Java Bootcamp 2004

Expectations:

We assume that you have studied C, C++, or Java and know about the following:

- Variables and variable declarations
- Expressions (integer, boolean)
- Assignment statement
- If-statement and if-else statement
- While-loop and for-loop

We concentrate on the following Java constructs:

- Primitive types, variables, expressions
- Casting between types
- The class as a definition of the format of an object (instance, manilla folder)
- The new-expression
- Referencing instance variables and methods
- Methods (procedures, functions, constructors)
- Subclasses, inheritance, and overriding

DrJava

We use the IDE (Interactive Development Environment) DrJava in this course. We use it to demo during lecture. Its **Interactions pane** allows us to evaluate expressions and execute statements (including method calls) without having to have a complete Java application.

If you have your own computer, please get on the course website, download DrJava, and practice using it. Use it to learn about Java.

We'll use it in this bootcamp.

Resources for learning Java

See website for reading material

- **ProgramLive**, by Gries & Gries. Has a CD, which has 250 2-4-minute lectures with synched animation. Used in CS100J this year. The glossary of the CD is a good source of information.
- **Course textbook**.
- **Java Precisely**.
- **Java in a Nutshell**.
- **Java tutorial**: http://java.sun.com/docs/books/tutorial/

Primitive types

type	range of values	space used	
byte	–128..127	1 byte	
short	–32768..32767	2 bytes	
int	$-2^{31}..2^{31}$	4 bytes	***
long	$-2^{63}..2^{63}$	8 bytes	
float	6 significant digits, $10^{-46}..10^{38}$	4 bytes	
double	15 sig. digits, $10^{-324}..10^{308}$	8 bytes	***
char	Unicode character	2 bytes	***
boolean	{**false**, **true**}	1 bit or 1 byte	***

Use mainly these types

type	range of values	space used
int	$-2^{31}..2^{31}$	4 bytes
double	15 sig. digits, $10^{-324}..10^{308}$	8 bytes
char	Unicode character	2 bytes
boolean	{**false**, **true**}	1 bit or 1 byte

Operations on type int

– h

h + k   h – k

h * k   h / k   h % k

h / k yields an **int**: 7 / 2 is 3!!!!

h % k is the remainder when h is divided by k.

min, max values for primitive types

Short.MAX_VALUE	smallest **short** value
Short.MIN_VALUE	largest **short** value
Integer.MAX_VALUE	smallest **int** value
Integer.MIN_VALUE	largest **int** value
Double.MAX_VALUE	smallest POSITIVE **double**
Double.MIN_VALUE	largest POSITIVE **double**

etc.

Type boolean

Values: **true** and **false**

Complement:	! b
And (conjunction):	b && c

value: if b is **false**, then **false**; otherwise, whatever c is.

Or (disjunction):	b || c

value: if b is **true**, then **true**; otherwise, whatever c is

SHORT-CIRCUIT EVALUATION

x = 0 || 5 / x = 1 is **true** x $\boxed{0}$

5 / x = 1 || x = 0 GIVES AN EXCEPTION

C, C++ use 1 and 0 for **true** and **false**. Java has a special type, with two values: **true** and **false**.

Type boolean

Don't write	**Write**
if (b == **true**)	**if** (b)
if (b == **false**)	**if** (!b)

if (x == 0)
y= **true**;
else y= **false**
	y= x == 0;

Basic variable declarations

Variable: a name with associated integer; a named box, with a value in the box.

Basic declaration: <type> <variable> ;

int x;	x $\boxed{5}$
boolean b;	b $\boxed{\textbf{true}}$

Whether a variable has a specific default value when declared depends on where it is declared. Discuss later.

Assignment statement

syntax: <variable> = <expression> ;

semantics: Evaluate the <expression> and store its value in the <variable>.

read as: <variable> *becomes* <expression>

$\boxed{\textbf{true}}$

```
x= x + 1;      // Add 1 to x
               // or add 1 to the value of x
x= y;          // The value of the expression is the value in
               // y. The value is stored in x.
```

Basic initializing declarations

Basic initializing declaration:

<type> <variable> = <expression> ;

Equivalent to:

<type> <variable>;
<variable>= <expression>;

int x= 5 * 3;	x $\boxed{15}$
boolean b= **true**;	b $\boxed{\textbf{true}}$

Casting

narrowest type widest type

byte -> **short** -> **int** -> **long** -> **float** -> **double**

There are no operations in types **byte** and **short**. If they appear as operand of an operation, they are promoted (automatically cast) to **int** or **long** and the operation is performed in **int** or **long**.

If one operand of x + y is **long**, the other is cast to **long** and a **long** addition is done. Otherwise, the operation is an **int** addition.

Casting

narrowest type widest type

byte -> **short** -> **int** -> **long** -> **float** -> **double**

byte b= 5;
b= b + 5; // illegal, because exp 5 + 5 has type **int**.
b= (**byte**) (b + 5);

(**byte**) is a prefix operator, called a **caste**.

It casts its operand to type byte.

Use any type as a caste, e.g. **(int**)

Widening casts performed automatically.

Narrowing casts have to be explicit.

Type char

narrowest type widest type

char -> **int** -> **long** -> **float** -> **double**

Values of type **char**: the characters,

‘b’ ‘5’ ‘&’

‘\n’ new-line character

‘\\’ backslash char

(**int**) ‘A’ is the integer that represents char ‘A’: 65

(**char**) 65 is the character that is represented by 65: ‘A’

You don’t need to remember much about type **char**. Just that it exists, and you can look it up whenever you want. Best reference: ProgramLive.

Class String

Mentioned now because you may hear about it from time to time. An object of class String is a sequence of **char**s:

“abcd123#\n” **Note:** double quotes for String, single quotes for **char**

1 + “abc” + 2.1 is “1abc2.1”

1 + 2 + “abc” is 3 + “abc” is “3abc”

If at least one operand of + is a String, then + denotes “catenation”. The other operand is converted to a String, if necessary.

s.length()	number of characters in String s
s.charAt(i)	character at position i of s (0 is first)
s.substring(h..k)	substring s[h..k-1]

The Class

- All variables, methods (procs, funcs) are declared in a **class**.
- **Class def.** defines format of objects (instances) of the class.

public class C {

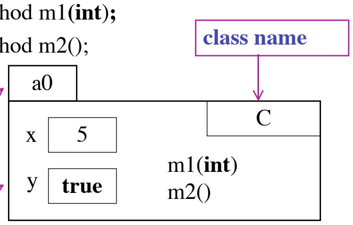
Declaration of instance variable (field) x;

Declaration of instance variable (field) y;

Declaration of instance method m1**(int);**

Declaration of instance method m2();

}



class name

Tab contains name of object (address in memory)

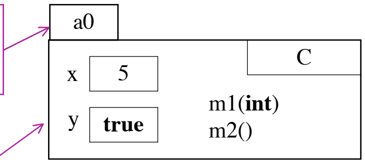
Object drawn like a manilla folder

The Class

```
public class C {
   private int x;
   private double y;
   public void m1(int) { …}
   public int m2() { … }
```

public components can be referenced anywhere; **private** ones only in the class itself.

Generally, but not always, fields are private and methods are public.



Tab contains name of object (address in memory)

Object drawn like a manilla folder

Three kinds of method

```
public void proc(par. decs) {
   body
}
```

proc is a *procedure*. It does not return a value. A call of it is a statement.

```
public int func(par. decs) {
   body
}
```

func is a *function*. It returns a value of type int. A call of it is an expression.

```
public C(par. decs) {
  body
}
```

C is a *constructor*. Can be defined only in class C. Can be called only in restricted places. Purpose: initialize (some) fields of a new object of class C.

body: sequence of statements and declarations.

The new-expression

new C1(5)

```
public class C1 {
   private int x= 4;
   private double y= 2.0;
   public void m1(int) { …}
   public C1(int p) {
       x= p;
   }
}
```

The new-expression is evaluated in 3 steps:

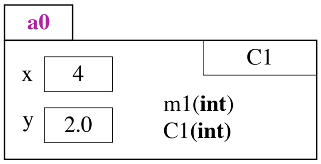
1. Create an object of class C1 (that's what "**new** C1" says), initializing fields acc. to their declarations.
2. Execute constructor call C1(5).
3. Yield as the value of the new-expression the name of the object.

The new-expression

new C1(5)

```
public class C1 {
   private int x=  4;
   private double y= 2.0;
   public void m1(int) { …}
   public C1(int p) {
       x= p;
   }
}
```

1. Create an object of class C1 (that's what "**new** C1" says), initializing fields acc. to their declarations

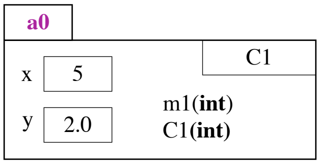

a0
C1
x 4
y 2.0
m1(**int**)
C1(**int**)

The new-expression

new C1(5)

```
public class C1 {
   private int x=  4;
   private double y= 2.0;
  public void m1(int) { …}
   public C1(int p) {
       x= p;
  }
}
```

1. Create an object of class C1 (that's what "**new** C1" says), initializing fields acc. to their declarations
2. Execute constructor call C1(5).

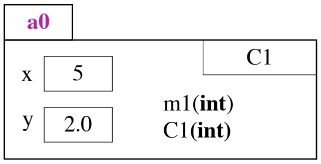

a0
C1
x 5
y 2.0
m1(**int**)
C1(**int**)

The new-expression

new C1(5)
a0

```
public class C1 {
   private int x=  4;
   private double y= 2.0;
  public void m1(int) { …}
  public C1(int p) {
       x= p;
   }
}
```

1. Create an object of class C1 (that's what "**new** C1" says), initializing fields acc. to their declarations
2. Execute constructor call C1(5).
3. Yield as the value of the new-expression the name of the object.

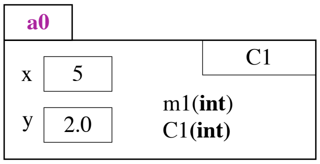

a0
C1
x 5
y 2.0
m1(**int**)
C1(**int**)

The new-expression –used in an assignment

```
public class C1 {
   private int x=  4;
   private double y= 2.0;
   public void m1(int) { …}
   // Constructor: an instance with x field equal to p
   public C1(int p) {
       x= p;
   }
}
```

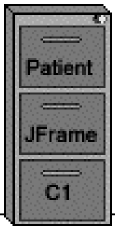
```
C1 c;
…
c=  new C1(5);
```

c a0


a0
C1
x 5
y 2.0
m1(**int**)
C1(**int**)

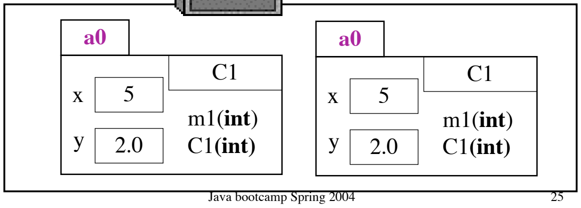
Our analogy for explaining classes and objects

Class is a file drawer: contains the objects (manilla folders) of the class.



```
public class C1 {
    private int x;
    private double y;
    public void m1(int) { …}
     public C1(int p) {  … }
}
```

file drawer C1



Default constructor

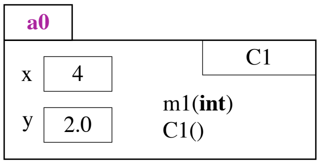
If you don't declare a constructor, then the following one is automatically declared for you:

public C1() {}

```
public class C1 {
    private int x= 4;
    private double y= 2.0;
    public void m1(int) { …}
}
```

c= **new** C1();

c a0



Static components

Static variable or **class variable**.

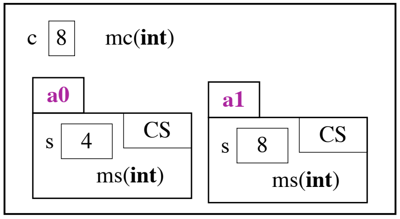
Static method or **class method**.

Static component goes not in object but in file drawer.

Only ONE copy of each static component.

```
public class CS {
    private static int c= 4;
    private  int s= 2.0;
    public void mc(int) { …}
    public static void ms(int) { …
    }
}
```

File drawer for CS



Java class Math contains only static components

```
public class Math {
    public static final double PI; // pi
    public static final double E; // base of natural log
    public static double sin(double angle)
    public static int abs(int x)
    public static double abs(double x)
    public static int ceil(int x)
    public static double ceil(double)
    …
}
```

final: no other assignments to it. Can't be changed

Some method names are **overloaded**. Distinguish between them based on number and types of arguments.

Getter methods

Generally, fields are private, so that they cannot be accessed directly. To allow access, provide a public **getter method**, i.e. a function that returns the value of the field. This technique provides some security and is a good software engineering technique.

The example to the right shows conventions for naming and specifying getter methods.

```
public class C1 {
    private int x= 4;
    private double y= 2.0;
    /** = field x */
    public int getX() {
        return x;
    }
    /** = field y */
    public double getY() {
        return y;
    }
}
```

Setter methods

A setter method is an instance procedure that saves some value in an object.

The example to the right shows conventions for naming and specifying setter methods.

```
public class C1 {
    private int x= 4;
    private double y= 2.0;
    /** = field x */
    public int getX() {
        return x;
    }
    /** = field y */
    public double getY() {
        return y;
    }
    /** Set field x to p */
    public void setX(int p) {
        x= p;
    }
}
```

Method toString

```
public class Point {
   /* The point is (x, y) */
   private int x= 4;
   private int y= 2;

   /** = description  of
          this instance */
   public String toString() {
      return "(" + x + ", " + y + ")";
   }
}
```

Define instance method toString in almost every class. It yields a description of the object in which it occurs, using a format that makes sense for the class.

System.out.println(new Point());

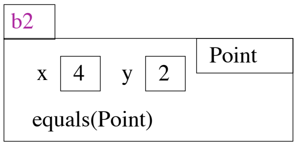
prints

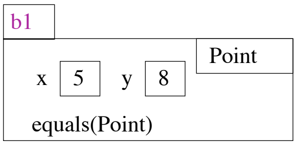
(4, 2)

In some contexts where a String is expected and an object appears, the objects' toString method is called.

Keyword this.When it appears in a method, this refers to the object in which the method occurs.

```
public class Point {
   /* The point is (x, y) */
  private int x= 4;
  private int y= 2;

   /** = point p equals point q */
   public static boolean equals(Point p, Point q) {
       return p.x == q.x   &&  p.y == q.y;
   }

   /** = this point equals point p */
   public boolean equals(Point p) {
       return equals(p, this);
   }
}
```





Summary of classes in Java

- Class defines content of file drawer and format of objects:
- File drawer contains static components and created objects, drawn as manilla folders. The name of an object —its location in memory— is drawn on the tab of the folder.
- new-expression, used to create objects. Know the 3 steps in evaluating it.
- Constructor: called in new-expression to initialize fields.
- Use of **private** and **public.**
- Getter and setter methods.
- static vs. non-static variables (instance variables or fields).
- static vs. non-static methods (instance methods).
- Method toString.
- Two uses of keyword **this**.

Subclasses

The subclass definition has this form:

public class *subclass-name* **extends** *superclass-name* {

declarations of

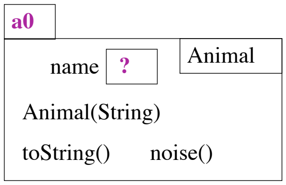
- *instance variables*
- *instance methods*
- *class variables*
- *class methods*

}

only difference between a subclass definition and a superclass definition

Class Animal

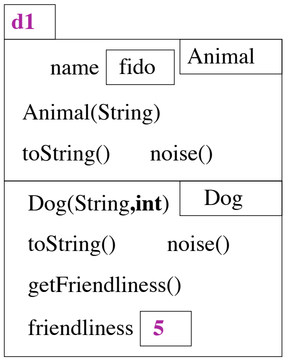
```
public class Animal {
    private String name= "";

    /** Constructor: instance with name n */
    public Animal(String n)
       { name= n; }

    /** = a description of this Animal */
   public String toString()
       { return name; }

  /** = noise this animal makes */
   public String noise()
       { return null; }
 }
```



Subclass Dog

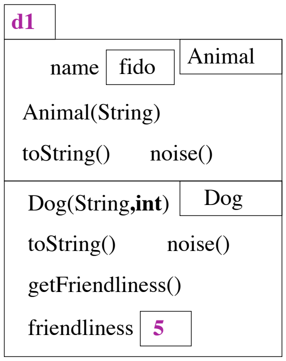
```
public class Dog extends Animal {
  private int friendliness;

  /** Constructor: instance with
       name n, friendly rank r  */
  public Dog(String n, int r) { …}

  /** = desc. of this Dog */
  public String toString() {…}

  /** = noise this animal makes */
  public String noise() {…}

  /** = friendliness of this Dog */
  public int getFriendliness() {…}
}
```

subclass inherits all components of superclass



Subclass Dog: constructor

```
public class Dog extends Animal {
  private int friendliness;

  /** Constructor: instance with
      name n, friendly rank r  */
  public Dog(String n, int r) {
      super(n);
      friendliness= r;
  }
  …
}
```



constructor can't reference field name —it's private.

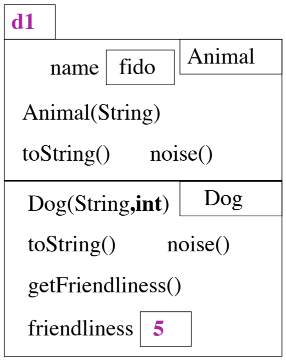
FIRST statement of constructor can be a call on a superclass constructor.

Example shows how to do it.

Subclass Dog: Overriding

```
public class Dog extends Animal {

  public Dog(String n, int r) { … }
  public String toString() { … }
  …
}
```

d= new Dog("fido", 10);
d.toString();

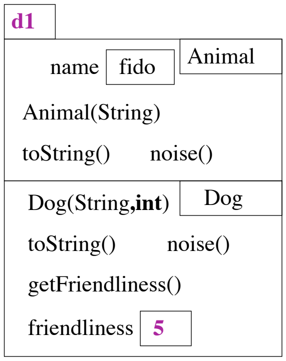


declaration of toString in Dog overrides declaration of toString in Animal.

When determining which method to call, use the bottom-up rule: start at bottom of folder and work upword

Calling the inherited method: another use of super

```
public class Dog extends Animal {

   public String toString() {
      return super.toString() +
            ", friendliness " +
            friendliness;
   }
  …
}
```



Within toString, a call toString() refers to the same method toString! To refer to the toString method of the superclass, prefix the call with "super.".

Casting up and down (narrowing and widening)

Assume Dog and Cat are subclasses of Animal

```
Animal a;
Dog d;
Cat c;

a=  d;

a= (Animal) d;


d= (Dog) a;


d= a;
```

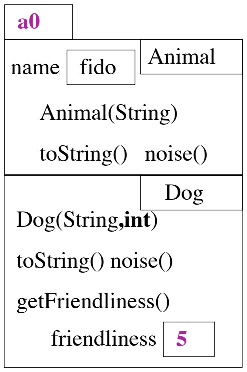
implicit

object d is cast up (widened) to Animal

explicit

object a is cast down (narrowed) to Dog

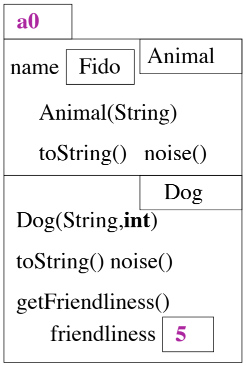
illegal



Casting up and down (narrowing and widening)

Assume Dog and Cat are subclasses of Animal

```
Animal a;
Dog d;
Cat c;

a=  d;
…
if (d instanceof Dog) {
   …
} else { … }
```



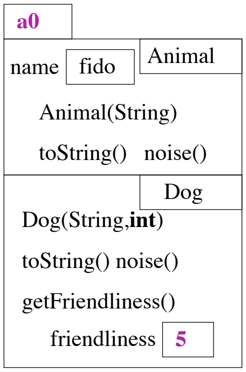
Why cast?

Assume Dog and Cat are subclasses of Animal

Have an array of objects of class Animal, each of which is a Cat or a Dog.

Assignment to x[k] of a Dog or Cat object is an up-cast!!

```
Animal[ ] x= new Animal[100];

x[1]=  new Dog("Fido", 10);
x[2]=  new Dog("Pitty", 0);
x[3]=  new Cat("Tabby", … );
```



Apparent and real class-type of a variable?

Assume Dog and Cat are subclasses of Animal

Animal[ ] x= **new** Animal[100];

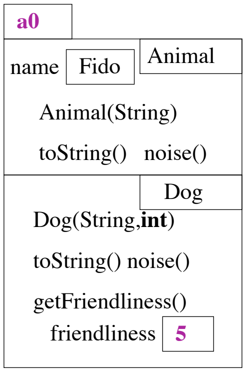
x[1]= **new** Dog(“Fido”, 10);

Apparent type of x[1] is Animal.

Syntactic property. *Apparently*, looking at the declaration of x, x[1] contains an Animal.

real type of x[1] is Dog.

Semantic property. Really, x[1] is a Dog. Real type can change at runtime when x[1]= e; is executed.


a0
name Fido
Animal
Animal(String)
toString() noise()
Dog
Dog(String,**int**)
toString() noise()
getFriendliness()
friendliness 5

Apparent class-type determines what components of object can be referenced

Animal[ ] x= **new** Animal[100];

x[1]= **new** Dog(“Fido”, 10);

Apparent type of x[1] is Animal.

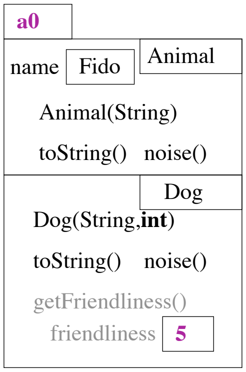
Syntactically legal:

x[1].name

x[1].toString()

x[1].noise()

Other components are there but cannot be referenced.


a0
name Fido
Animal
Animal(String)
toString() noise()
Dog
Dog(String,**int**)
toString() noise()
getFriendliness()
friendliness 5

Apparent class-type determines what components of object can be referenced

Animal[ ] x= **new** Animal[100];

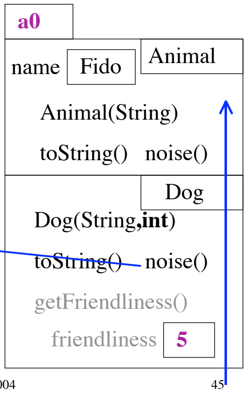
x[1]= **new** Dog(“Fido”, 10);

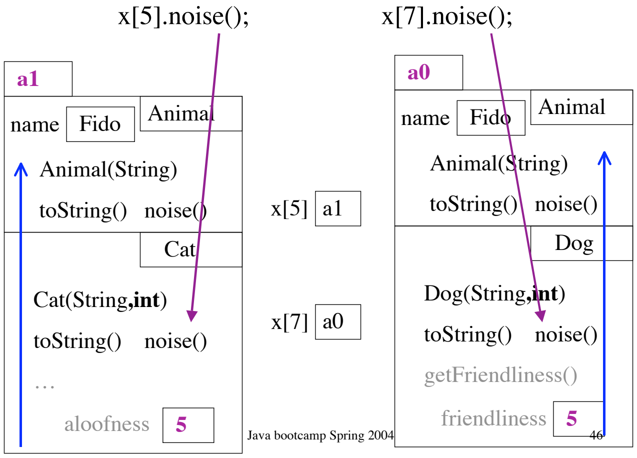
real type of x[1] is Dog.

x[1].noise()

CALLS THIS METHOD

That’s what the bottom-up rule says.


a0
name Fido
Animal
Animal(String)
toString() noise()
Dog
Dog(String,**int**)
toString() noise()
getFriendliness()
friendliness 5


x[5].noise();
x[7].noise();
a1
name Fido
Animal
Animal(String)
toString() noise()
Cat
Cat(String,**int**)
toString() noise()
…
aloofness 5
x[5] a1
x[7] a0
a0
name Fido
Animal
Animal(String)
toString() noise()
Dog
Dog(String,**int**)
toString() noise()
getFriendliness()
friendliness 5

Class Object
the superest class of them all

```
public class Object {
  /** = description of this Object */
  public String toString() { … }

  /** Object ob and this object are the same */
  public boolean equals(Object ob)
    {  return this = ob; }
  …
}
```

Object, in package java.lang, automatically is the superclass of all classes that do not extend another class.

Object has a few methods. For us, the most important ones are toString and equals. They are inherited by all classes. Usually, they are overridden.

Overriding function equals

```
/** An instance is a point in the plane */
public class Object {
  private int x;
  private int y;
   /** = description of this Point */
  public String toString()
    { return “(“ + x + “, “ + y + “)”; }
  /** Object ob and this object describe the same point */
  public boolean equals(Object ob) {
    return  ob != null   &&   ob instanceof  Point   &&
                 x = ((Point) ob).x   &&   y = ((Point)ob).y;
  }
}
```

Object, in package java.lang, is the superclass of all classes that do not extend another class.

Overriding function equals

```
public class Object {
  private int x;        private int y; /** Object ob and this
   object describe the same point */
  public boolean equals(Object ob) {
    return  ob != null   &&   ob instanceof  Point   &&
               x = ((Point) ob).x   &&   y = ((Point)ob).y;
  }
}
```

Java says: equals should be an **equivalence relation**, i.e.

Reflexive: b.equals(b)

Symmetric: b.equals(c) = c.equals(b)

Transitive: if b.equals(c) and c.equals(d), then bequals(d)

Four kinds of variable

```
public class C {
    private int ins;
    public static int cla;

    public void p(int par) }
      if (…) {
         par= par+1;
         int loc;
          …
      }
   }
}
```

ins: instance variable or field. Belongs in each folder (object, instance) of class C. Created when folder is created.

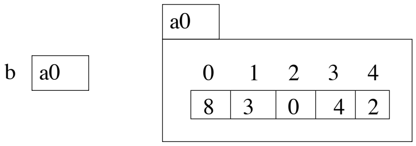
cla: class variable. Belongs in file drawer class C. Created at start of program.

par: parameter. Created when frame for a call on p is created; destroyed when frame is erased. Scope: method body.

loc: local variable. Created when frame for a call on p is created; destroyed when frame is erased. Scope: all the statements after its declaration in the block in which it is declared.

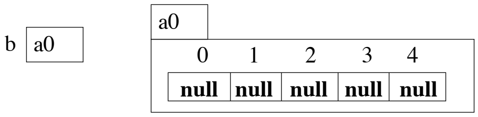
Array: an object that contains a bunch of variables of the same type.

```
int[] b;           // Declaration of variable b of type int[]  (int array).
b= new int[4]; //  Create an assign to b an object that is an array of 4 int vars.
                   //  Vars are named b[0], b[1], b[2], b[3].
                   // Number of  elements in the array is b.length.
b[k]= b[j]+2;    // Evaluate b[j]+2, store the value in b[k].
```

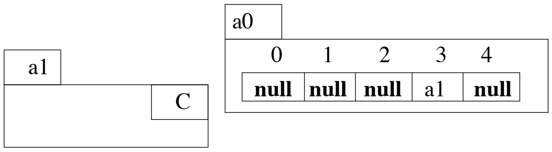


Array: an object that contains a bunch of variables of the same type.

```
C[] b;           // Declaration of variable b of class-type C[]  (C array).
b= new C[4];  //  Create and assign to b an object that is an array of 4 C vars.
```



b[3]= **new** C(); // Create and assign to b[3] a new folder of class C.


a1
C
a0
0
1
2
3
4
null
null
null
a1
null