



Java™

КЛАСОВЕ

Съдържание

- Обекти, класове и ООП
 - Връзка м/у класове и обекти
 - абстракция
- Анатомия на класовете
 - полета
 - инстанции на методите
 - конструктори
 - encapsulation
- Допълнителна информация за класовете!
 - preconditions, postconditions, and invariants
 - Специални методи: `toString` and `equals`
 - Служебната дума `this`

Класове, типове и обекти

- **КЛАС:**

1. Понятие 1: Модул, който може да се стартира като програма.
2. Понятие 2: „Образец“ за даден тип обект.

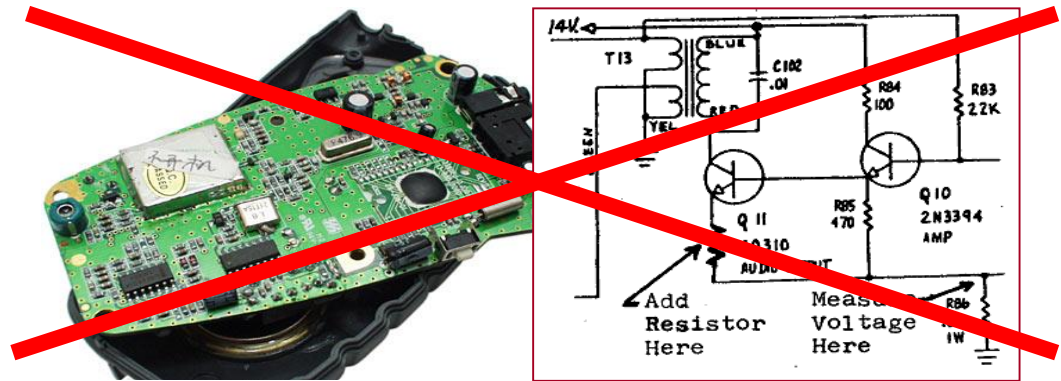
- Може да създаваме класове, които да не са изпълними, а в тях да се дефинират нови типове и обекти.
 - Ние може да използваме тези обекти в нашите програми.
- Защо това се използва?

Обекти и ООП

- **обект:** използва се за капсулиране на данните и поведението.
- **ООП:** Създават се програми, в които се описват взаимодействието на обектите.
- Досега сте използвали следните обекти:
 - String
 - Point
 - Scanner
 - DrawingPanel
 - Graphics
 - Color
 - Random
 - File
 - PrintStream

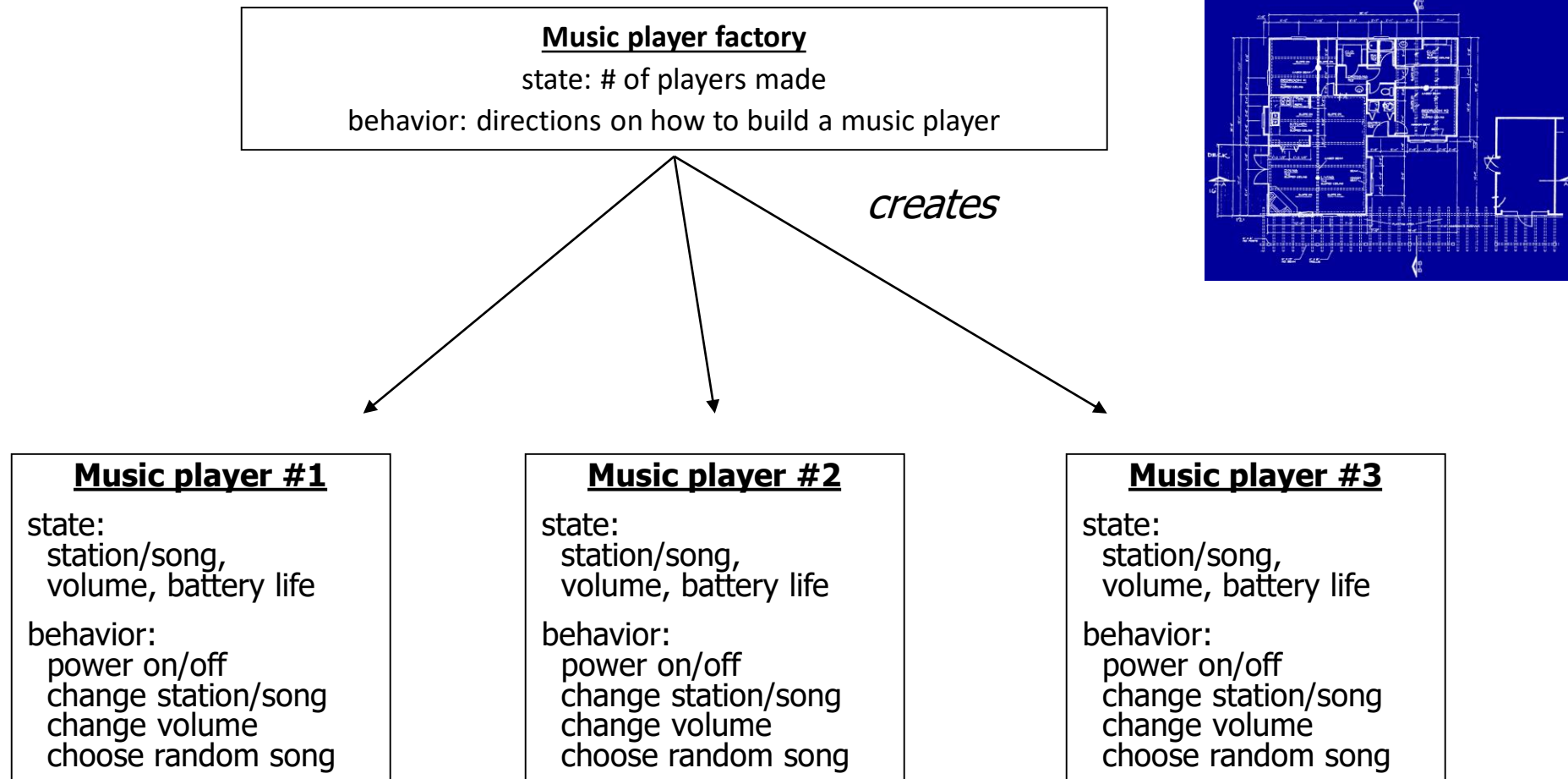
Абстракция

- **Абстракция:** Разликата между идеите и детайлите
 - Обектите в Java дават възможност за ниво на абстракции, защото ние ги използваме без да знаем как функционират.
- Вие използвате абстракции всеки ден, когато си взаимодействате с технологични обекти като например portable music player.
 - Вие разбирате тяхното външно поведение (buttons, screen, etc.)
 - Не знаете как работи, а и не е нужно за да ползвате устройството.



Factory/blueprint analogy

- В действителния живот, с factory може да се създадат много подобни обекти. Аналогия на индиго едно време.



Recall: Point objects

- Java has a class of objects named `Point`.
 - To use `Point`, you must write: `import java.awt.*;`

- Constructing a `Point` object, general syntax:

```
Point <name> = new Point(<x>, <y>);
```

```
Point <name> = new Point(); // the origin, (0, 0)
```

- Examples:

```
Point p1 = new Point(5, -2);
```

```
Point p2 = new Point();
```

- `Point` objects are useful for several reasons:
 - They store two values, an (x, y) pair, in a single variable.
 - They have useful methods we can call in our programs.

Recall: Point data/methods

- Data stored in each `Point` object:

Field name	Description
<code>x</code>	the point's x-coordinate
<code>y</code>	the point's y-coordinate

- Useful methods of each `Point` object:

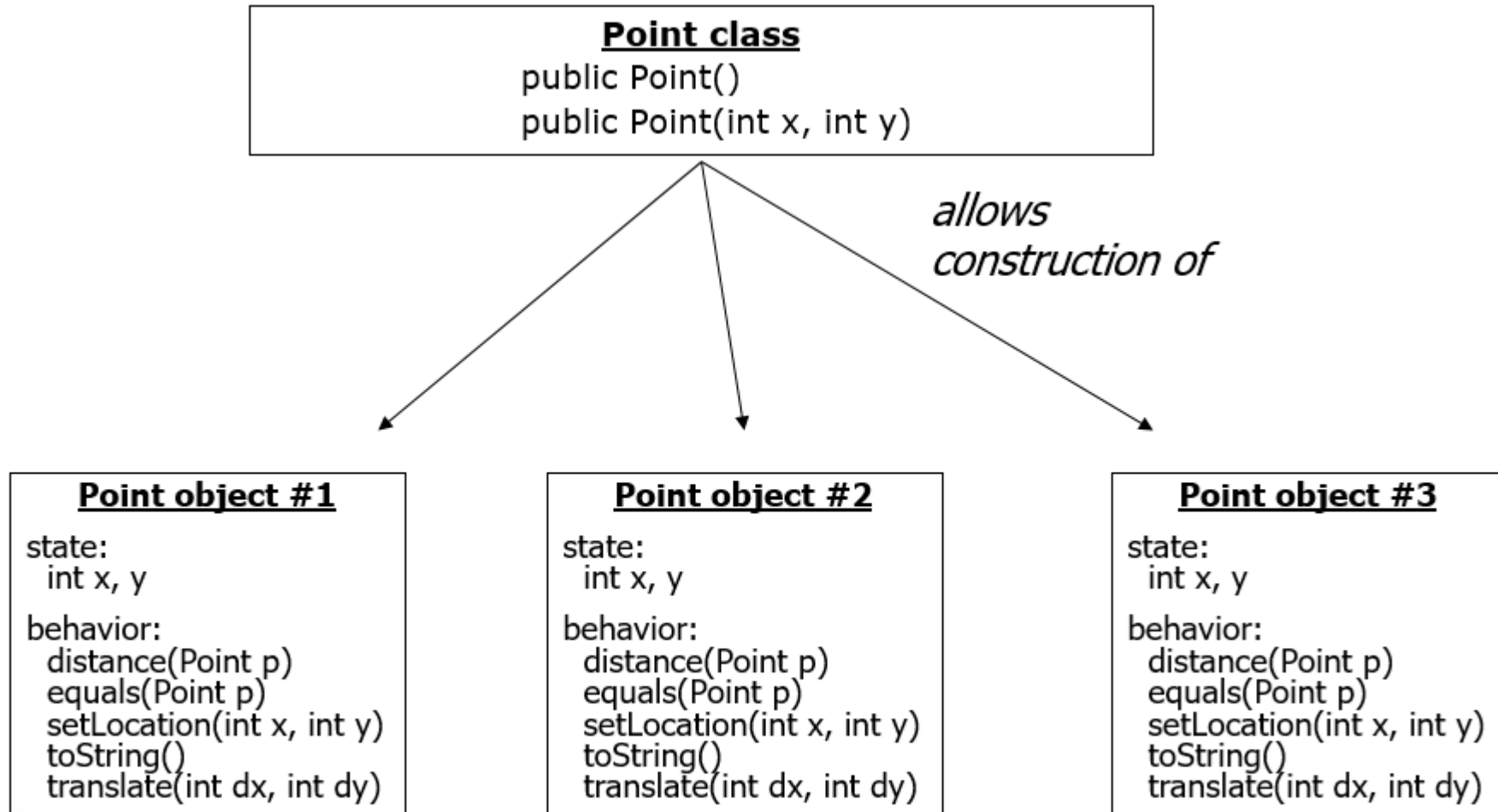
Method name	Description
<code>distance(<i>p</i>)</code>	how far away the point is from point <i>p</i>
<code>setLocation(<i>X</i>, <i>y</i>)</code>	sets the point's x and y to the given values
<code>translate(<i>dx</i>, <i>dy</i>)</code>	adjusts the point's x and y by the given amounts

- `Point` objects can also be printed using `println` statements:

```
Point p = new Point(5, -2);
```

```
System.out.println(p);    // java.awt.Point[x=5,y=-2]
```


Клас POINT



Object state:
fields

Point class, версия 1

- Създава се нов клас Point.

```
public class Point {  
    int x;  
    int y;  
}
```

- Записваме го във файла Point.java.
- Всеки обект съдържа две части:
 - int x,
 - int y.
- Point нямат поведение.

Поleta

- **поле**: Променливата в обекта, която представя част от състоянието на обекта.
 - Всеки обект има свое копие от даното поле.
- Декларация на поле:
<type> <name> ;
 - Пример:

```
public class Student {  
    String name;      // всеки обект студент има  
    double gpa;       // поле name и gpa.  
}
```

Поле за достъп

В кода на други класове може да достъпваме полето на обекта.

Поле за достъп, синтаксис:

<variable name> . <field name>

Модифициране на полето, синтаксис:

<variable name> . <field name> = <value> ;

Примери:

```
System.out.println("the x-coord is " +  
p1.x);      // модификатор  
p2.y = 13;  // за достъп
```

До тук научихте, че *encapsulation*, която променя начина, по които достъпваме данните вътре в обекта.

Client code

- **client code:** кодът, който използва обекта.
 - По-долу е даден client code (PointMain.java) и той използва Point класа.

```
public class PointMain {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point();  
        p1.x = 5;  
        p1.y = 2;  
        Point p2 = new Point();  
        p2.x = 4;  
        p2.y = 3;  
  
        // print each point  
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
  
        // move p2 and then print it again  
        p2.x += 2;  
        p2.y += 4;  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
    }  
}
```

OUTPUT:

```
p1 is (5, 2)  
p2 is (4, 3)  
p2 is (6, 7)
```

ЗАДАЧА:

Напишете клиентска програма, която да използва класа `Point` и да получи резултат:

```
p1 is (7, 2)
p1's distance from origin = 7.280109889280518
p2 is (4, 3)
p2's distance from origin = 5.0
p1 is (18, 8)
p2 is (5, 10)
```

Използвайте формулата за изчисление на разликата между две точки (x_1, y_1) и (x_2, y_2) :

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Клиентска програма – излишен код

Нашата клиентска програма има код който дава мястото на обекта от класа Point.

```
// move p2 and then print it again
p2.x += 2;
p2.y += 4;
System.out.println("p2 is (" + p2.x + ", " +
p2.y + ")");
```

Ако преместим няколко пъти точките, то горния код ще се повтори няколко пъти в клиентската програма.

**Object behavior:
instance methods**

Премахване на излишния код – В. 1

- Може да се съкрати кода като се използва static метод в клиентската част, по-точно в изчислението на координатите:

```
// Shifts the location of the given point.  
public static void translate(Point p, int dx, int dy) {  
    p.x += dx;  
    p.y += dy;  
}
```

- Защо метода не връща модифицираната стойност?
- Клиента ще извика метода по следния начин:

```
// move p2 and then print it again  
translate(p2, 2, 4);  
System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
```

Class behavior

- Използването на статичен метод не е добро решение:
 - Синтаксиса не отговаря на начина, по който ще ползваме обектите.
`translate(p2, 2, 4);`
 - Цялостната идея за създаването на класовете е да им се зададат съответно състояние и поведение. Това поведение е свързано с данните `x, y` за обектите от клас `point`.
- Обектите, които използваме ще съдържат поведението в самите себе си.
 - Когато искаме да използваме това поведение, ще извикваме метод на обекта.
`// move p2 and then print it again`
`p2.translate(2, 4);`
`System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");`
- Нека да разгледаме как се добавя метод към обект от класа `Point`.

Instance methods

- **instance method**: метод(без static), дефиниращ поведението на обектите.
 - Този обект може да има референция към свои полета и методи ако е необходимо.

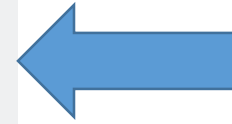
- instance method декларация, синтаксис:

```
public <type> <name> ( <parameter(s)> ) {  
    <statement(s)> ;  
}
```

- Пример (този код е вътре в класа Point):

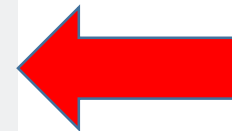
```
public void translate(int dx, int dy) {  
    ...  
}
```

```
class Math(){  
    public static long multiply(long a, long b){  
        return a*b;  
    }  
    public static void main(String[]args){  
        System.out.println(Math.multiply());  
    }  
}
```



Static
method

```
class User(){  
    private String pass;  
    private String uname;  
  
    public User(String p,String u){  
        pass=p;  
        uname=u;  
    }  
    public boolean authenticate(){  
        if("secret".equals(this.pass) && "Grrrr".equals(this.uname){  
            return true;  
        }else{  
            return false;  
        }  
    }  
    public static void main(String[]args){  
        User u = new User("wrong secret","grrr");  
        System.out.println(u.authenticate());  
    }  
}
```



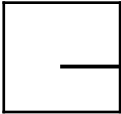
Instance
method

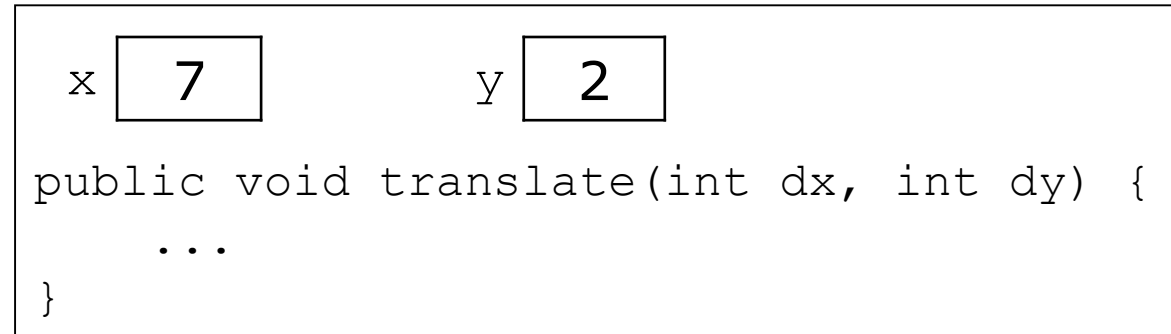
Диаграма за обекти от клас Point

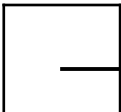
- За всеки обект от `Point` има копие на метода `translate` с операнди стойностите на обект:

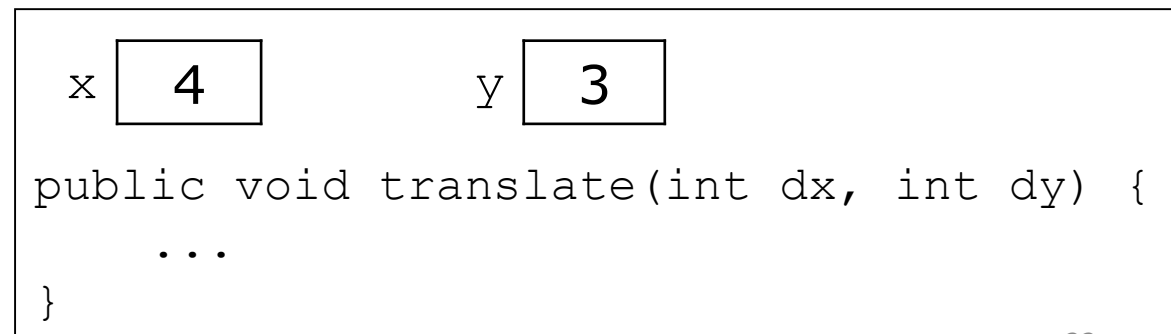
```
Point p1 = new Point();  
p1.x = 7;  
p1.y = 2;
```

```
Point p2 = new Point();  
p2.x = 4;  
p2.y = 3;
```

p1 



p2 



The implicit parameter

- **implicit parameter:** Обектът, който инстантния метод извиква.
 - Инстантния метод извиква отделните обекти:
 - During the call `p1.translate(11, 6);`, the object referred to by `p1` is the implicit parameter.
 - During the call `p2.translate(1, 7);`, the object referred to by `p2` is the implicit parameter.
 - Инстантни методи могат да имат референция към полета на обектите.
(Казва се, че на тези методи кода се изпълнява в обръщението към самите обекти.)
- В резултат на по-горното `translate` метода ще е следния:

```
public void translate(int dx, int dy) {  
    x += dx;  
    y += dy;  
}
```

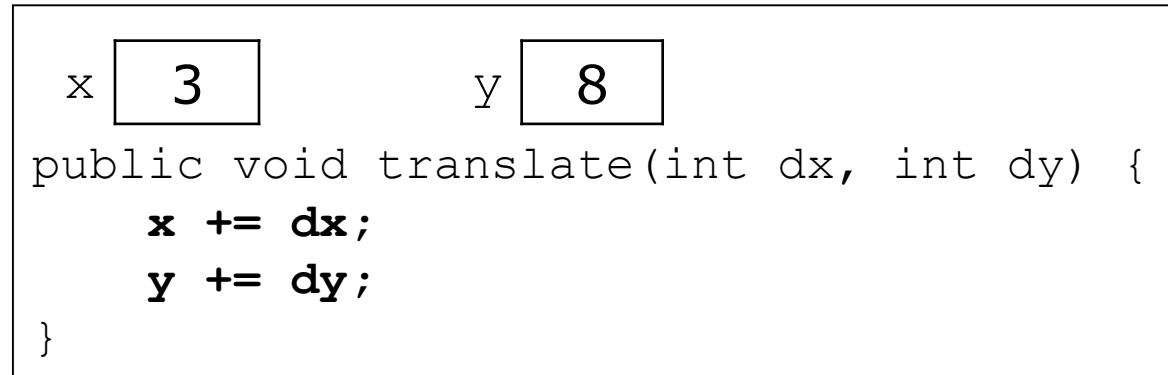
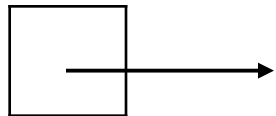
Tracing instance method calls

- Как се изпълняват извикванията от различни обекти на инстания метод?

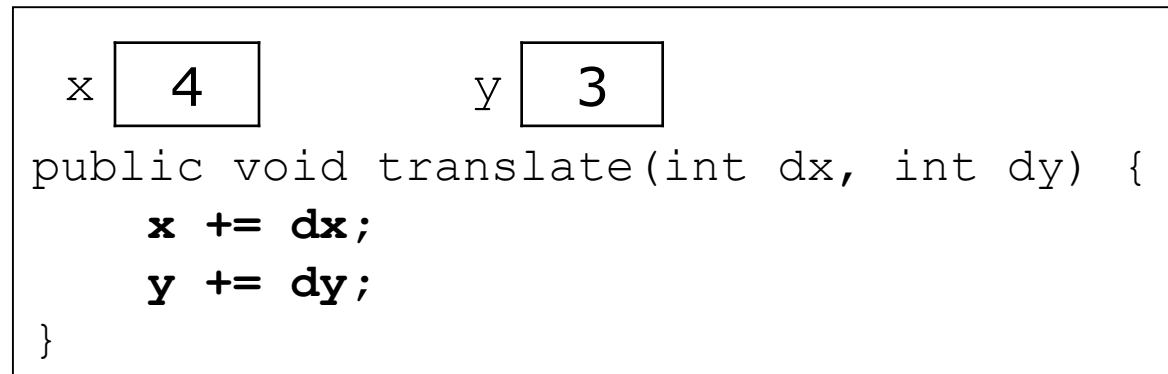
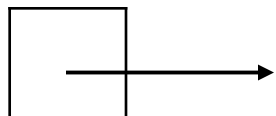
```
p1.translate(11, 6);
```

```
p2.translate(1, 7);
```

p1



p2



Клас Point, версия 2

- В тази версия за всеки обект от класа има метод `translate`:

```
public class Point {  
    int x;  
    int y;  
  
    // Changes the location of this Point object.  
    public void translate(int dx, int dy) {  
        x += dx;  
        y += dy;  
    }  
}
```

- Всеки обект от класа вече съдържа метод с поведение, което модифицира неговите `x` и `y` координати зададени като параметър.

Instance method questions

- Създайте инстантния метод `distanceFromOrigin`, който да изчислява и връща разликата `m/y` текущия обект `Point` и оригиналния, `(0, 0)`.

Използвайте следната формула: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- Създайте инстантния метод `distance`, който да приема като параметър обект от `Point` и изчислете разликата `m/y` него и текущия `Point`.
- Създайте инстантния метод `setLocation`, който да приема стойностите `x` и `y` за параметри и променя мястото на `Point` спрямо тези стойности.
- Може да искате да refactor този клас `Point` за да използва този метод.
- Променете клиентската програма така че да използва тези методи.

Accessor and mutators

Two common categories of instance methods:

- **accessor:** A method that provides access to information about an object.
 - Generally the information comes from (or is computed using) the object's state stored in its fields.
 - The `distanceFromOrigin` and `distance` methods are accessors.
- **mutator:** A method that modifies an object's state.
 - Sometimes the modification is based on parameters that are passed to the mutator method, such as the `translate` method with parameters for `dx` and `dy`.
 - The `translate` and `setLocation` methods are mutators.

Клиентска програма – в. 2

- Клиентската част (записва се в `PointMain2.java`) използва модифицирания клас `Point`:

```
public class PointMain2 {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point();  
        p1.x = 5;  
        p1.y = 2;  
        Point p2 = new Point();  
        p2.x = 4;  
        p2.y = 3;  
  
        // print each point  
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
    }  
}
```

OUTPUT:

```
p1 is (5, 2)  
p2 is (4, 3)  
p2 is (6, 7)
```

Въпроси:

- Извикването отнова на клиентската програма ще има следния резултат:
p1 is (7, 2)
p1's distance from origin = 7.280109889280518
p2 is (4, 3)
p2's distance from origin = 5.0
p1 is (18, 8)
p2 is (5, 10)
- Модифицирайте програмата, така че да използвате нов инстантен метод.
Освен това да изписва следния резултат:
distance from p1 to p2 = 3.1622776601683795

Object initialization: constructors

Initializing objects

- It is tedious to have to construct an object and assign values to all of its data fields manually.

```
Point p = new Point();  
p.x = 3;  
p.y = 8;                                // tedious
```

- We'd rather be able to pass in the fields' values as parameters, as we did with Java's built-in `Point` class.

```
Point p = new Point(3, 8);  // better!
```

- To do this, we need to learn about a special type of method called a *constructor*.

Constructors

- **constructor:** Initializes the state of new objects.
 - Constructors may accept parameters to initialize the object.
 - A constructor looks like a method, but it doesn't specify a return type, because it implicitly returns a new `Point` object.

- Constructor syntax:

```
public <type> ( <parameter(s)> ) {  
    <statement(s)> ;  
}
```

- Example:

```
public Point(int initialX, int initialY) {  
    ...  
}
```


Клас Point, версия 3

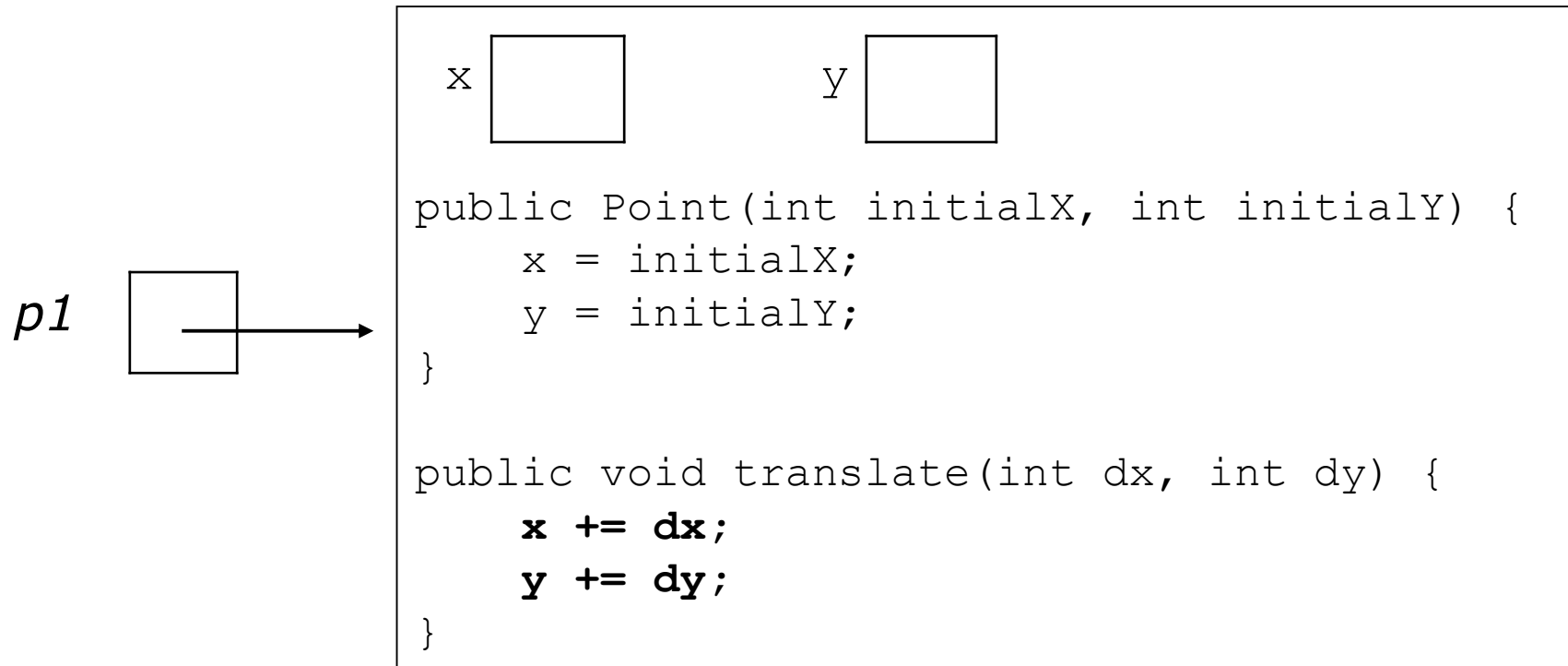
- This third version of the `Point` class provides a constructor to initialize `Point` objects:

```
public class Point {  
    int x;  
    int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public void translate(int dx, int dy) {  
        x += dx;  
        y += dy;  
    }  
}
```

Constructors

- What happens when the following call is made?

`Point p1 = new Point(7, 2);`



Version 3

- The following client code (stored in `PointMain3.java`) uses our `Point` constructor:

```
public class PointMain3 {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point(5, 2);  
        Point p2 = new Point(4, 3);  
  
        // print each point  
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");  
    }  
}
```

OUTPUT:

```
p1 is (5, 2)  
p2 is (4, 3)  
p2 is (6, 7)
```

Client code question

- Recall our client program that produces this output:

```
p1 is (7, 2)
```

```
p1's distance from origin = 7.280109889280518
```

```
p2 is (4, 3)
```

```
p2's distance from origin = 5.0
```

```
p1 is (18, 8)
```

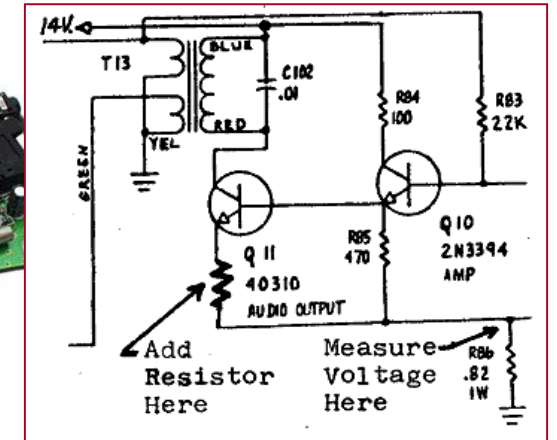
```
p2 is (5, 10)
```

- Modify the program to use our new constructor.

Encapsulation

Encapsulation

- **encapsulation:** Hiding the implementation details of an object from the clients of the object.
 - (Protecting the object's fields from modification by clients.)
- Encapsulating objects provides *abstraction*; we can use them without knowing how they work. The object has:
 - an external view (its behavior)
 - an internal view (the state that accomplishes the behavior)



Implementing encapsulation

- Fields can be declared *private* to indicate that no code outside their own class can change them.

- Declaring a private field, general syntax:

```
private <type> <name> ;
```

- Examples:

```
private int x;
```

```
private String name;
```

- Once fields are private, client code cannot directly access them. The client receives an error such as:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                        ^
```

Encapsulation and accessors

- Once fields are private, we often provide accessor methods to examine their values:

```
public int getX() {  
    return x;  
}
```

- This gives clients "read-only" access to the object's fields.
- If so desired, we can also provide mutator methods:

```
public void setX(int newX) {  
    x = newX;  
}
```
- Question: Is there any difference between a public field and a private field with a `get` and `set` method?

Benefits of encapsulation

- Encapsulation helps provide a clean layer of abstraction between an object and its clients.
- Encapsulation protects an object from unwanted access by clients.
 - For example, perhaps we write a program to manage users' bank accounts. We don't want a malicious client program to be able to arbitrarily change a `BankAccount` object's balance.
- Encapsulation allows the class author to change the internal representation later if necessary.
 - For example, if so desired, the `Point` class could be rewritten to use polar coordinates (a radius r and an angle ϑ from the origin), but the external view could remain the same.

Клас Point Версия 4

```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```

Preconditions,
postconditions, and
invariants

Pre/postconditions

- **precondition:** Something that you assume to be true when your method is called.
- **postcondition:** Something you promise to be true when your method exits.
 - Pre/postconditions are often documented as comments.

- Example:

```
// Sets this Point's location to be the given (x, y).  
// Precondition: newX >= 0 && newY >= 0  
// Postcondition: x >= 0 && y >= 0  
public void setLocation(int newX, int newY) {  
    x = newX;  
    y = newY;  
}
```

Class invariants

- **class invariant:** An assertion about an object's state that is true throughout the lifetime of the object.
 - An invariant can be thought of as a postcondition on every constructor and mutator method of a class.
 - Example: "No BankAccount object's balance can be negative."
 - Example: "The speed of a SpaceShip object must be ≤ 10 ."
- Example: Suppose we want to ensure that all `Point` objects' x and y coordinates are never negative.
 - We must ensure that a client cannot construct a `Point` object with a negative x or y value.
 - We must ensure that a client cannot move an existing `Point` object to a negative (x, y) location.

Violated preconditions

- What if your precondition is not met?
 - Sometimes the client passes an invalid value to your method.

- Example:

```
Point pt = new Point(5, 17);  
Scanner console = new Scanner(System.in);  
System.out.print("Type the coordinates: ");  
int x = console.nextInt();    // what if the user types  
int y = console.nextInt();    // a negative number?  
pt.setLocation(x, y);
```

- How can we prevent the client from misusing our object?

Dealing with violations

- One way to deal with this problem would be to return out of the method if negative values are encountered.
 - However, it is not possible to do something similar in the constructor, and the client doesn't expect this behavior.
- A more common solution is to have your object *throw an exception*.
- **exception:** A Java object that represents an error.
 - When a precondition of your method has been violated, you can generate ("throw") an exception in your code.
 - This will cause the client program to halt. (That'll show 'em!)

Throwing exceptions example

- Throwing an exception, general syntax:

```
throw new <exception type> ();
```

```
or throw new <exception type> ("<message>") ;
```

- The **<message>** will be shown on the console when the program crashes.

- Example:

```
// Sets this Point's location to be the given (x, y).  
// Throws an exception if newX or newY is negative.  
// Postcondition: x >= 0 && y >= 0  
public void setLocation(int newX, int newY) {  
    if (newX < 0 || newY < 0) {  
        throw new IllegalArgumentException();  
    }  
  
    x = newX;  
    y = newY;  
}
```


Encapsulation and invariants

Encapsulation helps you enforce invariants.

- Ensure that no `Point` is constructed with negative `x` or `y`:

```
public Point(int initialX, int initialY) {  
    if (initialX < 0 || initialY < 0) {  
        throw new IllegalArgumentException();  
    }  
    x = initialX;  
    y = initialY;  
}
```

- Ensure that no `Point` can be moved to a negative `x` or `y`:

```
public void translate(int dx, int dy) {  
    if (x + dx < 0 || y + dy < 0) {  
        throw new IllegalArgumentException();  
    }  
    x += dx;  
    y += dy;  
}
```

- Other methods require similar modifications.

**Special instance
methods:
toString,
equals**

Problem: object printability

- By default, Java doesn't know how to print the state of your objects, so it prints a strange result:

```
Point p = new Point(10, 7);  
System.out.println("p is " + p); // p is Point@9e8c34
```

- We can instead print a more complex string that shows the object's state, but this is cumbersome.

```
System.out.println("(" + p.x + ", " + p.y + ")");
```

- We'd like to be able to print the object itself and have something meaningful appear.

```
// desired behavior:  
System.out.println("p is " + p); // p is (10, 7)
```

Методът toString

- The special method `toString` tells Java how to convert your object into a `String` as needed.
 - The `toString` method is called when your object is printed or concatenated with a `String`.

```
Point p1 = new Point(7, 2);  
System.out.println("p1 is " + p1);
```
 - If you prefer, you can write the `.toString()` explicitly.

```
System.out.println("p1 is " + p1.toString());
```
- Every class contains a `toString` method, even if it isn't written in your class's code.
 - The default `toString` behavior is to return the class's name followed by a hexadecimal (base-16) number:

```
Point@9e8c34
```

Синтаксис на метода toString

- You can replace the default behavior by defining an appropriate `toString` method in your class.
 - Example: The `Point` class in `java.awt` has a `toString` method that converts a `Point` into a `String` such as: `"java.awt.Point[x=7,y=2]"`
- The `toString` method, general syntax:

```
public String toString() {  
    <statement(s) that return an appropriate String> ;  
}
```

 - The method must have this exact name and signature.
- Example:

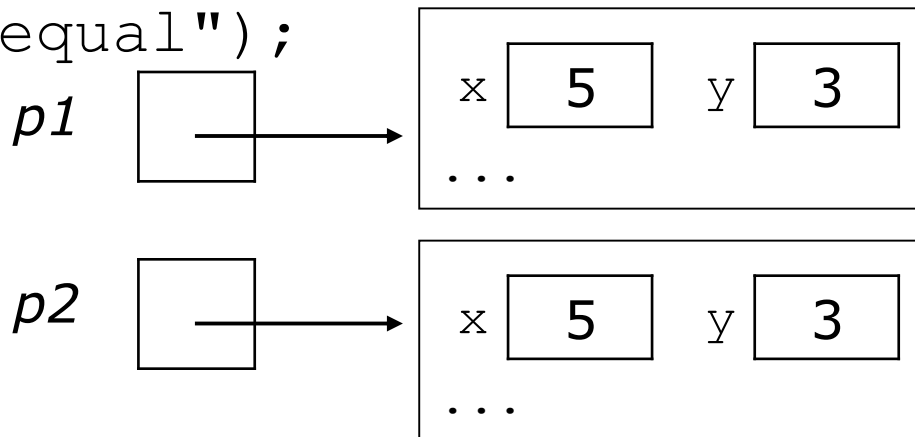
```
// Returns a String representing this Point.  
public String toString() {  
    return "(" + x + ", " + y + ")";  
}
```

Recall: comparing objects

- The `==` operator does not work well with objects.
 - `==` compares references to objects and only evaluates to `true` if two variables refer to the same object.
 - It doesn't tell us whether two objects have the same state.

- Example:

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
if (p1 == p2) { // false  
    System.out.println("equal");  
}
```



The equals method

- The `equals` method compares the state of objects.
 - When we write our own new classes of objects, Java doesn't know how to compare their state.
 - The default `equals` behavior acts just like the `==` operator.

```
if (p1.equals(p2)) { // still false
    System.out.println("equal");
}
```

- We can replace this default behavior by writing an `equals` method.
 - The method will actually compare the state of the two objects and return `true` for cases like the above.

Initial flawed equals method

- You might think that the following is a valid implementation of the `equals` method:

```
public boolean equals(Point other) {  
    if (x == other.x && y == other.y) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- However, it has several flaws that we should correct.
- One initial flaw: the body can be shortened to:

```
return x == other.x && y == other.y;
```


equals and the Object class

- `equals` should not accept a parameter of type `Point`.
 - It should be legal to compare `Points` to any other object, e.g.:

```
Point p = new Point(7, 2);  
if (p.equals("hello")) {    // false  
    ...  
}
```

- The `equals` method, general syntax:

```
public boolean equals(Object <name>) {  
    <statement(s) that return a boolean value> ;  
}
```

- The parameter to a proper `equals` method must be of type `Object` (meaning that an object of any type can be passed).

Another flawed version

- You might think that the following is a valid implementation of the equals method:

```
public boolean equals(Object o) {  
    if (x == o.x && y == o.y) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- However, it does not compile.

```
Point.java:36: cannot find symbol  
symbol   : variable x  
location: class java.lang.Object  
if (x == o.x && y == o.y) {  
           ^
```

Type-casting objects

- The object that is passed to `equals` can be cast from `Object` into your class's type.

- Example:

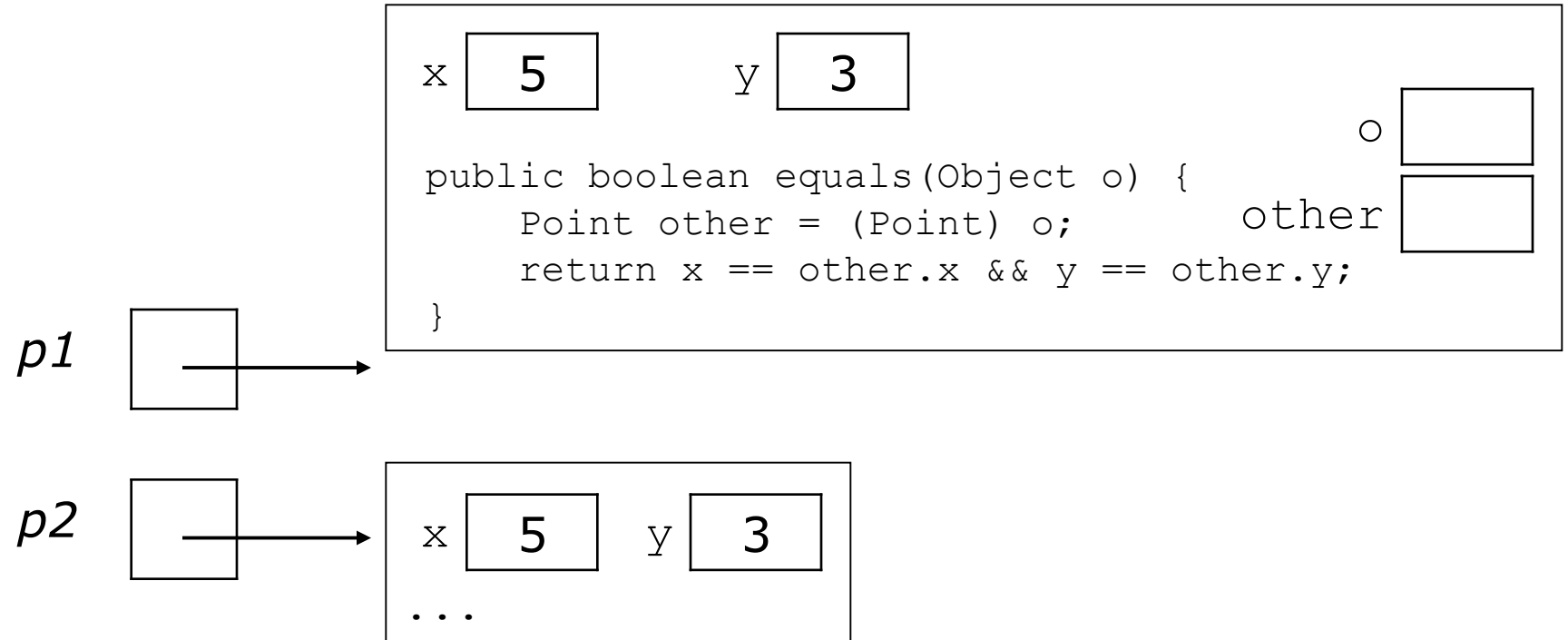
```
public boolean equals(Object o) {  
    Point other = (Point) o;  
    return x == other.x && y == other.y;  
}
```

- Type-casting with objects behaves differently than casting primitive values.
 - We are really casting a reference of type `Object` into a reference of type `Point`.
 - We're promising the compiler that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
if (p1.equals(p2)) {  
    System.out.println("equal");  
}
```



Comparing different types

- Our equals code still is not complete.
 - When we compare Point objects to any other type of objects,

```
Point p = new Point(7, 2);  
if (p.equals("hello")) {    // false  
    ...  
}
```
 - Currently the code crashes with the following exception:
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
 at Point.equals(Point.java:25)
 at PointMain.main(PointMain.java:25)
 - The culprit is the following line that contains the type-cast:

```
public boolean equals(Object o) {  
    Point other = (Point) o;
```

Comparing different types

- Our `equals` code still is not complete.
 - When we compare `Point` objects to any other type of objects,

```
Point p = new Point(7, 2);  
if (p.equals("hello")) {    // false  
    ...  
}
```
 - Currently the code crashes with the following exception:
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
 at Point.equals(Point.java:25)
 at PointMain.main(PointMain.java:25)
 - The culprit is the following line that contains the type-cast:

```
public boolean equals(Object o) {  
    Point other = (Point) o;
```

The instanceof keyword

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.
 - The `instanceof` keyword, general syntax:
<variable> instanceof <type>
 - The above is a `boolean` expression that can be used as the test in an `if` statement.
 - Examples:
`String s = "hello";`
`Point p = new Point();`

expression	result
<code>s instanceof Point</code>	false
<code>s instanceof String</code>	true
<code>p instanceof Point</code>	true
<code>p instanceof String</code>	false
<code>null instanceof String</code>	false

Final version of equals method

- This version of the `equals` method allows us to correctly compare `Point` objects against any other type of object:

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object.
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    else {
        return false;
    }
}
```


Използване на служебната дума `this`

- Тази версия на метода `equals` позволява да се сравнят обекти `Point` с обекти от друг тип:

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object.
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    else {
        return false;
    }
}
```

Сенчести променливи

- **shadowed variable**: Полето, което покрива локалната променлива или параметрите в едно и също име.
 - Нормално е да е легално да има поне 2 променливи с еднакво поле на действие.
 - За да се избегне ситуацията сенчести променлив, ние наименоваме параметирте на setLocation newX и newY:

```
public void setLocation(int newX, int newY) {  
    if (newX < 0 || newY < 0) {  
        throw new IllegalArgumentException();  
    }  
    x = newX;  
    y = newY;  
}
```

Избягване на shadowing с this

- The `this` keyword lets us use the same names and still avoid shadowing:

```
public void setLocation(int x, int y) {  
    if (x < 0 || y < 0) {  
        throw new IllegalArgumentException();  
    }  
    this.x = x;  
    this.y = y;  
}
```

- Когато `this.` не се вижда, параметъра се използва.
- Когато `this.` се вижда, полето се използва.

Multiple constructors

- Съществуват повече от един конструктор в класа.
 - Конструкторът има различен брой параметри.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        x = 0;  
        y = 0;  
    }  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    ...  
}
```

Използване на множество конструктори <-> this

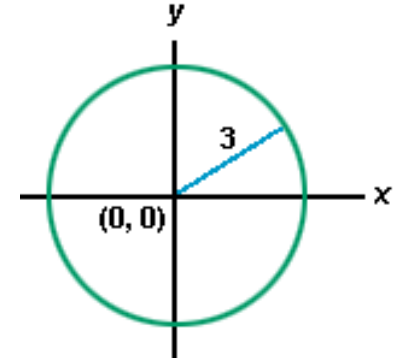
- Един конструктор може да извика друг с `this`.
 - Освен това може да се използва полето `this` за задаване на стойности на параметрите в конструктора.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0);    // calls the (x, y) constructor  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    ...  
}
```

Други проблеми
в КЛАСОВЕТЕ

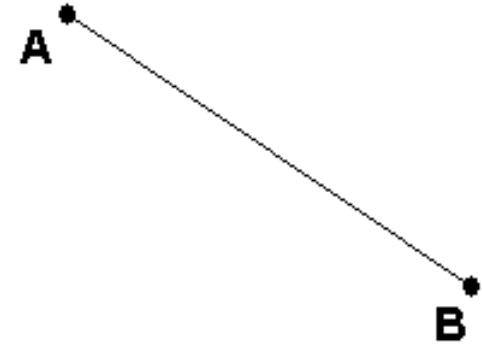
ЗАДАЧА 1

- Create a class named `Circle`.
 - A circle is represented by a point for its center, and its radius.
 - Make it possible to construct the unit circle, centered at $(0, 0)$ with radius 1, by passing no parameters to the constructor.
- Circles should be able to tell whether a given point is contained inside them.
- Circles should be able to draw themselves using a `Graphics`.
- Circles should be able to be printed on the console, and should be able to be compared to other circles for equality.



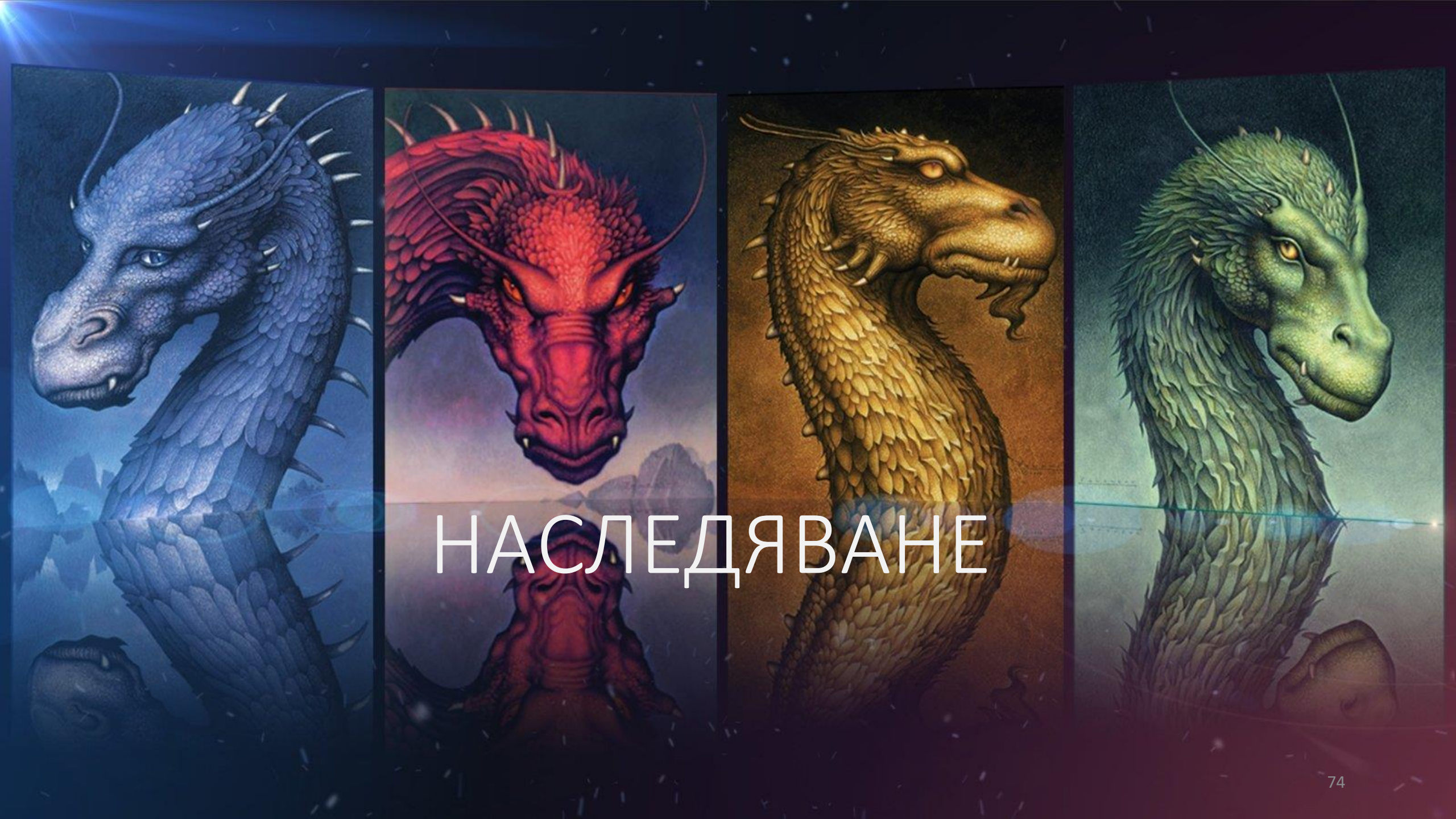
ЗАДАЧА 2

- Create a class named `LineSegment`.
 - A line segment is represented by two endpoints (x_1, y_1) and (x_2, y_2) .
 - A line segment should be able to compute its slope $(y_2 - y_1) / (x_2 - x_1)$.
 - A line segment should be able to tell whether a given point intersects it.
 - Line segments should be able to draw themselves using a `Graphics` object.
 - Line segments should be able to be printed on the console, and should be able to be compared to other lines for equality.



Пример – клиентска програма Калкулатор

```
public class CalculatorMain {  
    public static void main(String[] args) {  
        Calculator calc = new Calculator();  
  
        // first computation: calculate 329 + 1748 = 2077  
        calc.addDigit(3);  
        calc.addDigit(2);  
        calc.addDigit(9);  
  
        calc.setOperator("+");  
  
        calc.addDigit(1);  
        calc.addDigit(7);  
        calc.addDigit(4);  
        calc.addDigit(8);  
  
        int result = calc.compute();  
  
        System.out.println(calc);  
        System.out.println("result = " + result);  
    }  
}
```



НАСЛЕДЯВАНЕ

СЪДЪРЖАНИЕ:

- ОСНОВИ
 - Категории служители
 - Връзки и йерархии
- наследяване
 - Създаване на подкласове
 - Поведение при наследяването
 - Множествено наследяване
 - Взаимодействие с суперкласове, използвайки резервираната дума `super`
 - Дизайн и наследяване
- полиморфизъм
 - Проблеми "polymorphism mystery"
- интерфейси

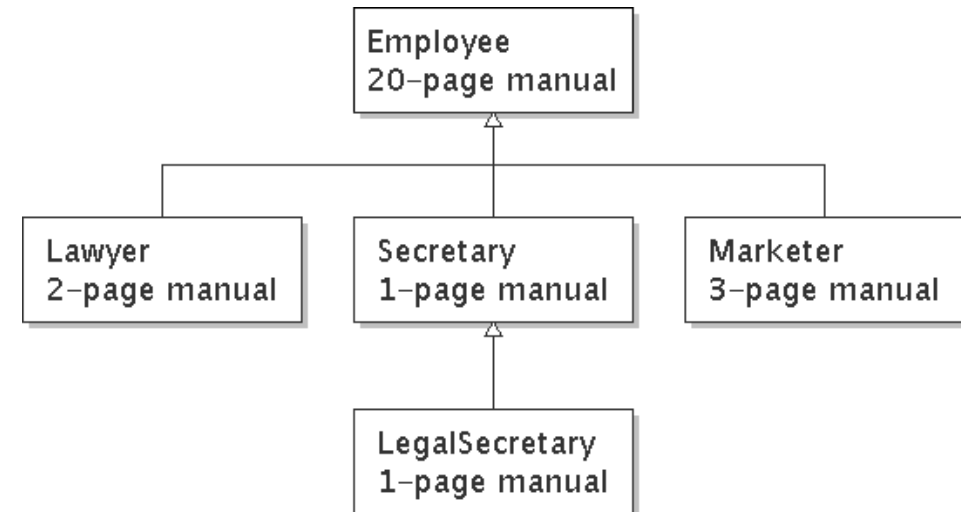
НАСЛЕДЯВАНЕ

- **СОФТУЕРНО ИНЖЕНЕРСТВО:** Използването на различни идеологии, като концептуализация, дизайн, разработка, документация и тестване на large-scale компютърни програми.
- Large-scale проекти:
 - Необходими са много програмисти
 - Създаване на кода навреме
 - Тестване на кода
 - Фиксиране на грешките
 - Поддръжка, подобряване и повторно използване на съществуващ код.
- **Повторното използване:** Практиката е следната – създава се код, който може да се използва в последствие многократно.

Аналогия с работниците

В една фирма за адвокати има различни видове служители.

- По подразбиране правилата: часове, почивни дни, бонуси, ...
 - Всички служители трябва да спазват общите правила
 - Всеки служител подписва договор, в който са указани неговите задължения.
- Всеки отдел има допълнителни правила и условия

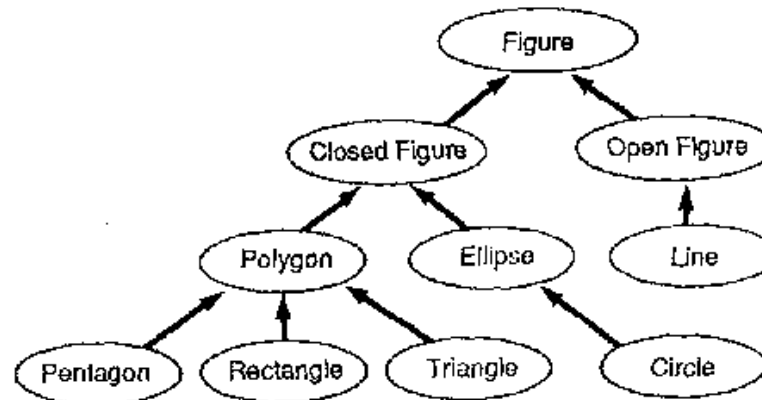


Separating behavior

- Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?
- Some advantages of the separate manuals:
 - maintenance: If a common rule changes, we'll need to update only the common manual.
 - locality: A person can look at the lawyer manual and quickly discover all rules that are specific to lawyers.
- Some key ideas from this example:
 - It's useful to be able to describe general rules that will apply to many groups (the 20-page manual).
 - It's also useful for a group to specify a smaller set of rules for itself, including being able to replace rules from the overall set.

Is-a relationships, hierarchies

- **is-a relationship:** A hierarchical connection where one category can be treated as a specialized version of another.
 - every marketer is an employee
 - every legal secretary is a secretary
- **inheritance hierarchy:** A set of classes connected by is-a relationships that can share common code.
 - Often drawn as a downward tree of connected boxes or ovals representing classes:



Employee regulations

- Consider the following employee regulations:
 - Employees work 40 hours per week.
 - Employees make \$40,000 per year, except legal secretaries who make \$5,000 extra per year (\$45,000 total), and marketers who make \$10,000 extra per year (\$50,000 total).
 - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
 - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.
- Each type of employee has some unique behavior:
 - Lawyers know how to sue.
 - Marketers know how to advertise.
 - Secretaries know how to take dictation.
 - Legal secretaries know how to prepare legal documents.

General employee code

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;          // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```

- Exercise: Implement class `Secretary`, based on the previous employee regulations.

Redundant secretary code

```
// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40;          // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;          // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Desire for code-sharing

- The `takeDictation` method is the only unique behavior in the `Secretary` class.
- We'd like to be able to say the following:

```
// A class to represent secretaries.  
public class Secretary {  
    <copy all the contents from Employee class.>  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

Inheritance

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
 - a way to group related classes
 - a way to share code between two or more classes
- We say that one class can *extend* another by absorbing its state and behavior.
 - **superclass**: The parent class that is being extended.
 - **subclass**: The child class that extends the superclass and inherits its behavior.
 - The subclass receives a copy of every field and method from its superclass.

Inheritance syntax

- Creating a subclass, general syntax:

```
public class <name> extends <superclass name> {
```

- Example:

```
public class Secretary extends Employee {  
    ....  
}
```

- By extending `Employee`, each `Secretary` object now:
 - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
 - can be treated as an `Employee` by any other code (seen later)
(e.g. a `Secretary` could be stored in a variable of type `Employee` or stored as an element of an `Employee[]`)

Improved secretary code

```
// A class to represent secretaries.  
public class Secretary extends Employee {  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Now we only have to write the portions that are unique to each type.
 - Secretary **inherits** `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` **methods from** `Employee`.
 - Secretary **adds the** `takeDictation` **method**.

Implementing Lawyer

- Let's implement a `Lawyer` class.
- Consider the following employee regulations:
 - Lawyers who get an extra week of paid vacation (a total of 3).
 - Lawyers use a pink form when applying for vacation leave.
 - Lawyers have some unique behavior: they know how to sue.
- The problem: We want lawyers to inherit *most* of the behavior of the general employee, but we want to replace certain parts with new behavior.

Overriding methods

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
 - There is no special syntax for overriding.
To override a superclass method, just write a new version of it in the subclass. This will replace the inherited version.

- Example:

```
public class Lawyer extends Employee {  
    // overrides getVacationForm method in Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
  
    ...  
}
```

- Exercise: Complete the `Lawyer` class.

Complete Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;    // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Now complete the `Marketer` class. Marketers make \$10,000 extra (\$50,000 total) and know how to advertise.

Complete Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return 50000.0;    // $50,000.00 / year
    }
}
```

Levels of inheritance

- Deep hierarchies can be created by multiple levels of subclassing.
 - Example: The legal secretary is the same as a regular secretary except for making more money (\$45,000) and being able to file legal briefs.

```
public class LegalSecretary extends Secretary {  
    ...  
}
```

- Exercise: Complete the `LegalSecretary` class.

Complete LegalSecretary class

```
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double getSalary() {
        return 45000.0;           // $45,000.00 / year
    }
}
```

Interacting with the superclass:
the `super` keyword

Changes to common behavior

- Imagine that a company-wide change occurs that affects all employees.

Example: Because of inflation, everyone is given a \$10,000 raise.

- The base employee salary is now \$50,000.
 - Legal secretaries now make \$55,000.
 - Marketers now make \$60,000.
- We must modify our code to reflect this policy change.

Modifying the superclass

- This modified `Employee` class handles the new raise:

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 50000.0;       // $50,000.00 / year
    }

    ...
}
```

- What problem now exists in the code?
- The `Employee` subclasses are now incorrect.
 - They have overridden the `getSalary` method to return other values such as 45,000 and 50,000 that need to be changed.

An unsatisfactory solution

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        return 55000.0;  
    }  
    ...  
}
```

```
public class Marketer extends Employee {  
    public double getSalary() {  
        return 60000.0;  
    }  
    ...  
}
```

- The employee subtypes' salaries are tied to the overall base employee salary, but the subclasses' `getSalary` code does not reflect this relationship.

Calling overridden methods

- A subclass can call an overridden method with the `super` keyword.
- Calling an overridden method, syntax:

`super . <method name> (<parameter(s)>)`

- Example:

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        double baseSalary = super.getSalary() ;  
        return baseSalary + 5000.0;  
    }  
    ...  
}
```

- Exercise: Modify the `Lawyer` and `Marketer` classes to also use the `super` keyword.

Improved subclasses

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company.
 - For each year worked, we'll award 2 additional vacation days.
- When an Employee object is constructed, we'll pass in the number of years the person has been with the company.
- This will require us to modify our `Employee` class and add some new state and behavior.
- Exercise: Make the necessary modifications to the `Employee` class.

Modified Employee class

```
public class Employee {  
    private int years;  
  
    public Employee(int years) {  
        this.years = years;  
    }  
  
    public int getHours() {  
        return 40;  
    }  
  
    public double getSalary() {  
        return 50000.0;  
    }  
  
    public int getVacationDays() {  
        return 10 + 2 * years;  
    }  
  
    public String getVacationForm() {  
        return "yellow";  
    }  
}
```

Problem with constructors

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol   : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
      ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
- The long explanation: (next slide)

The detailed explanation

- Constructors aren't inherited.
 - The `Employee` subclasses don't inherit the `public Employee(int years) constructor`.
 - Since our subclasses don't have constructors, they receive a default parameterless constructor that contains the following:

```
public Lawyer() {  
    super();           // calls public Employee() constructor  
}
```

- But our `public Employee(int years)` replaces the default `Employee` constructor.
 - Therefore all the subclasses' default constructors are now trying to call a non-existent default superclass constructor.

Calling superclass constructor

- Syntax for calling superclass's constructor:

```
super ( <parameter(s)> ) ;
```

- Example:

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years);    // call Employee constructor  
    }  
    ...  
}
```

- The call to the superclass constructor must be the first statement in the subclass constructor.
- Exercise: Make a similar modification to the `Marketer` class.

Modified Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }

    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

- **Exercise: Modify the `Secretary` subclass to make it compile:**
 - Secretaries' years of employment are not tracked and they do not earn extra vacation for them.
 - `Secretary` objects are also constructed without a `years` parameter.

Modified Secretary class

```
// A class to represent secretaries.  
public class Secretary extends Employee {  
    public Secretary() {  
        super(0) ;  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Note that since the `Secretary` doesn't require any parameters to its constructor, the `LegalSecretary` now compiles without a constructor (its default constructor calls the parameterless `Secretary` constructor).
- This isn't the best solution; it isn't that Secretaries work for 0 years, it's that they don't receive a bonus. How can we fix it?

Inheritance and fields

- Suppose that we want to give lawyers a \$5000 raise for each year they've been with the company.
- The following modification doesn't work:

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years);  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 5000 * years;  
    }  
    ...  
}
```

- The error is the following:

```
Lawyer.java:7: years has private access in Employee  
    return super.getSalary() + 5000 * years;  
                                   ^
```

Private access limitations

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years);  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 5000 * years;  
    }  
    ...  
}
```

- The error is the following:

```
Lawyer.java:7: years has private access in Employee  
    return super.getSalary() + 5000 * years;  
                                   ^
```

- Private fields cannot be directly accessed from other classes, not even subclasses.
 - One reason for this is to prevent malicious programmers from using subclassing to circumvent encapsulation.
 - How can we get around this limitation?

Improved Employee code

Add an accessor for any field needed by the superclass.

```
public class Employee {
    private int years;

    public Employee(int years) {
        this.years = years;
    }

    public int getYears() {
        return years;
    }
    ...
}

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * getYears();
    }
    ...
}
```

Revisiting Secretary

- The `Secretary` class currently has a poor solution.
 - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
 - If we call `getYears` on a `Secretary` object, we'll always get 0.
 - This isn't a good solution; what if we wanted to give some other reward to *all* employees based on years of service?
- Let's redesign our `Employee` class a bit to allow for a better solution.

Improved Employee code

Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```
public class Employee {
    private int years;

    public Employee(int years) {
        this.years = years;
    }

    public int getVacationDays() {
        return 10 + getSeniorityBonus();
    }

    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

- How does this help us improve the Secretary?

Improved Secretary code

The `Secretary` can selectively override the `getSeniorityBonus` method, so that when it runs its `getVacationDays` method, it will use this new version as part of the computation.

- Choosing a method at runtime like this is called *dynamic binding*.

```
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Polymorphism

ПОЛИМОРФИЗЪМ

- **ПОЛИМОРФИЗЪМ:** Способността един и същи код да се използва с различни типове обекти и да има различно поведение в зависимост от типа на обекта.
- Референтна променлива от тип T, може легално да се използва от всички обекти подкласове на T.

```
Employee person = new Lawyer(3);  
System.out.println(person.getSalary());           // 65000.0  
System.out.println(person.getVacationForm());     // pink
```

- Може да се вика всеки метод от `Employee` за различни хора, не и методи специфични за `Lawyer` (като например `sue`).
- **Щом като метода е извикан от обект, той придобива поведението на обекта (както `Lawyer`, а не нормално `Employee`).**

Полиморфизъм + параметри

- Може да декларирате методи, с които да достъпвате типове от superclass като параметри, тогава ще подавате параметри на подтиповете.

```
public class EmployeeMain {  
    public static void main(String[] args) {  
        Lawyer lisa = new Lawyer(3);  
        Secretary steve = new Secretary(2);  
        printInfo(lisa);  
        printInfo(steve);  
    }  
  
    public static void printInfo(Employee empl) {  
        System.out.println("salary = " + empl.getSalary());  
        System.out.println("days = " + empl.getVacationDays());  
        System.out.println("form = " + empl.getVacationForm());  
        System.out.println();  
    }  
}
```

- **OUTPUT:**

```
salary = 65000.0  
vacation days = 21  
vacation form = pink  
  
salary = 50000.0  
vacation days = 10  
vacation form = yellow
```

Полиморфизъм + масиви

- Може да декларирате тип от superclass, и да запишете обекти на всякакъв подтип като елементи от масива.

```
public class EmployeeMain2 {  
    public static void main(String[] args) {  
        Employee[] employees = {new Lawyer(3), new Secretary(2),  
                                new Marketer(4), new LegalSecretary(1)};  
        for (int i = 0; i < employees.length; i++) {  
            System.out.println("salary = " +  
                               employees[i].getSalary();)  
            System.out.println("vacation days = " +  
                               employees[i].getVacationDays();)  
            System.out.println();  
        }  
    }  
}
```

- OUTPUT:
salary = 65000.0
vacation days = 21

salary = 50000.0
vacation days = 10

salary = 60000.0
vacation days = 18

salary = 55000.0
vacation days = 10

полиморфизъм - задача

- Приемете, че следващите четири класове са декларирани:

```
public class Foo {  
    public void method1() {  
        System.out.println("foo 1");  
    }  
  
    public void method2() {  
        System.out.println("foo 2");  
    }  
  
    public String toString() {  
        return "foo";  
    }  
}  
  
public class Bar extends Foo {  
    public void method2() {  
        System.out.println("bar 2");  
    }  
}
```

(продължава на следващия слайт)

Полиморфизъм – задача

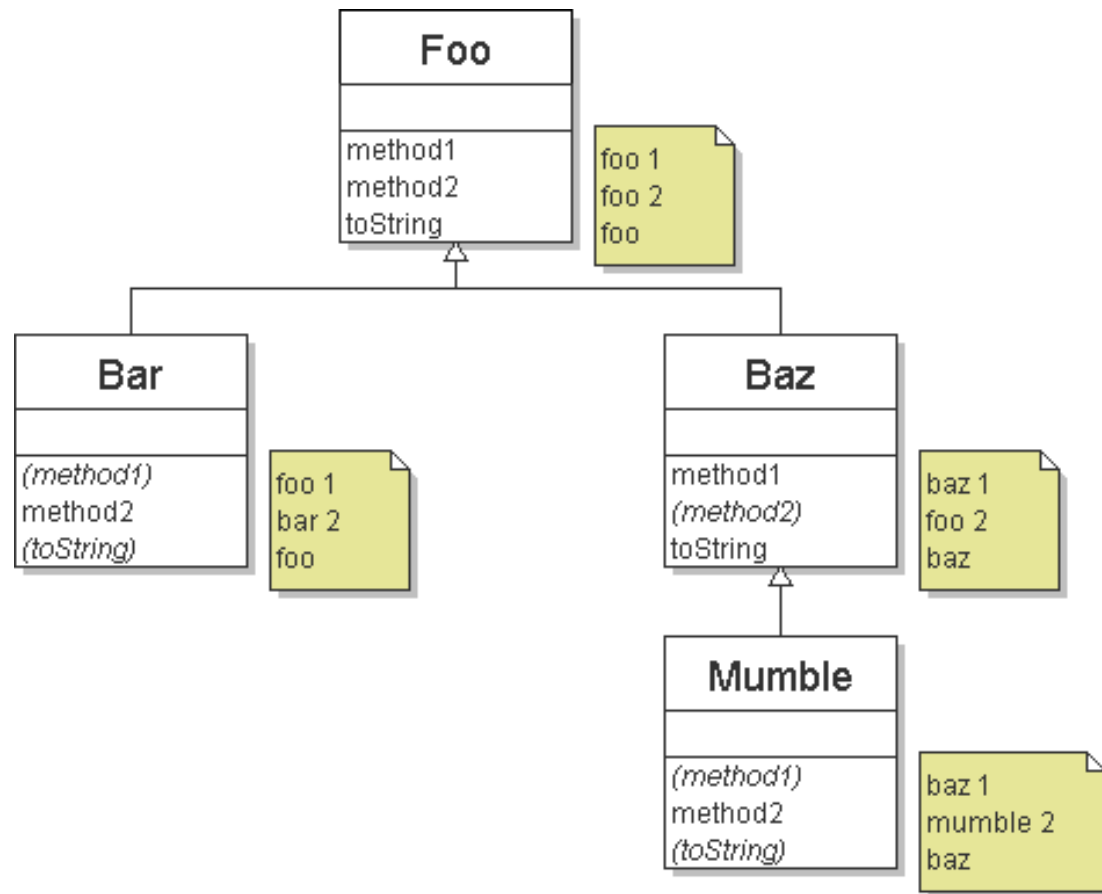
```
public class Baz extends Foo {  
    public void method1() {  
        System.out.println("baz 1");  
    }  
  
    public String toString() {  
        return "baz";  
    }  
}  
  
public class Mumble extends Baz {  
    public void method2() {  
        System.out.println("mumble 2");  
    }  
}
```

- Какъв ще бъде резултата ?

- ```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
 System.out.println(pity[i]);
 pity[i].method1();
 pity[i].method2();
 System.out.println();
}
```

# Използвайте диаграми за решаването на задачата:

- Първият начин да определите изхода е да създадете диаграма за всеки клас и неговите методи, включително и резултата:
  - Добавяйте класовете отгоре(superclass) надолу (subclass).
  - Включете методите за наследяване и техните изходи.



# Използвайте таблици за изхода:

- Друга техника за откриването на изхода е да създадем таблици за класовете и методите и в тях да въвеждаме резултатите.

| <b>method</b> | <b>Foo</b> | <b>Bar</b>   | <b>Baz</b>   | <b>Mumble</b> |
|---------------|------------|--------------|--------------|---------------|
| method1       | foo 1      | <i>foo 1</i> | baz 1        | <i>baz 1</i>  |
| method2       | foo 2      | bar 2        | <i>foo 2</i> | mumble 2      |
| toString      | foo        | <i>foo</i>   | baz          | <i>baz</i>    |

# Полиморфизъм като отговор на проблема!

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
 System.out.println(pity[i]);
 pity[i].method1();
 pity[i].method2();
 System.out.println();
}
```

- **Резултатът от кода:**

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```



# Друг проблем:

- Нека да приемем, че следващите класове са декларирани:  
Последователността в класовете е променена, както и в клиентската част.

```
public class Lamb extends Ham {
 public void b() {
 System.out.println("Lamb b");
 }
}

public class Ham {
 public void a() {
 System.out.println("Ham a");
 }

 public void b() {
 System.out.println("Ham b");
 }

 public String toString() {
 return "Ham";
 }
}
...
```

## Друг проблем - 2

```
public class Spam extends Yam {
 public void a() {
 System.out.println("Spam a");
 }
}

public class Yam extends Lamb {
 public void a() {
 System.out.println("Yam a");
 }

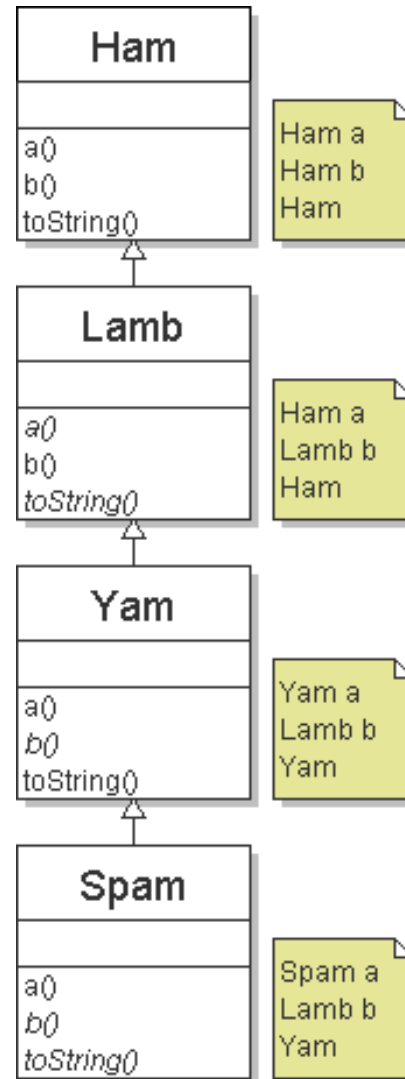
 public String toString() {
 return "Yam";
 }
}
```

- **Какъв ще е резултатът от изпълнението на следния код?**

```
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
 System.out.println(food[i]);
 food[i].a();
 food[i].b();
 System.out.println();
}
```

# UML class diagram

- Диаграмата описва поведението на класа:



# The table

- В таблицата е отбелязано поведението на класа:

| <b>method</b> | <b>Ham</b> | <b>Lamb</b>  | <b>Yam</b>    | <b>Spam</b>   |
|---------------|------------|--------------|---------------|---------------|
| a             | Ham a      | <i>Ham a</i> | Yam a         | Spam a        |
| b             | Ham b      | Lamb b       | <i>Lamb b</i> | <i>Lamb b</i> |
| toString      | Ham        | <i>Ham</i>   | Yam           | <i>Yam</i>    |

# Отговор на задачата

```
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
 System.out.println(food[i]);
 food[i].a();
 food[i].b();
 System.out.println();
}
```

- **Резултат:**

Yam  
Spam a  
Lamb b

Yam  
Yam a  
Lamb b

Ham  
Ham a  
Ham b

Ham  
Ham a  
Lamb b

# Интерфейси

# Взаимовръзки на типовете

- Създайте класовете за представяне на 2D форми като `Circle`, `Rectangle`, и `Triangle`.
- Има атрибути или операции, които са еднакви за всички форми.
  - perimeter** - обиколката на формата
  - area** - лицето на формата
- Всяка форма има три атрибути, но всеки изчислява различно.

# Shape area, perimeter

- Rectangle (as defined by width  $w$  and height  $h$ ):

$$\text{area} = w h$$

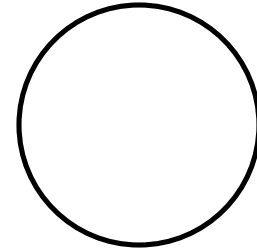
$$\text{perimeter} = 2w + 2h$$



- Circle (as defined by radius  $r$ ):

$$\text{area} = \pi r^2$$

$$\text{perimeter} = 2 \pi r$$

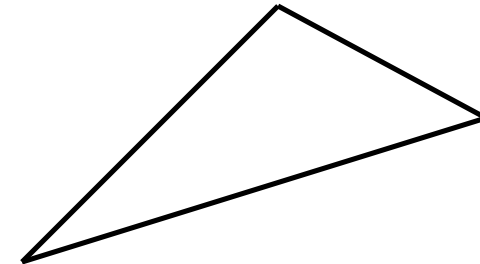


- Triangle (as defined by side lengths  $a$ ,  $b$ , and  $c$ )

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

$$\text{where } s = \frac{1}{2}(a+b+c)$$

$$\text{perimeter} = a + b + c$$





# Сходство в поведението на различните класове

- Нека да създадем класове с методи `perimeter` и `area`.
- Трябва да се напише и клинетска програма, която да създава обекти с различни форми, но имащи общо поведение, като напр.:
  - Създайте метод, който да разпечатва на различните форми лицето и периметъра;
  - Създайте масив от форми, който да съдържа различни обекти(от различни типове форми).
  - Напишете метод, който да връща правоъгълник, окръжност, триъгълник или някоя друга форма.
  - Направете метода `DrawingPanel` да показва различните форми на екрана.

# Интерфейси

- **interface:** Списък от методи, който могат да бъдат реализирани в класове
- Наследяването им изгражда връзка и дава възможност да се сподели кода.
  - Обект от класа Lawyer може да бъде третиран като обект от класа Employee, и Lawyer наследява Employee's кода.
- Интерфейсите ви дават връзка и без да се споделя код.
  - Обект от клас Rectangle може да бъде третиран като обект от клас Shape.
- Аналогично на идеята на правилата и сертификатите
  - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits, and do management consulting."
  - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."

# Interface синтаксис

- Interface declaration, general syntax:

```
public interface <name> {
 public <type> <name>(<type> <name>, ..., <type> <name>);
 public <type> <name>(<type> <name>, ..., <type> <name>);
 ...
 public <type> <name>(<type> <name>, ..., <type> <name>);
}
```

Примери:

```
public interface Vehicle {
 public double getSpeed();
 public void setDirection(int direction);
}
```

- За да се приложи интерфейса в производен клас се използва декларацията

```
[public] class fille implements service {
 ...
}
```

- **abstract method:** метод без реализация.
  - Не се описва тялото на метода, защото ние искаме да позволим на всеки клас да реализира свое поведение.
  - ЗАДАЧА: Напишете интерфейс за shapes.

# АБСТРАКТНИ КЛАСОВЕ И МЕТОДИ

- съществуват и т.нар *абстрактни методи*. Това са методи, които съдържат само декларация, но нямат имплементация. Декларират се с ключовата дума **abstract**. Ако декларираме поне един такъв метод в рамките на даден клас, то този клас задължително трябва да съдържа в декларацията си думата **abstract**, т.е. трябва да бъде абстрактен.

**Определение:** Абстрактен клас е такъв клас, който е деклариран с ключовата дума **abstract** и съдържащ или не абстрактни методи.

- В един абстрактен клас може да напишем както абстрактни методи, така и такива които не са абстрактни— ще ги наричаме конкретни методи.

# ПРИМЕР:

```
public abstract class Figure {
 public abstract double area(float a, float b);
}

class Triangle extends Figure{
 @Override
 public double area(float a, float b) { return a*b/2; }
}

class Rectangle extends Figure{
 @Override
 public double area(float a, float b) { return a*b; }
}
```

# ЗАЩО СА НЕОБХОДИМИ АБСТРАКТНИТЕ КЛАСОВЕ?

- Абстрактните класове, както подсказва името им носят известно ниво абстракция. В йерархията на наследяването те могат да обобщят определена функционалност или белези, характерни за определено ниво е нея.

## Rules for Java Abstract class



1

An abstract class must be declared with an abstract keyword.

2

It can have abstract and non-abstract methods.

3

It cannot be instantiated.

4

It can have final methods

5

It can have constructors and static methods also.

# ЗАЩО СА НЕОБХОДИМИ АБСТРАКТНИТЕ КЛАСОВЕ?

- Нека имаме клас Животно и няколко други класове – Котка, Куче, Кон. Последните три класа наследяват класа Животно, така че те имат няколко общи навици(методи), които имат всички животни, затова ги наследяват от Животно. Ако осмислите примера, ще разберете, че няма смисъл да създаваме инстанция на клас Животно, защото няма как да знаем какво е това животно. Затова спазваме следната философия в ООП: ако имаме клас, който да обобщи поведение и неспецифични, а общи белези на някои други класове, то го декларираме като абстрактен клас, а наследниците му – като конкретни класове, които задължително да имплементират поведението, но всеки по свой начин. Ако за Животно имаме метод(поведение) „издаване на звук“, то това поведение би било абстрактно, защото различните животни издават различни звуци. Например клас Куче ще имплементира функционалността „издаване на звук“ като „Бау-бау“, Котка – като „Мяу – мяу“ и тн. Ако създадем обект от тип Куче и извикаме метода му за издаване на звук, то ще лае, но ако създадем обект (хипотетично) от клас Животно и извикаме метода му за издаване на звук, то няма да знае какъв звук да издаде.



## Интерфейси:

Ако имаме абстрактен клас, който съдържа само абстрактни методи, то спокойно можем да го направим на интерфейс.

Ето и определение за интерфейс:

Интерфейсът е подобен на абстрактния клас, но с тази разлика, че може да съдържа абстрактни методи и член променливи. С навлизането на Java 8, в интерфейсите вече може да има както абстрактни, така и конкретни – имплементирани методи. Последните се наричат `default methods` – нововъведение, позволяващо да се задава функционалност по подразбиране. Променливите, декларирани в интерфейса са само публични, статични и константни – `public final static`.

| Основа за сравнение    | клас                                                                                    | интерфейс                                                                                                                 |
|------------------------|-----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Основен                | Клас се създава за създаване на обекти.                                                 | Интерфейсът никога не може да бъде инстанциран, тъй като методите не могат да изпълняват никакви действия по извикването. |
| Ключова дума           | клас                                                                                    | интерфейс                                                                                                                 |
| Спецификатор за достъп | Членовете на даден клас могат да бъдат частни, публични или защитени.                   | Членовете на интерфейса са винаги публични.                                                                               |
| методи                 | Методите на даден клас са дефинирани за извършване на конкретно действие.               | Методите в интерфейса са чисто абстрактни.                                                                                |
| Прилагане / Удължи     | Един клас може да реализира произволен брой интерфейс и може да разшири само един клас. | Един интерфейс може да разшири няколко интерфейса, но не може да реализира всеки интерфейс.                               |
| конструктор            | Един клас може да има конструктори за инициализиране на променливите.                   | Един интерфейс никога не може да има конструктор, тъй като едва ли има някаква променлива, която да се инициализира.      |

## Ключови разлики между клас и интерфейс в Java

1. Класът може да бъде инстанциран чрез създаване на неговите обекти. Интерфейсът никога не е инстанциран, тъй като методите, декларирани в интерфейс, са абстрактни и не изпълняват никакви действия, така че няма смисъл да се създава интерфейс.
2. Клас се декларира с ключовата дума `class`. По същия начин се създава интерфейс с помощта на интерфейсна дума `interface`.
3. Членовете на даден клас могат да имат спецификатор за достъп като публичен, частен, защитен. Но членовете на интерфейса са винаги публични, тъй като трябва да бъдат достъпни от класовете, които ги прилагат.
4. Методите вътре в класа се дефинират, за да се извърши действие върху полетата, декларирани в класа. Тъй като интерфейсът липсва в декларирането на полета, методите вътре в интерфейса са чисто абстрактни.
5. Един клас може да реализира произволен брой интерфейси, но може да разшири само един супер клас. Един интерфейс може да разшири произволен брой интерфейси, но не може да реализира всеки интерфейс.
6. Един клас има конструктори, определени вътре в него, за да се инициализира променливата. Но интерфейсът няма никакви конструктори, тъй като няма полета, които да се инициализират. Полетата на интерфейса се инициализират само по време на декларацията.

# Интерфейс Shape

- Какво съдържа този интерфейс:

```
public interface Shape {
 public double area();
 public double perimeter();
}
```

- Трябва да описва общите характеристики за всички форми. (всяка форма има лице и периметър.)

# Реализация на интерфейси

- Класът трябва да декларира, че реализира интерфейс.
  - Това означава, че класът съдържа реализацията на всеки абстрактен метод от интерфейса.  
(Иначе, класът няма да се компилира.)

- Implementing an interface, general syntax:

```
public class <name> implements <interface name> {
 ...
}
```

- Example:

```
public class Bicycle implements Vehicle {
 ...
}
```

(What must be true about the `Bicycle` class for it to compile?)

# Изисквания към интерфейсите

- При създаването на клас, ние задаваме че наследява Shape но не реализираме методите `area` и `perimeter`, тогава няма да се компилира.

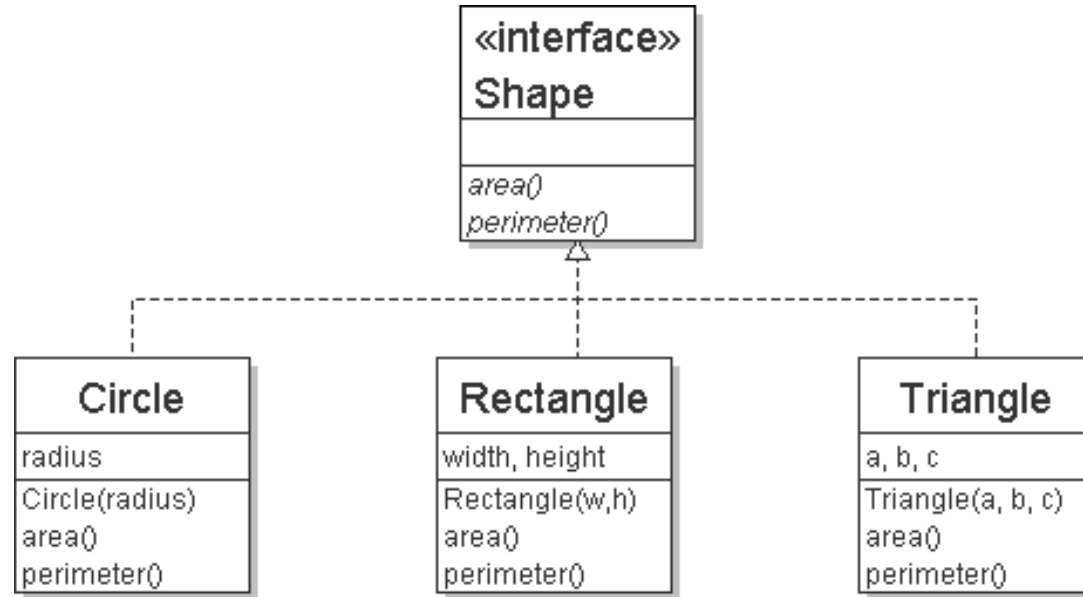
- Example:

```
public class Banana implements Shape {
 ...
}
```

- The compiler error message:

```
Banana.java:1: Banana is not abstract and does not
override abstract method area() in Shape
public class Banana implements Shape {
 ^
```

# UML class diagram - interfaces



- We draw arrows upward from the classes to the interface(s) they implement.
  - There is a supertype-subtype relationship here; e.g., all Circles are Shapes, but not all Shapes are Circles.
  - This kind of picture is also called a *UML class diagram*.
- Exercise: Implement the `Circle`, `Rectangle`, and `Triangle` classes.

# Circle class

```
// Represents circles.
public class Circle implements Shape {
 private double radius;

 // Constructs a new circle with the given radius.
 public Circle(double radius) {
 this.radius = radius;
 }

 // Returns the area of this circle.
 public double area() {
 return Math.PI * radius * radius;
 }

 // Returns the perimeter of this circle.
 public double perimeter() {
 return 2.0 * Math.PI * radius;
 }
}
```



# Rectangle class

```
// Represents rectangles.
public class Rectangle implements Shape {
 private double width;
 private double height;

 // Constructs a new rectangle with the given dimensions.
 public Rectangle(double width, double height) {
 this.width = width;
 this.height = height;
 }

 // Returns the area of this rectangle.
 public double area() {
 return width * height;
 }

 // Returns the perimeter of this rectangle.
 public double perimeter() {
 return 2.0 * (width + height);
 }
}
```

# Triangle class

```
// Represents triangles.
public class Triangle implements Shape {
 private double a;
 private double b;
 private double c;

 // Constructs a new Triangle given side lengths.
 public Triangle(double a, double b, double c) {
 this.a = a;
 this.b = b;
 this.c = c;
 }

 // Returns this triangle's area using Heron's formula.
 public double area() {
 double s = (a + b + c) / 2.0;
 return Math.sqrt(s * (s - a) * (s - b) * (s - c));
 }

 // Returns the perimeter of this triangle.
 public double perimeter() {
 return a + b + c;
 }
}
```

# Интерфейси и полморфизъм

- Използвайки интерфейсите не носи облекчение за проектирането на класовете, предимството е за клиентския код.

- Има връзка м/у интерфейса като поведение и клиентския код, който взима предимството на полиморфизма.

- Наприме:

```
public static void printInfo(Shape s) {
 System.out.println("The shape: " + s);
 System.out.println("area : " + s.area());
 System.out.println("perim: " + s.perimeter());
 System.out.println();
}
```

- Всеки обект, който реализира интерфейса може да бъде предаван като параметър от по-горен метод.

```
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

# Масиви от интерфейсен тип

- Може да се създаде масив, чийто елементи да са от интерфейсен тип или да записват обекти, които се реализират от интерфейса като елементи.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);

Shape[] shapes = {circ, tri, rect};
for (int i = 0; i < shapes.length; i++) {
 printInfo(shapes[i]);
}
```

- Всеки елемент от масива има сходно поведение на своя обект когато се извиква метода `printInfo`, или когато `area` или `perimeter` са извикани.