.seniority=seniority;
    }
    public void whoAml(){
        System.out.println("I am Leader");
    }
    public String toString(){
        return "Leader " +super.toString() + " with seniority " + seniority;
    }
}

```

מה היה לנו כאן?  
הבא ונעבור לעמוד הבא

# Up and Down Casting

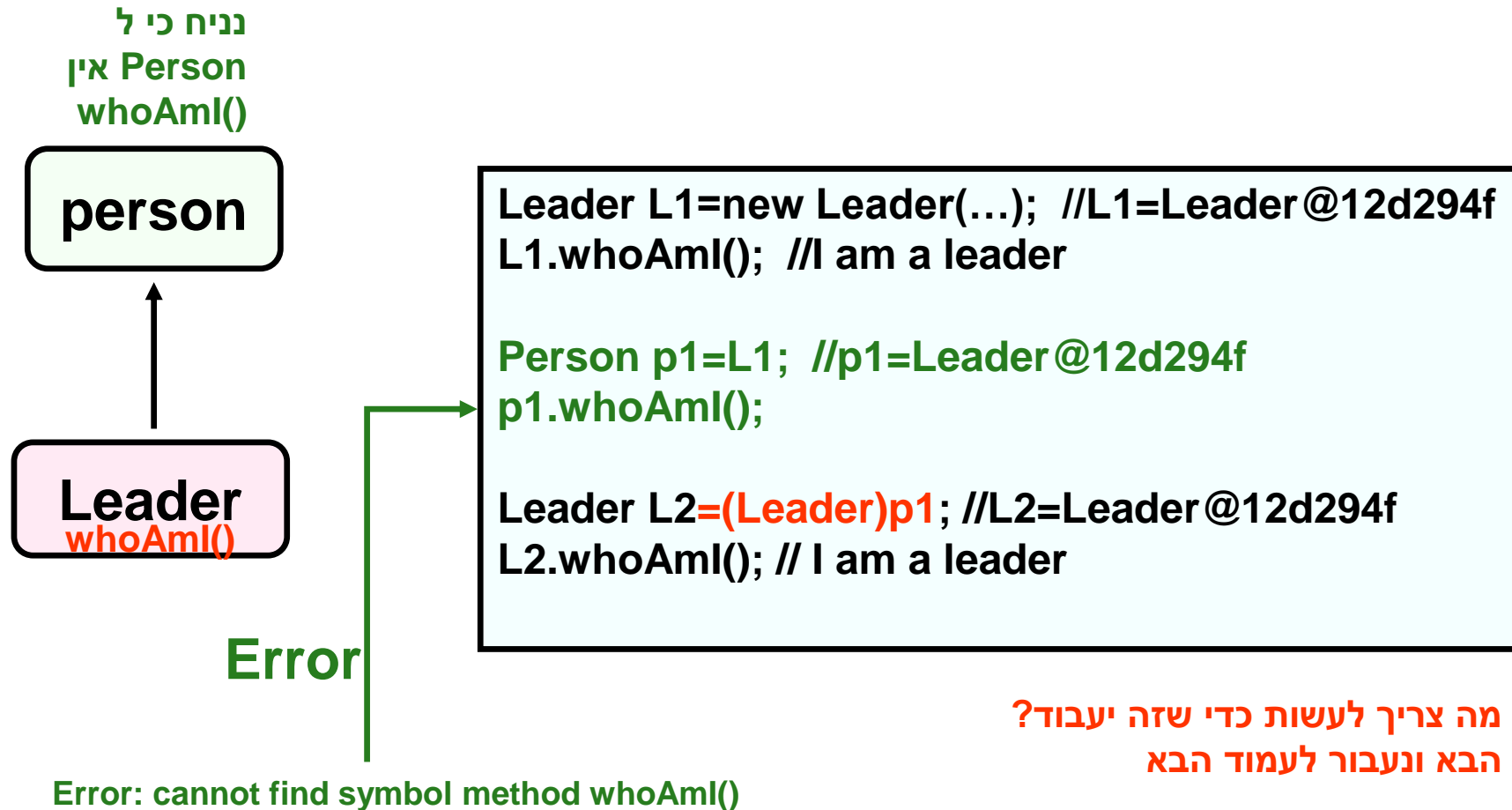


```
Leader L1=new Leader(...); //L1=Leader@12d294f
L1.whoAml(); //I am a leader
```

```
Person p1=L1; //p1=Leader@12d294f
p1.whoAml(); // I am a leader
```

```
Leader L2=(Leader)p1; //L2=Leader@12d294f
L2.whoAml(); // I am a leader
```

# Up and Down Casting



מה צריך לעשות כדי שזה יעבוד?  
הבא ונעבור לעמוד הבא

# Up and Down Casting

ל- Person אין  
whoAml()

person



Leader  
whoAml()

תיקון

```
Leader L1=new Leader(...); //L1=Leader@12d294f  
L1.whoAml(); //I am a leader
```

```
Person p1=L1; //p1=Leader@12d294f  
((Leader)p1).whoAml(); // I am a Leader
```

```
Leader L2=(Leader)p1; //L2=Leader@12d294f  
L2.whoAml(); // I am a leader
```

# Virtual Functions & Virtual Tables

- By default, C++ matches a function call with the correct function definition at **compile time**. This is called **static binding**. By declaring a function as **virtual** a compiler match a function call with the correct function definition at **run time**; this is called **dynamic binding**.
- The **virtual** keyword indicates to the compiler that it should choose the appropriate definition of a function not by the type of reference, but by the type of object that the reference refers to.
- A virtual function is a member function you may redefine for other derived classes, and can ensure that the compiler **will call the redefined virtual function for an object of the corresponding derived class**, even if you call that function with a pointer or reference to a base class of the object. In other words Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class.
- **A virtual function cannot be global or static because**, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. i.e because virtual functions are called only for objects of class types, you cannot declare global or static functions as virtual.
- A class that declares or inherits a virtual function is called a **polymorphic class**.
- **In Java**, all non-static methods are by default virtual functions - methods marked with the keyword **final**, which cannot be overridden, along with private methods, which are not inherited, are non-virtual.

```

class Base { //C++ example
public:
    virtual void function1() {
        cout<<"Base :: function1()\n";
    };
    virtual void function2() {
        cout<<"Base :: function2()\n";
    };
};
class D1: public Base {
public:
    virtual void function1() {
        cout<<"D1 :: function1()\n";
    };
};
class D2: public Base {
public:
    virtual void function2() {
        cout<< "D2 :: function2\n";
    };
};

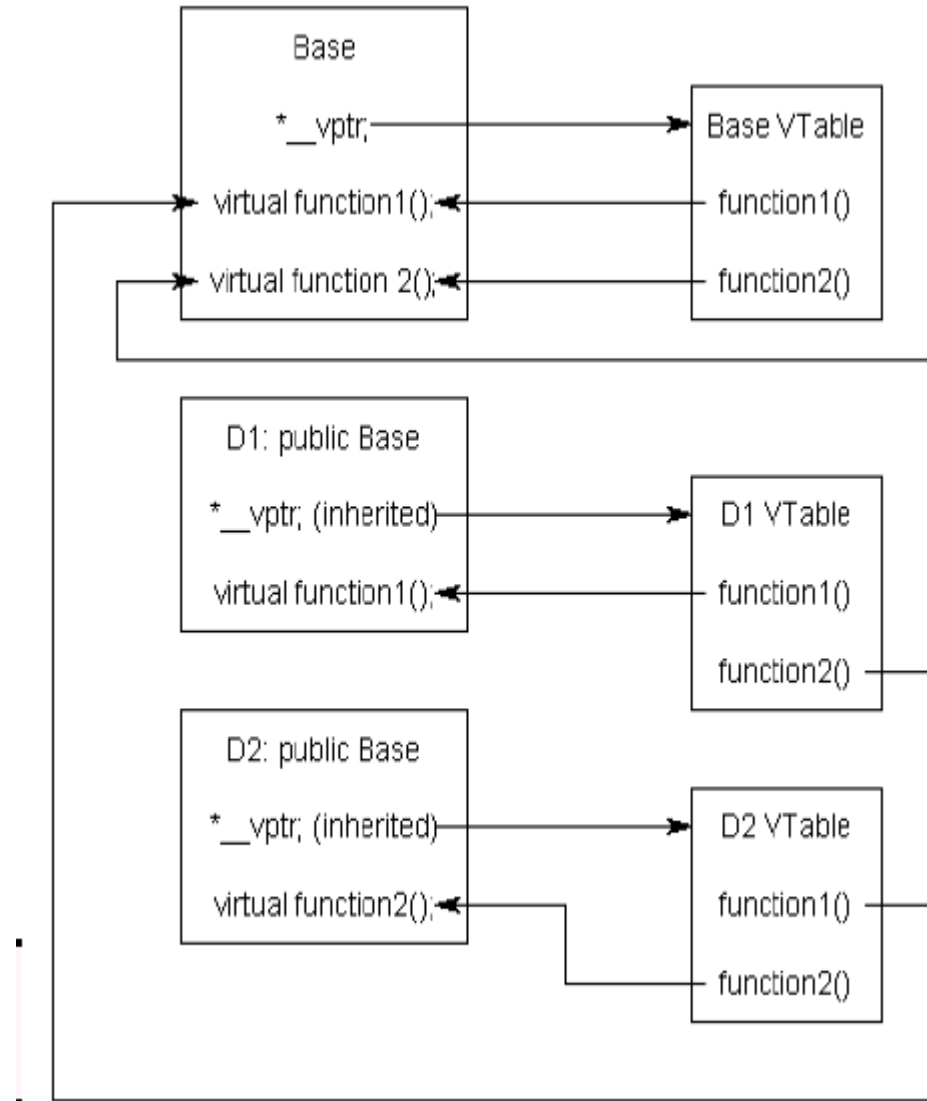
```

```

int main() {
    D1 *d = new D1;;
    Base *b = d;
    b->function1();
    b->function2();
}

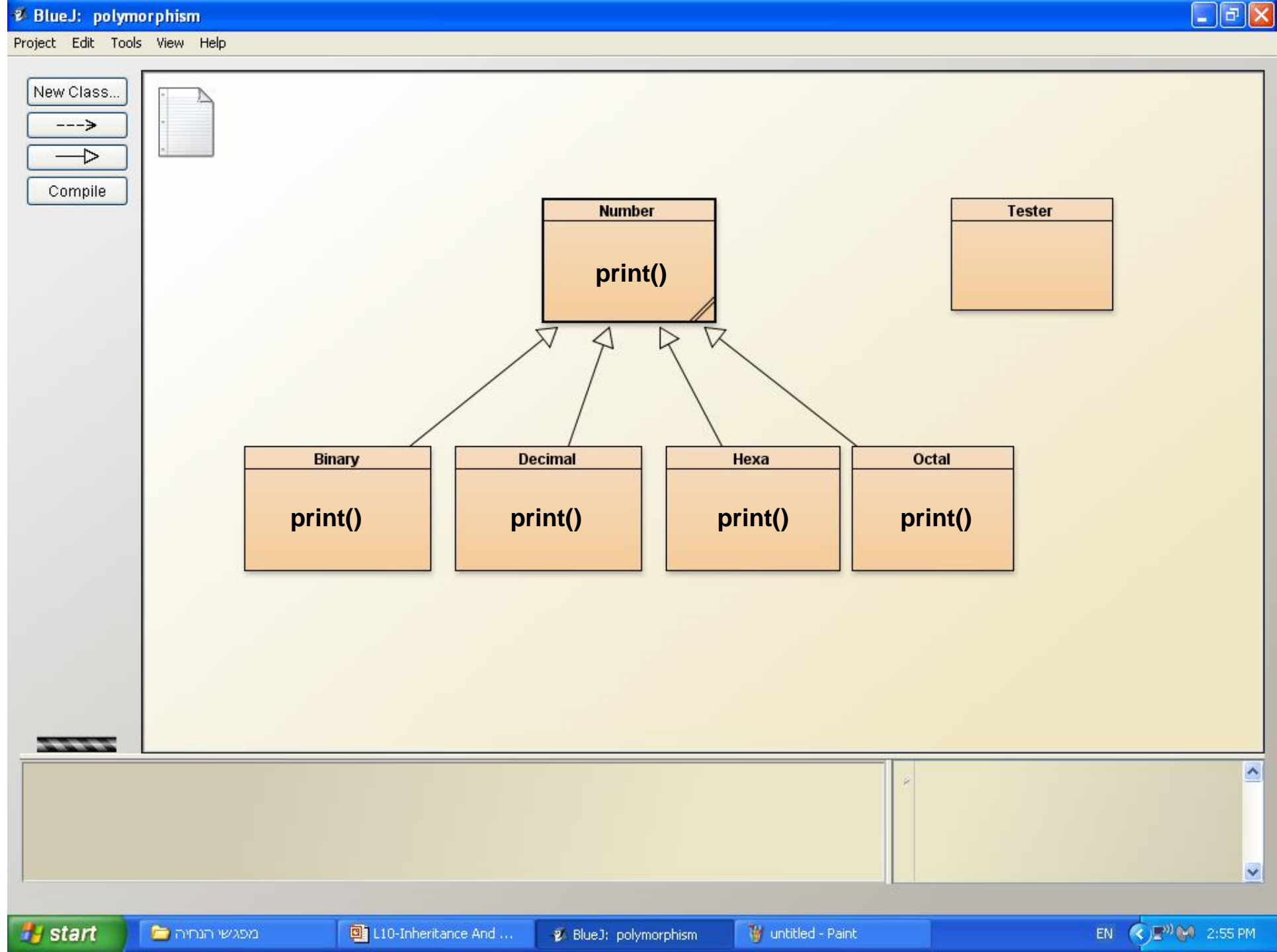
```

output:  
D1 :: function1()  
Base :: function2()



נתונה המחלקה Number אשר מייצגת ערך מספרי מסוג Integer.  
צריך ליצור את המחלקות Hexa, Binary, Octal, Decimal אשר כל אחת יורשת מהמחלקה Number ומציגה את הייצוג המתאים של הערך המספרי

```
public class Number{  
    protected int value;  
    public Number(int v) {  
        value=v;  
    }  
    public void print() {  
        System.out.println("Choose Base");  
    }  
}
```





```
import java.lang.Integer;
public class Decimal extends Number{
    public Decimal(int v) {
        super( v );
    }
    public void print(){
        System.out.println("Decimal:" + value);
    }
}
```

```
import java.lang.Integer;
public class Binary extends Number{
    public Binary(int v){
        super( v );
    }
    public void print(){
        String s = Integer.toBinaryString(value);
        System.out.println("Binary:" + s);
    }
}
```

```
import java.lang.Integer;
public class Octal extends Number{
    public Octal(int v){
        super( v );
    }
    public void print(){
        String s = Integer.toOctalString(value);
        System.out.println("Octal:" + s);
    }
}
```

```
import java.lang.Integer;
public class Hexa extends Number{
    public Hexa(int v){
        super( v );
    }
    public void print(){
        String s = Integer.toHexString(value);
        System.out.println("Hexa:" + s);
    }
}
```

```
public class Tester {  
    public static void main(){  
        Number n1= new Number(12);  
        n1.print();  
        Hexa n2= new Hexa(16);  
        n2.print();  
        Octal n3=new Octal(8);  
        n3.print();  
        Binary n4=new Binary(2);  
        n4.print();  
        Decimal n5=new Decimal(10);  
        n5.print();  
    }  
}
```

**Output:**  
**Choose Base**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

מצביע של עצם הבסיס יכול להצביע  
בהתאם גם על עצם של מחלקה שנגזרה ממחלקת הבסיס

```
public class Tester {  
    public static void main(){  
        Number n1= new Number(12);  
        n1.print();  
        Number n2= new Hexa(16);  
        n2.print();  
        Number n3= new Octal(8);  
        n3.print();  
        Number n4= new Binary(2);  
        n4.print();  
        Number n5= new Decimal(10);  
        n5.print();  
    }  
}
```

שאלה: האם ניתן להכניס למערך  
כללי עצמים מהמחלקות הנגזרות?  
תשובה: כן. רמז  
פתרון: הבא ונסתכל בעמוד הבא

**Output:**  
**Choose Base**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

עקרון 1  
מצביע בירושה

השיטה מופעלת על האובייקט הנכון (בזמן ריצה) ולא מתוך מחלקת הבסיס אשר ממנה האובייקטים ירשו את השיטה

## עיקרון 2

### מתודות וירטואליות

איזו גרסת מתודת `print()` תיקרא כשנקרא לה בהקשר של עצמים במערך?

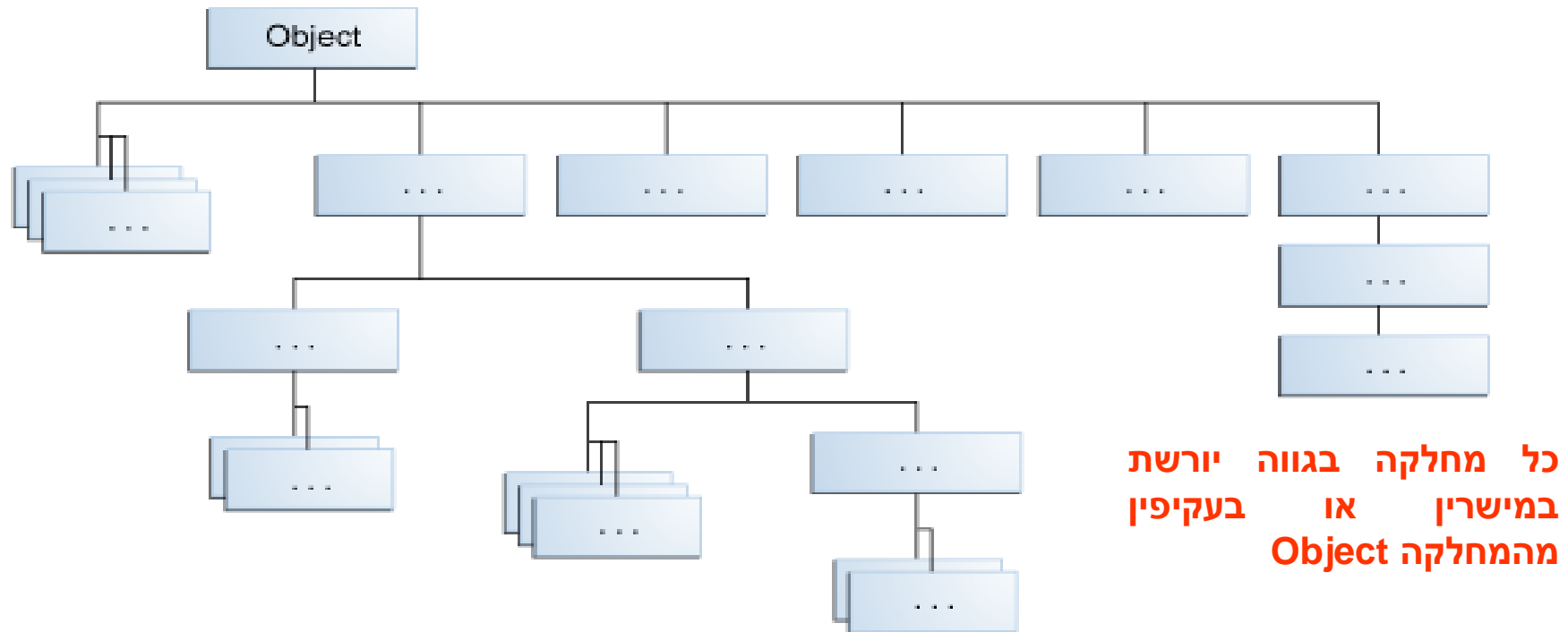
```
import java.util.Scanner;
public class Tester {
    public static void main(){
        Number[ ] arr={new Number(12),
                        new Hexa(16),
                        new Octal(8),
                        new Binary(2),
                        new Decimal(10)};

        //printing array
        for(int i=0;i<arr.length;i++)
            arr[i].print();
    }
}
```

**Output:**  
Choose Base  
Hexa:10  
Ocatl:10  
Binary:10  
Decimal:10

הערה:  
השיטה `print` חייבת להיות מוגדרת במחלקת הבסיס אחרת הזיהוי לא מתבצע ומתקבלת הודעת שגיאה בזמן קומפילציה

# The Java Platform Class Hierarchy



- The **Object** class, defined in the **java.lang package**, defines and implements behavior common to all classes - including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.
- At the top of the hierarchy, **Object is the most general** of all classes. **Classes near the bottom of the hierarchy provide more specialized behavior** (Taken from: <http://docs.oracle.com>).

Object (Java Platform SE 6) - Windows Internet Explorer

http://docs.oracle.com/javase/6/docs/api/

File Edit View Favorites Tools Help

Object (Java Platform SE 6)

Find: Object Previous Next Options More than 100 matches

## Method Summary

protected <b>Object</b>	<b>clone()</b> Creates and returns a copy of this <b>object</b> .
boolean	<b>equals(Object obj)</b> Indicates whether some other <b>object</b> is "equal to" this one.
protected void	<b>finalize()</b> Called by the garbage collector on an <b>object</b> when garbage collection determines that there are no more references to the <b>object</b> .
Class<?>	<b>getClass()</b> Returns the runtime class of this <b>Object</b> .
int	<b>hashCode()</b> Returns a hash code value for the <b>object</b> .
void	<b>notify()</b> Wakes up a single thread that is waiting on this <b>object</b> 's monitor.
void	<b>notifyAll()</b> Wakes up all threads that are waiting on this <b>object</b> 's monitor.
String	<b>toString()</b> Returns a string representation of the <b>object</b> .
void	<b>wait()</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> or <b>notifyAll()</b> method for this <b>object</b> .

Java™ Platform  
Standard Ed. 6  
[All Classes](#)  
Packages  
NumberFormatter  
NumberOfDocuments  
NumberOfInterventions  
NumberUp  
NumberUpSupporter  
NumericShaper  
NVList  
OAEPPParameterSpec  
OBJ\_ADAPTER  
**Object**  
**Object**  
**OBJECT NOT EXISTING**  
**ObjectAlreadyActive**  
**ObjectAlreadyActive**  
**ObjectChangeListener**  
**ObjectFactory**  
**ObjectFactoryBuilder**

http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#finalize()

Internet 135%

start מפגשי הנחיה L10-Inheritance Analysis BlueJ: polymorphism Windows Task Manager Object (Java Platform SE 6) EN 3:21 PM

BlueJ: polymorphism

Project Edit Tools View Help

New Class...  
-->  
->  
Compile

Object  
שאלה: האם ניתן להכניס למערך מסוג Object  
עצמים ממחלקות שלא נגזרות ישירות ממחלקות  
אחרות? למשל נניח כי המחלקות , Binary  
Decimal , Hexa , Octal לא יורשות מהמחלקה  
Number  
תשובה: כן.  
רמז: כל מחלקה נגזרת באופן אוטומטי מ Object  
פתרון:הבא ונסתכל בעמוד הבא

Tester

Binary  
Print()

Decimal  
Print()

Hexa  
Print()

Octal  
Print()

start

מסגשי הנחיה

L10-Inheritance An...

BlueJ: polymorphism

Object (Java Platfor...

untitled - Paint

EN

3:25 PM

```
import java.lang.Integer;
public class Decimal { //extends Object
    private int value;
    public Decimal(int v){
        value=v;
    }
    public void print(){
        System.out.println("Decimal:"+value);
    }
}
```

```
import java.lang.Integer;
public class Binary { //extends Object
    private int value;
    public Binary(int v){
        value=v;
    }
    public void print(){
        String s=Integer.toBinaryString(value);
        System.out.println("Binary:"+s);
    }
}
```

```
import java.lang.Integer;
public class Octal { //extends Object
    private int value;
    public Octal(int v){
        value=v;
    }
    public void print(){
        String s=Integer.toOctalString(value);
        System.out.println("Octal:"+s);
    }
}
```

```
import java.lang.Integer;
public class Hexa { //extends Object
    private int value;
    public Hexa(int v){
        value=v;
    }
    public void print(){
        String s=Integer.toHexString(value);
        System.out.println("Hexa:"+s);
    }
}
```



השיטה `print` חייבת להיות מוגדרת  
במחלקת הבסיס אחרת הזיהוי לא  
מתבצע ומתקבלת הודעת שגיאה  
בזמן קומפילציה

```
public class Tester {  
    public static void main(){  
        Object n1= new Hexa(16);  
        n1.print();  
        Object n2= new Octal(8);  
        n2.print();  
        Object n3= new Binary(2);  
        n3.print();  
        Object n4= new Decimal(10);  
        n4.print();  
    }  
}
```

**Error**

**Output:**  
cannot find symbol  
method print()

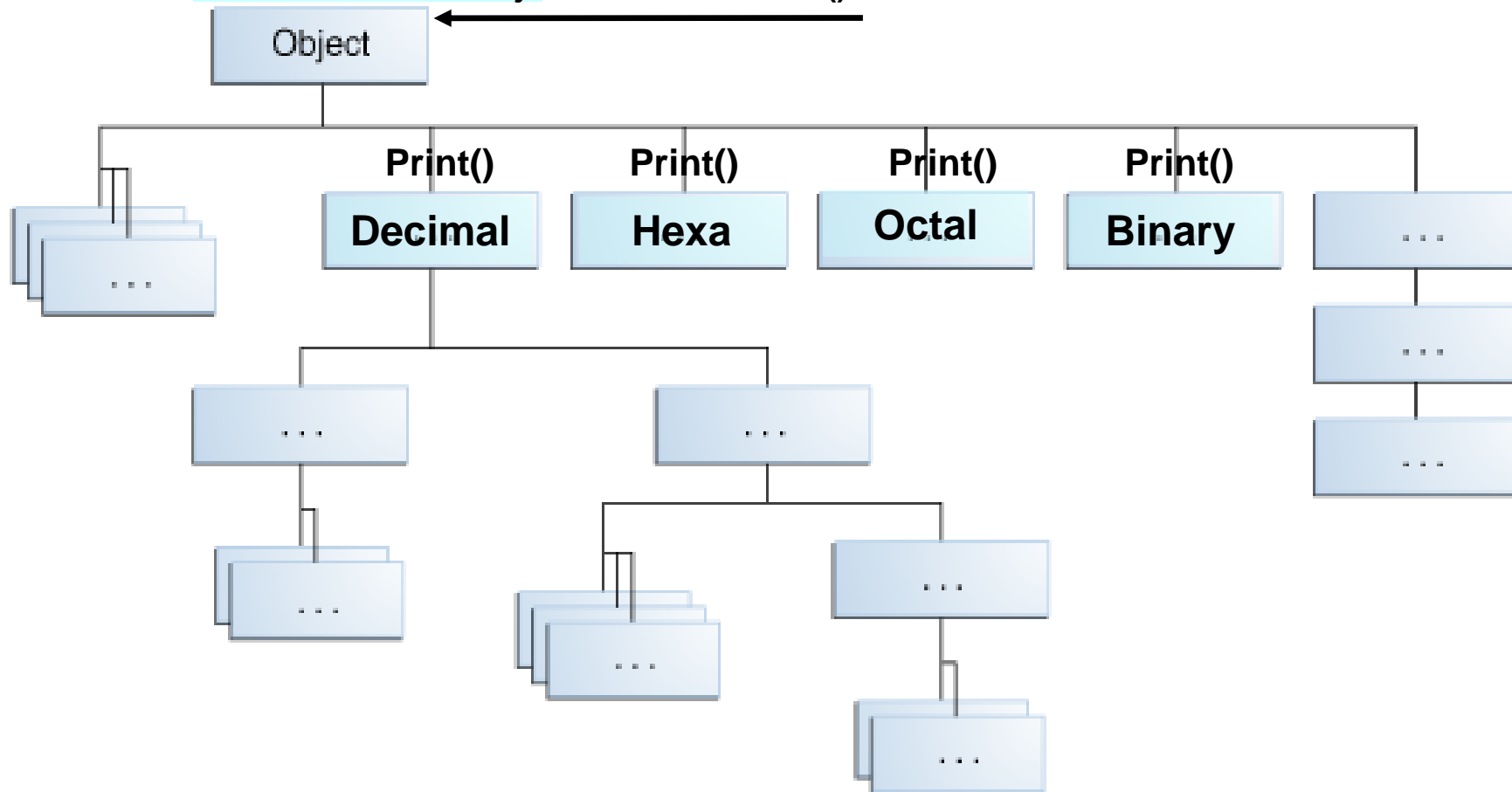
בכל זאת איך פותרים בעיה זו?  
הבא ונסתכל בעמוד הבא

## פתרון:

נשתמש באחת מהשיטות  
המוגדרות כבר ב `Object`  
השיטה המתאימה ביותר `toString`

```
equals(Object obj)
toString()
wait()
clone()
```

## Object לא נמצאת ב Print()



```
import java.lang.Integer;
public class Decimal { //extends Object
    private int value;
    public Decimal(int v){
        value=v;
    }
    public String toString(){
        return "Decimal:"+value;
    }
}
```

```
import java.lang.Integer;
public class Binary { //extends Object
    private int value;
    public Binary(int v){
        value=v;
    }
    public String toString(){
        String s=Integer.toBinaryString(value);
        return "Binary:"+s;
    }
}
```

```
import java.lang.Integer;
public class Octal { //extends Object
    private int value;
    public Octal(int v){
        value=v;
    }
    public String toString(){
        String s=Integer.toOctalString(value);
        return "Octal:"+s;
    }
}
```

```
import java.lang.Integer;
public class Hexa { //extends Object
    private int value;
    public Hexa(int v){
        value=v;
    }
    public String toString(){
        String s=Integer.toHexString(value);
        return "Hexa:"+s;
    }
}
```

```
public class Tester {  
    public static void main(){  
        Object n1=new Hexa(16);  
        System.out.println(n1);  
        Object n2=new Octal(8);  
        System.out.println(n2);  
        Object n3=new Binary(2);  
        System.out.println(n3);  
        Object n4=new Decimal(10);  
        System.out.println(n4);  
    }  
}
```

**Output:**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

# בינונו

```
import java.util.Scanner;
public class Tester {
    public static void main(){
        Object[ ] arr={new Hexa(16),
                        new Octal(8),
                        new Binary(2),
                        new Decimal(10)};

        //printing array
        for(int i=0 ; i<arr.length ; i++){
            System.out.println(arr[i]);
        }
    }
}
```

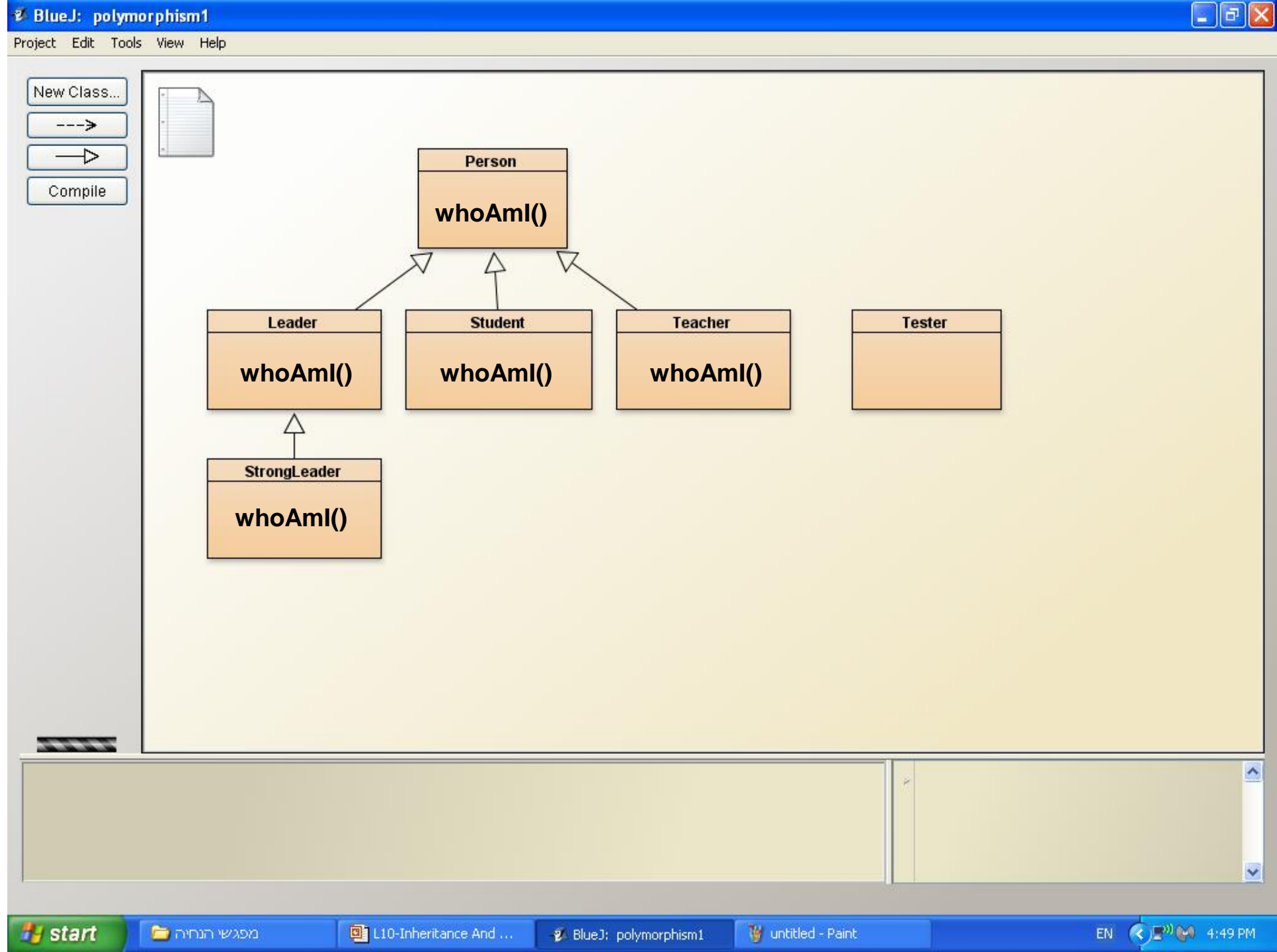
**Output:**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

# מסקנה

- כל המחלקות יורשות מ Object במישרין או בעקיפין ולכן:
- ע"י שימוש במחלקה Object ניתן להגדיר מערך של אובייקטים מסוגים שונים
- השיטות במחלקה של Object מאד מצומצמות ולכן זה מגביל אותנו בהגדרת שיטות וירטואליות
- הפתרון ניצור מחלקה שיש בה מספיק שיטות וירטואליות ואשר מיועדת להגדרת ממשק למחלקות הנגזרות ממנה (interface)
- בהמשך נראה דוגמא למחלקה מסוג כזה

נתונה המחלקה Person. צריך להגדיר את  
עץ ההורשה לפי התמונה בעמוד הבא. בכל אחת מהמחלקות  
הנגזרות צריך להגדיר את השיטה whoAmI(){...}

```
public class Person{  
    protected String name;  
    protected static int counter=0;  
    public Person(String name){  
        this.name=name;  
        counter++;  
    }  
    public void whoAmI(){  
        System.out.println("I am a person");  
    }  
    public String toString(){  
        return counter+": "+name;  
    }  
}
```





```
public class Leader extends Person{
    protected int seniority;
    public Leader(String name,int seniority){
        super(name);  this.seniority=seniority;
    }
    public void whoAmI(){
        System.out.println("I am a leader");
    }
    public String toString(){
        return "Leader "+super.toString()+" with seniority "+seniority;
    }
}
```

```
public class StrongLeader extends Leader{
    private int strength;
    public StrongLeader(String name,int seniority,int strength){
        super(name,seniority);  this.strength=strength;
    }
    public void whoAmI(){
        System.out.println("I am a strong leader");
    }
    public String toString(){
        return "Strong"+super.toString()+" and strength "+strength;
    }
}
```

```
public class Student extends Person{
    private String department;
    public Student(String name,String dept){
        super(name);
        department=dept;
    }
    public void whoAmI(){
        System.out.println("I am a student");
    }
    public String toString(){
        return "Student "+super.toString()+" Department "+department;
    }
}
```

```
public class Teacher extends Person{
    private String course;
    public Teacher(String name,String course){
        super(name);
        this.course=course;
    }
    public void whoAmI(){
        System.out.println("I am a teacher");
    }
    public String toString(){
        return "Teacher "+super.toString()+" Course "+course;
    }
}
```

נגדיר מערך מסוג Person. נאתחל מערך זה באובייקטים שונים  
מהמחלקות שיצרנו. צריך לבצע מעקב על ההדפסות

```
public class Tester {  
    public static void main(String[ ] args){  
        Person[ ] arr={  
            new Person("Avi"),  
            new Leader("Barak",7),  
            new StrongLeader("Peres",40,3),  
            new Student("Tom","History"),  
            new Teacher("Shadi","Java")};  
  
        for(int i=0 ; i<arr.length ; i++)  
            arr[i].whoAmI();  
    }  
}
```

Output:

I am a person  
I am a leader  
I am a strong leader  
I am a student  
I am a teacher

שאלה 1: בהנחה שהשיטה whoAml()  
לא הוגדרה במחלקות StrongLeader  
ו- Student מה היה מודפס?

```
public class Tester {  
    public static void main(String[ ] args){  
        Person[ ] arr={  
            new Person("Avi"),  
            new Leader("Barak",7),  
            new StrongLeader("Peres",40,3),  
            new Student("Tom","History"),  
            new Teacher("Shadi","Java")};  
  
        for(int i=0;i<arr.length;i++){  
            arr[i].whoAml();  
        }  
    }  
}
```

**Output:**  
I am a person  
I am a leader  
I am a leader  
I am a person  
I am a teacher

שאלה 2: בהנחה שהשיטה  
whoAmI() לא הוגדרה במחלקה  
Person. מה היה מודפס?

```
public class Tester {  
    public static void main(String[ ] args){  
        Person[ ] arr={  
            new Person("Avi"),  
            new Leader("Barak",7),  
            new StrongLeader("Peres",40,3),  
            new Student("Tom","History"),  
            new Teacher("Shadi","Java")};  
  
        for(int i=0; i<arr.length ; i++)  
            arr[i].whoAmI();  
    }  
}
```

**Error**

i=0

**Output:**  
cannot find symbol  
method whoAmI()

שאלה 3: בהנחה שהשיטה whoAml() לא  
הוגדרה במחלקה Person מה היה מודפס?  
שימו לב שהאובייקט new Person("Avi ")  
הוסר מהמערך

```
public class Tester {  
    public static void main(String[ ] args){  
        Person[ ] arr={  
            new Leader("Barak",7),  
            new StrongLeader("Peres",40,3),  
            new Student("Tom","History"),  
            new Teacher("Shadi","Java")};  
  
        for(int i=0;i<arr.length;i++)  
            arr[i].whoAml();  
    }  
}
```

**Error**

i=0

**Output:**  
cannot find symbol  
method whoAml()

כלל: כדי לפנות לשיטה  
ממחלקה נגזרת דרך מצביע  
של מחלקת הבסיס, חובה  
להגדיר את השיטה במחלקת  
הבסיס.

הערה: : בכול זאת ניתן  
להפעיל שיטה במחלקה  
הנגזרת גם אם לא הוגדרה  
במחלקת הבסיס על ידי פנייה  
ישירה לשיטה דרך האובייקט  
עצמו (הבא ונעבור לעמוד  
הבא)

# האופרטור instanceof

<object> instanceof <class>

■ האופרטור instanceof מספק מידע בזמן ריצה לגבי הטיפוס של עצם נתון

■ הוא מחזיר ערך true אם מתקיים אחד מהתנאים הבאים:

■ object הוא מופע של המחלקה הנתונה class

✓ object הוא מופע של מחלקה הנגזרת מהמחלקה הנתונה class

✓ object הוא מופע של מחלקה המממשת את המחלקה הנתונה class

הבא ונראה דוגמא בעמוד הבא

נוסיף את השיטה  
getDepartment() למחלקה  
Student מהתרגיל הקודם

```
public class Student extends Person{
    private String department;
    public Student(String name,String dept){
        super(name);
        department=dept;
    }
    public void whoAml(){
        System.out.println("I am a Student");
    }
    public String getDepartment(){
        return department;
    }
    public String toString(){
        return "Student "+super.toString()+" Department "+department;
    }
}
```



כמו כן נוסיף את השיטה  
Teacher **getCourse()** למחלקה  
מהתרגיל הקודם

```
public class Teacher extends Person{  
    private String course;  
  
    public Teacher(String name,String course){  
        super(name);  
        this.course=course;  
    }  
    public void whoAmI(){  
        System.out.println("I am a Teacher");  
    }  
    public String getCourse(){  
        return course;  
    }  
    public String toString(){  
        return "Teacher "+super.toString()+" Course "+course;  
    }  
}
```

BlueJ: polymorphism1

Project Edit Tools View Help

New Class...  
-->  
->  
Compile

```
classDiagram
    class Person {
        +whoAml()
    }
    class StrongLeader {
        +whoAml()
    }
    class Leader {
        +whoAml()
    }
    class Student {
        +whoAml()
        +getDepartment()
    }
    class Teacher {
        +whoAml()
        +getCourse()
    }
    class Tester {
    }
    Person <|-- StrongLeader
    Person <|-- Leader
    Person <|-- Student
    Person <|-- Teacher
    StrongLeader <|-- Leader
```

שאלה 4

הגדר מערך של אובייקטים מסוג Person אשר איבריו הם אובייקטים מהמחלקות הנגזרות. איך ניתן להתייחס לשתי השיטות החדשות בעת פניה לאובייקטים מהמערך? שימו לב ששתי השיטות לא מוגדרות במחלקת הבסיס

רמז : שימוש בהמרה (casting)

start

מסגש הנחיה

L10-Inheritance And ...


BlueJ: polymorphism1

untitled - Paint

EN

4:49 PM

```
public class Tester {  
    public static void main(String[] args){  
        Person[] arr={new Person("Avi"),  
                        new Leader("Barak",7),  
                        new StrongLeader("Peres",40,3),  
                        new Student("Tom","History"),  
                        new Teacher("Shadi","Java")};  
  
        for(int i=0 ; i<arr.length ; i++) {  
            System.out.println ( arr[i].getDepartment() );  
            System.out.println ( arr[i].getCourse() );  
        }  
    }  
}
```



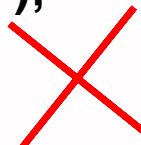
השיטות `getCourse` ו- `getDepartment` הופעלו דרך מצביע מסוג `Person`. כיוון ששיטות אלה לא הוגדרו במחלקה `Person` מתקבל הודעת שגיאה.  
שאלה האם ניתן בכל זאת להפעיל שיטות אלה על האובייקטים המתאמים?  
תשובה בעמוד הבא

```

public class Tester {
    public static void main(String[] args){
        Person[] arr={new Person("Avi"),
                       new Leader("Barak",7),
                       new StrongLeader("Peres",40,3),
                       new Student("Tom","History"),
                       new Teacher("Shadi","Java")};

        for(int i=0 ; i<arr.length ; i++) {
            if ( arr[i] instanceof Student )
                System.out.println ( arr[i].getDepartment() );
            if ( arr[i] instanceof Teacher )
                System.out.println ( arr[i].getCourse() );
        }
    }
}

```



כאן בוצע זיהוי לאובייקטים המתאמים , אבל בעת הפעלת השיטות שוב עשינו זאת דרך מצביע של מחלקת הבסיס. כלומר , השיטות getDepartment ו- getCourse הופעלו דרך מצביע מסוג Person. כיוון ששיטות אלה לא הוגדרו במחלקה Person מתקבל הודעת שגיאה.

מה עושים ?

תשובה בעמוד הבא

פתרון : המרת (down casting) האובייקט לטיפוס  
האמתי שלו ( Student ו- Teacher )

```
public class Tester {  
    public static void main(String[ ] args) {  
        Person[ ] arr = {new Person("Avi"),  
                           new Leader("Barak",7),  
                           new StrongLeader("Peres",40,3),  
                           new Student("Tom","History"),  
                           new Teacher("Shadi","Java")};  
  
        for (int i=0 ; i<arr.length ; i++) {  
            if ( arr[i] instanceof Student )  
                System.out.println ( ( (Student) arr[i] ).getDepartment() ); //History  
            if ( arr[i] instanceof Teacher )  
                System.out.println ( ( (Teacher) arr[i] ).getCourse() ); //Java  
        }  
    }  
}
```

↑  
לפני הפעלת השיטה מתבצעת המרה  
down casting לאובייקט האמתי

# מניעת הורשה ודריסה ( Overriding )

ניתן להגדיר מחלקה כ"עלה" בעץ הירושה כלומר מחלקה שלא ניתן לרשת ממנה. זוהי הגדרה מחמירה (עושים זאת במקרים מיוחדים). ניתן לבצע זאת על ידי שימוש במילה השמורה final כמו כן ניתן להגדיר מתודה שלא ניתנת לדריסה על ידי ציונה כ- final כפי שמתואר למטה

```
public final class Person {  
    :  
}
```

לא ניתן לרשת מ Person יותר

```
public final void print() {  
    :  
}
```

לא ניתן לדרוס print() יותר

# פולימורפיזם מופשט

■ פולימורפיזם מופשט הוא פולימורפיזם שמכיל אך ורק חתימת המתודות.  
כלומר החתימות מוגדרות במחלקת הבסיס והמימוש מבוצע במחלקות.

לדוגמא היה נכון להגדיר את המחלקות Shape, Person, Number  
כאבסטרקטיות שכן הן לא נועדו לייצור עצמים מאחר והן לא מתייחסות לשום ישות  
ספציפית.

■ ניתן לממש פולימורפיזם מופשט ע"י שימוש ב  
✓ מחלקות ומתודות אבסטרקטיות  
✓ ממשקים

# Abstract Classes and Methods

## מחלקות ומתודות אבסטרקטיות

- בדוגמאות שעשינו עם Number ו-Person ראינו שהיה ניתן ליצור עצמים ממחלקות אלו - דבר שהוא חסר היגיון. היה רצוי למנוע ממתכנתים מלשגות ולהגדיר עצמים מסוג זה.
- המתודות print() ו-whoAmI() הוגדרו במחלקות הבסיס אך ורק כי רצינו שיהיו וירטואליות ובכך נוכל להגדיר אותן יותר מאוחר במחלקות הנגזרות.
- לשמחתנו Java מספקת לנו מנגנון אשר יכול להתריע על מקרים מסוג זה. לפי המנגנון הזה ניתנת האפשרות (אם צריך) להגדיר מחלקה כאבסטרקטית ובכך נמענת האפשרות ליצור עצמים ממחלקה זו.
- בנוסף, ולשמחתנו גם, Java מאפשרת לנו להגדיר מתודה כאבסטרקטית דבר שמחייב את המתכנת להגדיר אותה במחלקה הנגזרת. מתודה אבסטרקטית היא מתודה ללא מימוש (רק הוכרז על החתימה שלה במחלקת הבסיס)



# הגדרת מחלקות ומתודות כאבסטרקטיות

```
public abstract class className {  
    //declare fields  
    //declare non-abstract methods  
    //declare abstract methods  
}
```

אם במחלקה כלשהי הגדרנו מתודה  
אבסטרקטית אז חובה גם שהמחלקה  
תהיה אבסטרקטית

מחלקה אבסטרקטית ←

```
modifier abstract type methodName (...);
```

מתודה אבסטרקטית ←

# תכונות מחלקות ומתודות אבסטרקטיות

- אם במחלקה כלשהי הגדרנו מתודה כאבסטרקטית אז חובה להגדיר גם את המחלקה כאבסטרקטית
- מחלקה אבסטרקטית יכולה להכיל שיטות לא אבסטרקטיות
- כלומר מחלקה אבסטרקטית יכולה להכיל שיטות עם מימוש
- מחלקות אבסטרקטיות יכולות להכיל שדות final וגם static
- מחלקה אבסטרקטית שמכילה אך ורק שיטות אבסטרקטיות – עדיף שתוגדר כממשק

הבא ונסתכל בדוגמא שבעמוד הבא

```

public class A {
    private int x;
    public A(int x){
        this.x=x;
    }
    X public abstract void print();
}

```



abstract method

```

public abstract class A {
    private int x;
    public A(int x){
        this.x=x;
    }
    ✓ public abstract void print();
}

```



abstract method

שימו לב אם הגדרנו שיטה  
כאבסטרקטית חובה להגדיר גם את  
המחלקה כאבסטרקטית אחרת  
מתקבלת הודעת שגיאה בזמן  
קומפילציה

נתונה המחלקה האבסטרקטית Number אשר מייצגת ערך מספרי מסוג Integer. צריך ליצור את המחלקות Decimal, Octal, Binary, Hexa אשר כל אחת יורשת מהמחלקה Number ומציגה את הייצוג המתאים של הערך המספרי

```
public abstract class Number{  
    protected int value;  
    public Number(int v) {  
        value=v;  
    }  
    public int getValue(){  
        return value;  
    }  
    public void setValue(int v){  
        value=v;  
    }  
    public abstract void print();  
}
```

← abstract method

BlueJ: polymorphism

Project Edit Tools View Help

New Class...

--->

→

Compile

```
classDiagram
    class Number {
        getValue()
        setValue(int v)
        print()
    }
    class Binary {
        getValue()
        setValue(int v)
        print()
    }
    class Decimal {
        getValue()
        setValue(int v)
        print()
    }
    class Hexa {
        getValue()
        setValue(int v)
        print()
    }
    class Octal {
        getValue()
        setValue(int v)
        print()
    }
    class Tester
    Number <|-- Binary
    Number <|-- Decimal
    Number <|-- Hexa
    Number <|-- Octal
```

Number

getValue()  
setValue(int v)  
print()

Tester

Binary

getValue()  
setValue(int v)  
print()

Decimal

getValue()  
setValue(int v)  
print()

Hexa

getValue()  
setValue(int v)  
print()

Octal

getValue()  
setValue(int v)  
print()

השיטה print לא מומשה במחלקת הבסיס  
אבל מומשה בכל אחת מהמחלקות הנגזרות

```
import java.lang.Integer;
public class Decimal extends Number{
    public Decimal (int v){
        super(v);
    }
    public void print(){
        System.out.println("Decimal:"+value);
    }
}
```

```
import java.lang.Integer;
public class Binary extends Number{
    public Binary (int v){
        super(v);
    }
    public void print(){
        String s=Integer.toBinaryString(value);
        System.out.println("Binary:"+s);
    }
}
```

```
import java.lang.Integer;
public class Octal extends Number{
    public Octal (int v){
        super(v);
    }
    public void print(){
        String s=Integer.toOctalString(value);
        System.out.println("Octal:"+s);
    }
}
```

```
import java.lang.Integer;
public class Hexa extends Number{
    public Hexa (int v){
        super(v);
    }
    public void print(){
        String s=Integer.toHexString(value);
        System.out.println("Hexa:"+s);
    }
}
```

המחלקה Number הוגדרה  
כאבסטרקטית לכן לא ניתן ליצור  
ממנה אובייקטים

```
public class Tester{  
    public static void main (String[] args){  
  
        Number n=new Number(10); ×  
  
    }  
}
```

**Output:**  
**Number is abstract- cannot**  
**be instantiated**

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

```
        Decimal n1=new Decimal(10);
```

```
        n1.print();
```

```
        Hexa n2=new Hexa(16);
```

```
        n2.print();
```

```
        Octal n3=new Octal(8);
```

```
        n3.print();
```

```
        Binary n4=new Binary(2);
```

```
        n4.print();
```

```
        n1.setValue(11);
```

```
        n1.print();
```

```
        n2.print();
```

```
        n3.print();
```

```
        n4.print();
```

```
    }  
}
```

Output:  
Decimal:10  
Hexa:10  
Ocatl:10  
Binary:10  
Decimal:11  
Hexa:10  
Ocatl:10  
Binary:10



שימו לב  
מותר למחלקת הבסיס להצביע על  
אובייקטים ממחלקות נגזרות גם אם  
היא אבסטרקטית

```
public class Tester{  
    public static void main(String[ ] args){  
        Number p1= new Decimal(10);  
        p1.print();  
        Number p2=new Hexa(16);  
        p2.print();  
        Octal o=new Octal(8);  
        Number p3=o;  
        p3.print();  
        Binary b=new Binary(2);  
        Number p4=b;  
        p4.print();  
    }  
}
```

Output:  
Decimal:10  
Hexa:10  
Ocatl:10  
Binary:10

הוספת אובייקט מסוג  
Number יגרום לשגיאת  
קומפילציה!!!

שאלה: האם ניתן להגדיר מערך  
מסוג Number אשר מכיל עצמים  
מהמחלקות הנגזרות?  
תשובה: כן מותר להגדיר מערך גם  
אם המחלקה אבסטרקטית

```
public class Tester {  
    public static void main(){  
        Number[ ] arr={//new Number(12),  
                        new Hexa(16),  
                        new Octal(8),  
                        new Binary(2),  
                        new Decimal(10)};  
  
        //printing array  
        for(int i=0;i<arr.length;i++)  
            arr[i].print();  
    }  
}
```

**Output:**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

# Interfaces

## ממשקים

- היא מוגדרת על ידי המילה השמורה interface במקום class
- משתמשים ב Interface -לקביעת הממשק עבור מחלקות בעץ ירושה מסוים
- ניתן להגדיר בממשק גם משתנים בנוסף למתודות
- משתנים בממשק מוגדרים כ- **public static final** אפילו אם זה לא צויין במפורש
- אי אפשר ליצור עצמים של interfaces
- כל מחלקה שמממשת ממשק תהיה חייבת לממש את כל הפונקציות שלו
- מחלקה יכולה לרשת רק ממחלקה בודדת אך היא יכולה לממש ממשקים רבים
- כל המתודות בממשק מוגדרת כ **public** אפילו אם זה לא צויין במפורש
- מימוש של כל המתודות במחלקות הנגזרות גם הוא יהיה **public**

```

public abstract class Number implements MyInterface {
    protected int value;
    public Number(int v) {
        value=v;
    }
    public int getValue(){
        return value;
    }
    public void setValue(int v){
        value=v;
    }
}

```

```

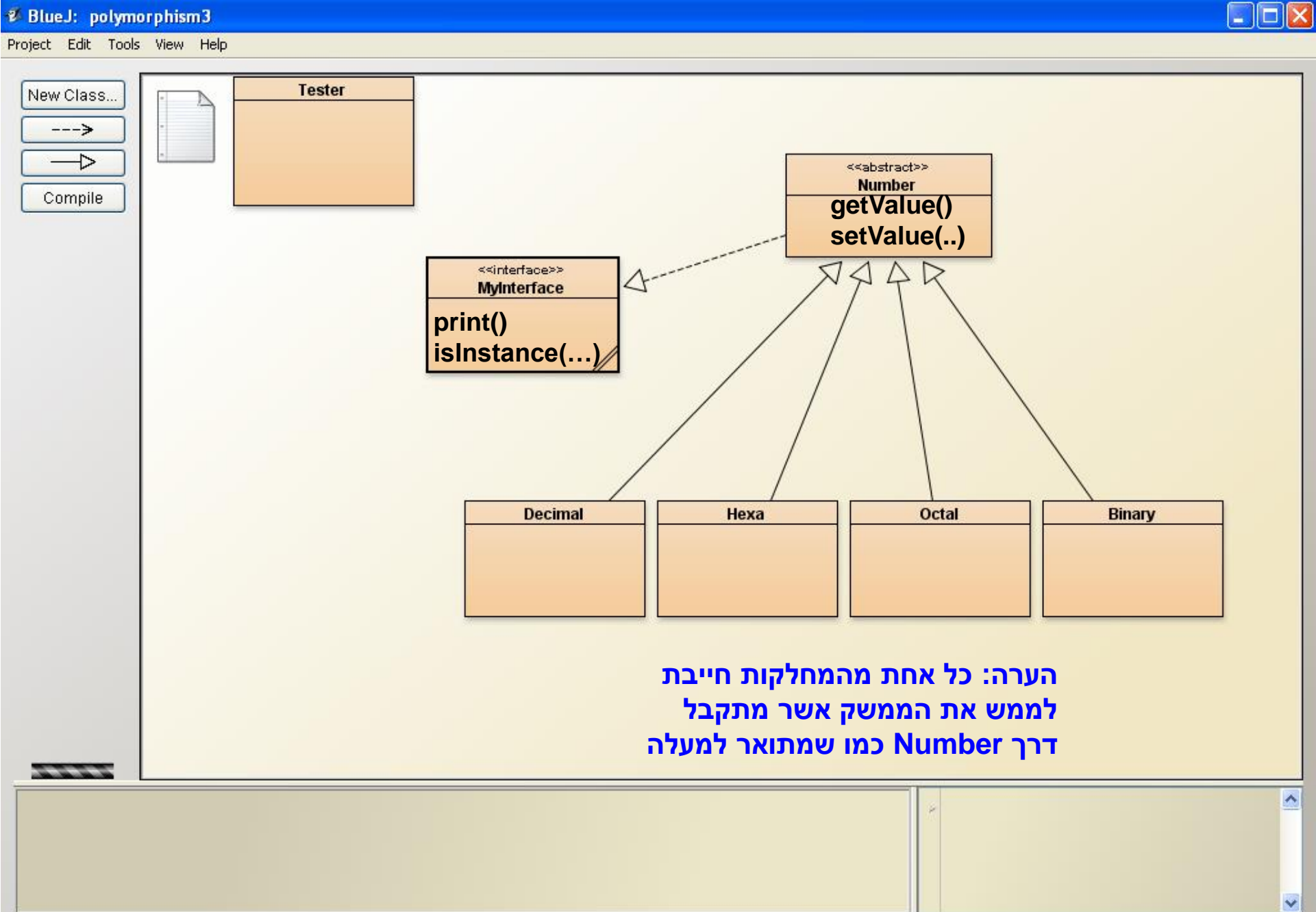
public interface MyInterface {
    public static final int counter=0;
    public void print();
    public boolean isInstance(Number temp);
}

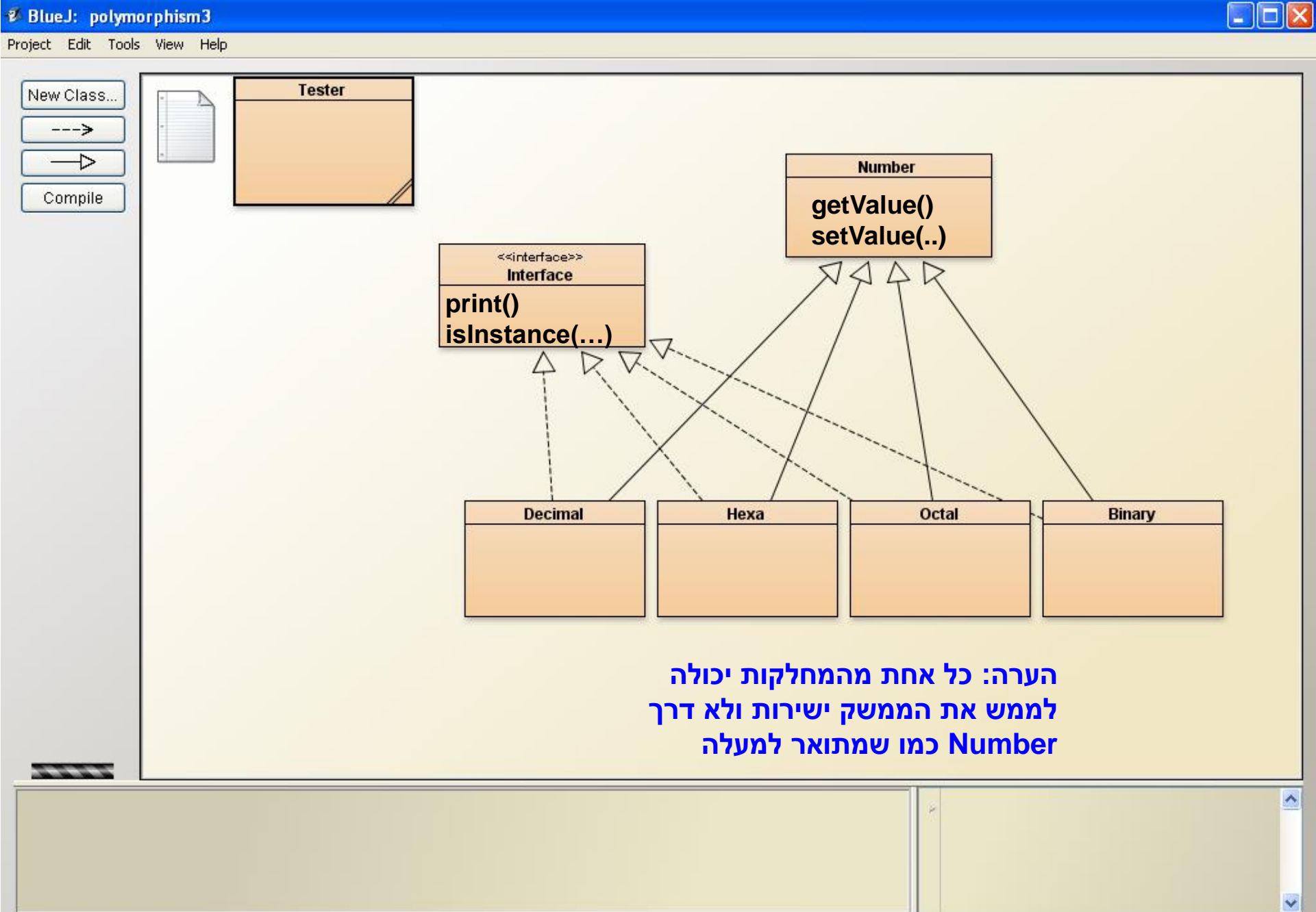
```

נתונה המחלקה Number  
 abstract אשר מייצגת ערך מספרי  
 מסוג Integer ומממשת את  
 הממשק MyInterface. צריך ליצור  
 את המחלקות Octal, Binary,  
 Decimal, Hexa אשר כל אחת  
 יורשת מהמחלקה Number.  
 ומציגה את הייצוג המתאים של  
 הערך המספרי.

שימו לב מחלקה שיורשת  
 ממחלקה אשר מממשת  
 ממשק כלשהו חייבת לממש  
 את אותו הממשק בעצמה אלא  
 אם מחלקה זו היא  
 אבסטרקטית.

שימו לב המחלקה Number לא מממשת  
 בפועל את השיטות שבממשק ( איך זה)  
 תשובה : מחלקה אבסטרקטית שבהכרזה  
 שלה אמורה לממש ממשק כלשהו לא  
 חייבת בפועל לממש את השיטות בממשק  
 אלא להעביר זאת למחלקות אחרות  
 שיורשות ממנה. בדוגמה שלנו המחלקות  
 Octal , Binary , Hexa , Decimal





```

public class Hexa extends Number
    public Hexa(int v){
        super(v);
    }
    public void print(){
        String s=Integer.toHexString(value);
        System.out.println("Hexa:"+s);
    }
    public boolean isInstance(Number temp){
        return (temp instanceof Hexa);
    }
}

```

```

public class Decimal extends Number {
    public Decimal(int v){
        super(v);
    }
    public void print(){
        System.out.println("Decimal:" + value);
    }
    public boolean isInstance(Number temp){
        return (temp instanceof Decimal);
    }
}

```

```

public class Binary extends Number
    public Binary(int v){
        super(v);
    }
    public void print(){
        String s=Integer.toBinaryString(value);
        System.out.println("Binary:"+s);
    }
    public boolean isInstance(Number temp){
        return (temp instanceof Binary);
    }
}

```

```

public class Octal extends Number {
    public Octal(int v){
        super(v);
    }
    public void print(){
        String s=Integer.toOctalString(value);
        System.out.println("Octal:"+s);
    }
    public boolean isInstance(Number temp){
        return (temp instanceof Octal);
    }
}

```

```
public class Tester{
    public static void main(String[ ] args){
        Number n=new Number(10);
    }
}
```

המחלקה Number היא  
אבסטרקטית ולכן לא ניתן  
ליצור ממנה אובייקטים

```
public class Tester {
    public static void main() {
        Decimal d1=new Decimal(10);
        Decimal d2=new Decimal(5);
        Hexa h=new Hexa(16);
        System.out.println(d1.isInstance(h));
        System.out.println(d1.isInstance(d2));
    }
}
```

Output:  
false  
true

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

        MyInterface n=new Decimal(10);
        n.print();

    }
}
```

Output:  
Decimal:10

שימו לב  
ניתן להצביע בעזרת מזהה של ממשק  
לאובייקטים ממחלקות שמממשות אותו )  
זה לא סותר את העובדה שלא ניתן ליצור  
אובייקטים מממשק)



```
public class Tester{  
    public static void main(String[ ] args){  
  
        Decimal n1=new Decimal(10);  
        n1.print();  
        Hexa n2=new Hexa(16);  
        n2.print();  
        Octal n3=new Octal(8);  
        n3.print();  
        Binary n4=new Binary(2);  
        n4.print();  
        System.out.println(n4.counter);//0  
    }  
}
```

Output:  
Decimal:10  
Hexa:10  
Ocatl:10  
Binary:10

שימו לב  
מותר לממשק להצביע על  
אובייקטים ממחלקות שמממשות אותו

```
public class Tester{  
    public static void main(String[ ] args){  
        MyInterface p1= new Decimal(10);  
        p1.print();  
        MyInterface p2=new Hexa(16);  
        p2.print();  
        Octal o=new Octal(8);  
        MyInterface p3=o;  
        p3.print();  
        Binary b=new Binary(2);  
        MyInterface p4=b;  
        p4.print();  
    }  
}
```

Output:  
Decimal:10  
Hexa:10  
Ocatl:10  
Binary:10

הגדרת מערך מסוג MyInterface אשר  
מכיל עצמים מהמחלקות המממשות אותו

הוספת אובייקט מסוג Number  
יגרום לשגיאת קומפילציה!!!

```
public class Tester {  
    public static void main(){  
        MyInterface[ ] arr={//new Number(12),  
                               new Hexa(16),  
                               new Octal(8),  
                               new Binary(2),  
                               new Decimal(10)};  
  
        //printing array  
        for(int i=0;i<arr.length;i++)  
            arr[i].print();  
    }  
}
```

**Output:**  
**Hexa:10**  
**Ocatl:10**  
**Binary:10**  
**Decimal:10**

## תרגיל 1

```
public class A {  
  
    public void print1( ) {  
        System.out.println("A");  
    }  
  
    public void print2( ){  
        print1();  
    }  
  
}
```

```
public class B extends A {  
  
    public void print1( ) {  
        System.out.println("B");  
    }  
  
}
```

```
public class Tester {  
    public static void main(String[ ] args) {  
  
        B b = new B( );  
        b.print2();  
  
    }  
}
```

Output:  
B

B קבלה בירושה את print1. במחלקה B בוצעה דריסה לשיטה print1 לכן אם תופעל השיטה print1 על אובייקט מסוג B – תופעל השיטה המעודכנת. האובייקט b הפעיל את השיטה print2 אשר התקבלה גם היא בירושה מ A. השיטה print2 פונה לשיטה print1 עם הגרסה המעודכנת לכן יודפס B.

```

public class A {
    public static int x=1;
    public void f() {
        System.out.println("A:" + x++);
    }
    public void f(Object obj) {
        System.out.println("AA:" + x++);
    }
}

```

```

public class B extends A {
    public void f() {
        System.out.println("B:" + x++);
    }
    public void f(A a) {
        System.out.println("AB:" + x++);
    }
    public void f(B b) {
        System.out.println("BB:" + x++);
    }
}

```

```

public class Tester {
    public static void main(String[ ] args){
        A a1 = new A();
        A a2 = new B();
        B b = new B();

        a1.f();
        a1.f(new Object());
        a1.f(a1);
        a1.f(a2);
        a1.f(b);

        a2.f();
        a2.f(new Object());
        a2.f(a1);
        a2.f(a2);
        a2.f(b);

        b.f();
        b.f(new Object());
        b.f(a1);
        b.f(a2);
        b.f(b);
    }
}

```

output:

```

A:1
AA:2
AA:3
AA:4
AA:5
B:6
AA:7
AA:8
AA:9
AA:10
B:11
AA:12
AB:13
AB:14
BB:15

```

```

public class A {
    public static int x=1;
    public void f() {
        System.out.println("A:" + x++);
    }

    public void f(Object obj) {
        System.out.println("AA:" + x++);
    }
}

```

```

public class B extends A {
    public void f() {
        System.out.println("B:" + x++);
    }

    public void f(A a) {
        System.out.println("AB:" + x++);
    }

    public void f(Object obj) {
        System.out.println("BO:" + x++);
    }
}

```

```

public class Tester {
    public static void main(String[ ] args){
        B b1 = new B();
        B b2 = new B();
        b1.f(b2);
    }
}

```

output:  
AB:1

```
public class A {
    public int x;

    public A()
    {
        x=10;
    }

    public A(int a){
        x=a;
    }
}
```

```
public class B extends A {
    public int x;

    public B(int a){
        x=a;
    }
}
```

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

        B b = new B(6);
        System.out.println(b.x);

        A ab = new B(6);
        System.out.println(ab.x);

        System.out.println(((B)ab).x);
    }
}
```

Output:  
6  
10  
6

```
public class A {
    public int x;
    public A(){
        x=10;
    }
    public A(int a){
        x=a;
    }
    public int f1(){
        return x;
    }
}
```

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

        B b = new B(6);
        System.out.println(b.f1());

        A ab = new B(6);
        System.out.println(ab.f1());

    }
}
```

Output:  
6  
6

```
public class B extends A {
    public int x;
    public B(int a){
        x=a;
    }
    public int f1(){
        return x;
    }
}
```



```
public class A {
    public int x;
    public A(){
        x=10;
    }
    public A(int a){
        x=a;
    }
    public int f1(){
        return x;
    }
}
```

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

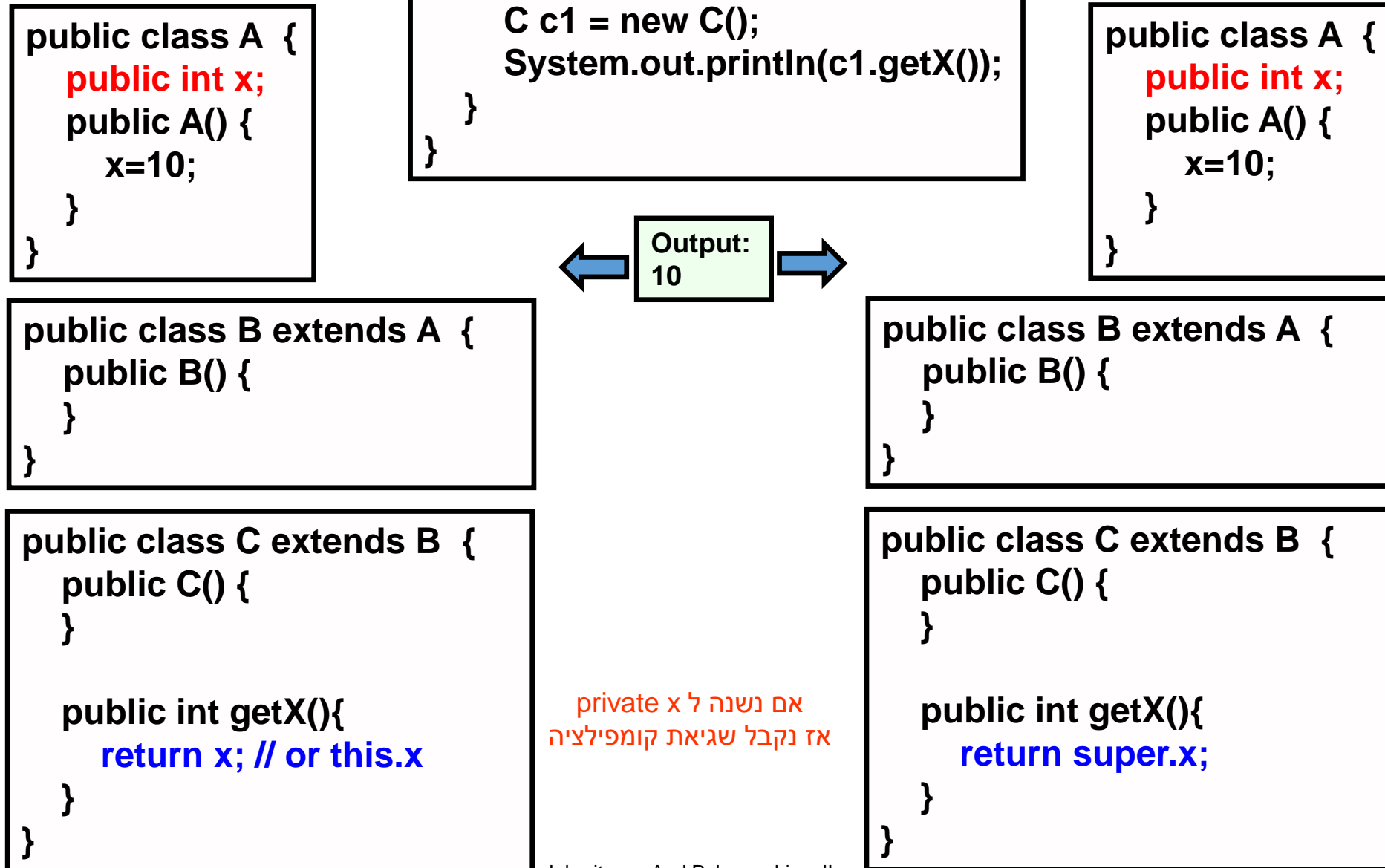
        B b = new B(6);
        System.out.println(b.f1());

        A ab = new B(6);
        System.out.println(ab.f1());

    }
}
```

Output:  
10  
10

```
public class B extends A {
    public int x;
    public B(int a){
        x=a;
    }
    public int f1(){
        return super.x;
    }
}
```



```

public class A {
    public A() {
        System.out.println ("A");
    }

    public void arik () {
        System.out.println ("Arik_A");
    }

    public void yosef () {
        arik();
    }
}

```

```

public class B extends A {
    public B() {
        System.out.println ("B");
    }

    public void arik () {
        System.out.println("Arik_B");
    }

    public void yosef () {
        System.out.println ("Yosef");
    }

    public void superYosef(){
        super.yosef();
    }
}

```

```

public class What {
    public static void main (String [ ] ags)
    {
        1) A a = (A) new B();
        2) A aa = new A();
        3) A ab = new B();
        4) B ba = new A();
        5) B bb = new B();
        6) aa.yosef();
        7) ab.yosef();
        8) bb.yosef();
        9) ((A) aa).yosef();
        10) ((A) bb).yosef();
        11) ((A) bb).superYosef();
        12) ((B) aa).yosef();
        13) ((B) aa).superYosef();
        14) ((B) ab).superYosef();
        15) ((B) bb).superYosef();
    }
}

```

**Output:**

- 1) A  
B
- 2) A
- 3) A  
B
- 4) Compilation time error  
incompatible types  
sub class cannot point  
to its super class
- 5) A  
B
- 6) Arik\_A
- 7) Yosef
- 8) Yosef
- 9) Arik\_A
- 10) Yosef
- 11) Compilation time error  
superYosef() is not  
define in class A
- 12) Run time error  
A cannot be cast to B
- 13) Run time error  
A cannot be cast to B
- 14) Arik\_B  
superYosef() call yosef()  
from A that call arik()  
from B
- 15) Arik\_B  
As in 14

```

public class A {
    protected int num;
    public A(int n){
        num = n;
    }
    public int getNum() {
        return num;
    }
    public boolean f(A a){
        return num == a.num * 2;
    }
}

```

```

public class B extends A {
    public B(int n) {
        super(n);
    }
    public boolean f(B b) {
        return num == b.num;
    }
}

```

```

public class C extends A {
    public C(int n) {
        super(n);
    }
    public boolean f(A a){
        return a instanceof C && num == a.num;
    }
}

```

```

public class Driver {
    public static void main(String[ ] args){
        A y1 = new B(10);
        B y2 = new B(10);
        A z1 = new C(10);
        C z2 = new C(10);

        System.out.println(y1.num == y2.num);
        System.out.println(y1.getNum()==((B)z1).getNum());
        System.out.println(y1.f(y2));
        System.out.println(z1.f(z2));
        System.out.println(z1.f(y1));
        System.out.println(z2.f((C)y2));
    }
}

```

```

true
Run time Error : C cannot be cast to B
false
true
false
Compilation Error: Inconvertible
types required C , found B

```

**לאחר :** `System.out.println(y1.num == y2.num)`  
**תשובה :** `true` (המשתנים עם הרשאה `protected` לכן ניתן לגשת אליהם מאותו `package`)

**לאחר :** `System.out.println(y1.getNum()==((B)z1).getNum())`  
**תשובה:** ה `casting` נדחה לזמן הרצה כיוון ש `z1` הוא מצביע מסוג `A` אשר נמצא מעל ל `B` ו- `C` לכן זה עובר קומפילציה אך בזמן ההרצה יש ניסיון להמיר את האובייקט ש `z1` מצביע עליו שהוא מטיפוס `C` לטיפוס `B`.  
בין `B` ל `C` אין קישר של הורשה ולכן מתקבלת שגיאה בזמן ההרצה עם ההודעה: `C cannot cast to B`

**לאחר :** `System.out.println(y1.f(y2))`  
**התוצאה :** ההחלטה לאיזה פונקציה הוא פונה נדחית לזמן ההרצה. כיוון ש `y1` הוא מצביע מסוג `A` הוא מפעיל את `f` שנמצא בראש ההיררכיה (אם ישנו) כלומר במחלקה `A`. הוא מגלה שיש ואז הוא בודק אם יש דריסה לפונקציה שבמחלקה `B` והוא מגלה שאין דריסה (יש העמסה עם פרמטרים שונים) ולכן מפעיל בלית ברירה את זו שיש לו. ולכן מחזיר `False` כי `10 == 20` לא מתקיים.

**שאלה :** מה יודפס אם ב `A` לא הייתה מוגדרת הפונקציה `f` ו `B` נשאר כמו שהו?  
**תשובה :** `y1` מגלה כבר בזמן קומפילציה שאין פונקציה `f` ולכן נותן הודעה על שגיאת קומפילציה.

**שאלה :** מה היה מודפס אם ב `A` לא הינו מבצעים שינוי ואילו ב- `B` היו הפונקציות הבאות :

```
public boolean f(A b) {  
    return num == b.num*10;  
}  
public boolean f(B b) {  
    return num == b.num;  
}
```

**תשובה :** הוא פונה לשיטה `public boolean f(A b)` אשר ב `B` ומחזיר `False`

**שאלה :** מה יודפס אם A ו- B נשארים כמו שהם אבל הפקודה מה main היא :  
`System.out.println(y2.f(y2));`

**תשובה :** מופעלת השיטה f שיש ב B אשר מחזירה true :

```
public boolean f(B b) {  
    return num == b.num;  
}
```

**לאחר :** `System.out.println(z1.f(z2))`

**תשובה:** z1 מפעיל את f שנמצאת ב A . זו בודקת אם יש פונקציה זהה לה (דריסה) ב C . היא מגלה שכן ואז מפעילה אותה. השיטה שב C בודקת אם האובייקט שנשלח אליה כפרמטר (a=z2) הוא מסוג של C ע"י שימוש בפקודה `a instanceof C` ; ומגלה שכן ואז בודקת אם הערכים של שניהם שווים וגם זה נכון ולכן תחזיר Ture

**לאחר :** `System.out.println(z1.f(y1))`

**תשובה:** z1 מפעיל את f שנמצאת ב A . זו בודקת אם יש פונקציה זהה לה (דריסה) ב C . היא מגלה שכן ואז מפעילה אותה. השיטה שב C בודקת אם האובייקט שנשלח אליה כפרמטר (a=y1) הוא מסוג של C ע"י שימוש בפקודה `a instanceof C` ; ומגלה שלא ולכן מחזירה False

**לאחר :** `System.out.println(z2.f((C)y2))`

**תשובה :** שגיאה בזמן קומפילציה כיוון שיישנו נסיון להמיר אובייקט מסוג B ל C וזה לא חוקי.

**שימו לב יש כמעט אותו דבר בסעיף ב. רק ששם יש משתנה מסוג A (z1) אשר מתבקש להתבצע עליו casting וכיוון ש A הוא בראש ההיררכיה הפעולה של ה casting נידחת לזמן הרצה כי זה יכול להשתנות במהלך התוכנית . בעוד שכאן y2 תמיד יישאר מסוג B לאורך כל התוכנית.**

**שאלה :** מה יודפס אם A ו- B נשארים כמו שהם אבל הפקודה מה main היא :  
`System.out.println(y2.f(y2));`

**תשובה :** מופעלת השיטה f שיש ב B אשר מחזירה true :

```
public boolean f(B b) {  
    return num == b.num;  
}
```

**לאחר :** `System.out.println(z1.f(z2))`

**תשובה:** z1 מפעיל את f שנמצאת ב A . זו בודקת אם יש פונקציה זהה לה (דריסה) ב C . היא מגלה שכן ואז מפעילה אותה. השיטה שב C בודקת אם האובייקט שנשלח אליה כפרמטר (a=z2) הוא מסוג של C ע"י שימוש בפקודה a instanceof C ; ומגלה שכן ואז בודקת אם הערכים של שניהם שווים וגם זה נכון ולכן תחזיר True

**לאחר :** `System.out.println(z1.f(y1))`

**תשובה:** z1 מפעיל את f שנמצאת ב A . זו בודקת אם יש פונקציה זהה לה (דריסה) ב C . היא מגלה שכן ואז מפעילה אותה. השיטה שב C בודקת אם האובייקט שנשלח אליה כפרמטר (a=y1) הוא מסוג של C ע"י שימוש בפקודה a instanceof C ; ומגלה שלא ולכן מחזירה False

**לאחר :** `System.out.println(z2.f((C)y2))`

**תשובה :** שגיאה בזמן קומפילציה כיוון שיישנו נסיון להמיר אובייקט מסוג B ל C וזה לא חוקי.

**שימו לב יש כמעט אותו דבר בסעיף ב. רק ששם יש משתנה מסוג A (z1) אשר מתבקש להתבצע עליו casting וכיוון ש A הוא בראש ההיררכיה הפעולה של ה casting נידחת לזמן הרצה כי זה יכול להשתנות במהלך התוכנית . בעוד שכאן y2 תמיד יישאר מסוג B לאורך כל התוכנית.**

בירושה כאשר אנו מבצעים דריסה ( לא העמסה) , מותר לשנות הרשאה של השיטה הנדרסת להרשאה גבוהה מההרשאה המקורית. כלומר אם במקורית יש public אז בהרשאה של השיטה במחלקה הנגזרת צריכה להיות גם public . אם במקורית היה protected אז בנדרסת יכול להיות protected או public בלבד ולא private :

א : מקרה זה תקין

```
public class A
{
    private void f1(){}
}
public class B extends A
{
    public void f1() {}
}
```

ב: לא תקין

```
public class A
{
    public void f1(){}
}
public class B extends A
{
    private void f1() {}
}
```

ג : תקין כי זה העמסה ולא ירושה :

```
public class A
{
    public void f1(){}
}
public class B extends A
{
    private void f1(int x) {}
}
```



**מחלקה יכולה לממש יותר ממשק.**

**כמו שהסברנו בכיתה גם אם בממשקים יש שיטה משותפת זה לא יוצר התנגשות  
במחלקה הממשת . לדוגמה :**

```
public interface I1 {  
    public void f1();  
}
```

```
public interface I2 {  
    public void f1();  
}
```

```
public class A implements I1,I2 {  
    public void f1(){ }  
}
```



***END***