Lecture 7

Methods ctd.
Objects
Reader Ch 20–24

Using methods of existing classes

- System.out.println, scanner.nextInt(),...
- String s = "abc";
  s = s.toUpperCase();
- See http://java.sun.com/javase/7/docs/api/
  for all built-in Java classes (a lot!)

Organization

Now we can:

1. Name elementary data (variables)
2. Group and name data of the same type (arrays)
3. Group and name "behaviour" (methods)
4. Group and name methods (classes)

- Goal: group and name arbitrary data together with behaviour

More organization possibilities needed

suppose we have students with names and grades: how to organize this?

- grades of students: `int[] grades`
- names of students: `String[] names`
- connection: same index, same student
- ☹ not visible that they belong together, hard to maintain, what if there are more student groups? ☹

Objects

- Two purposes:
  - an object represents (models) a thing in the "outside world" (domain):

    club member, invoice, playing card, ...

  - an object models a compound piece of data

    vector, list, ...

5

Objects with data

- Example: Garden

- Data: width and length

```
class Garden {
  double width;
  double length;
}
```

- One garden:

```
Garden g;
g = new Garden();
g.width = 10;
g.length = 12;
```

- More gardens

```
Garden h = new Garden();
h.width = 6;
h.length = 12
```

Objects with data

- Accessing data

```
double area = g.length * g.width;
System.out.println("Area first garden: "+area);
area = h.length * h.width;
System.out.println("Area 2nd garden: "+area);
```

- length and width are instance variables

  every instance (=object) has a **copy** of length and width

- you create instances with new

Objects with methods and data

- Accessing data

```
....g.length * g.width;
... h.length * h.width;
```

- Cumbersome, errorprone
- don't look under the gnome's hat ➡ use methods!

```
class Garden {
  double width;
  double length;

  double getArea() {
    return this.length * this.width;
      // or: return length * width;
  }
}
...
area = g.getArea();
```

- Or even: `System.out.println("Area first gdn: "+g.getArea());`
- In `this.length` is "`this.`" not required. It is recommended, though.

- We have three kinds of variables so far:

1. **local variables**
   declaration: in method body's
   use: direct
   scope: local (until closing brace)
   goal: store temporary results

2. **parameters**
   declaration: in method declaration
   use: direct
   scope: local (method body)
   goal: pass information from caller to method computation

3. **instance variables**
   declaration: in class
   use: via object
   scope: (most of the) class
   goal: store more permanent information

```
class Garden {
    double length; // meter
    double width;  // meter

    double getArea() {
        return this.length * this.width;
    }
    void setSize(double l, double w) {
        this.length = l;
        this.width = w;
    }
}

class Dorknoper {
    void doTax() {
        Garden g = new Garden();
        Garden h = new Garden();

        g.setSize(12, 10);
        h.setSize(20, 7);

        double area = g.getArea();
        System.out.println("the area of the first garden is "+ area);
        System.out.println("and of the second one "+ h.getArea() );
    }
}
```

```
public static void main(String[] args)
        new Dorknoper().doTax();
    }
}
```

The garden gnome metaphor

- Garden gnome does chores for you
- has a good memory
- Looking under the gnome's hat violates his privacy
- don't look under the gnome's hat ➡ use methods!

- Gnome is passive. Only acts when you ask!

More about objects

Objects and classes

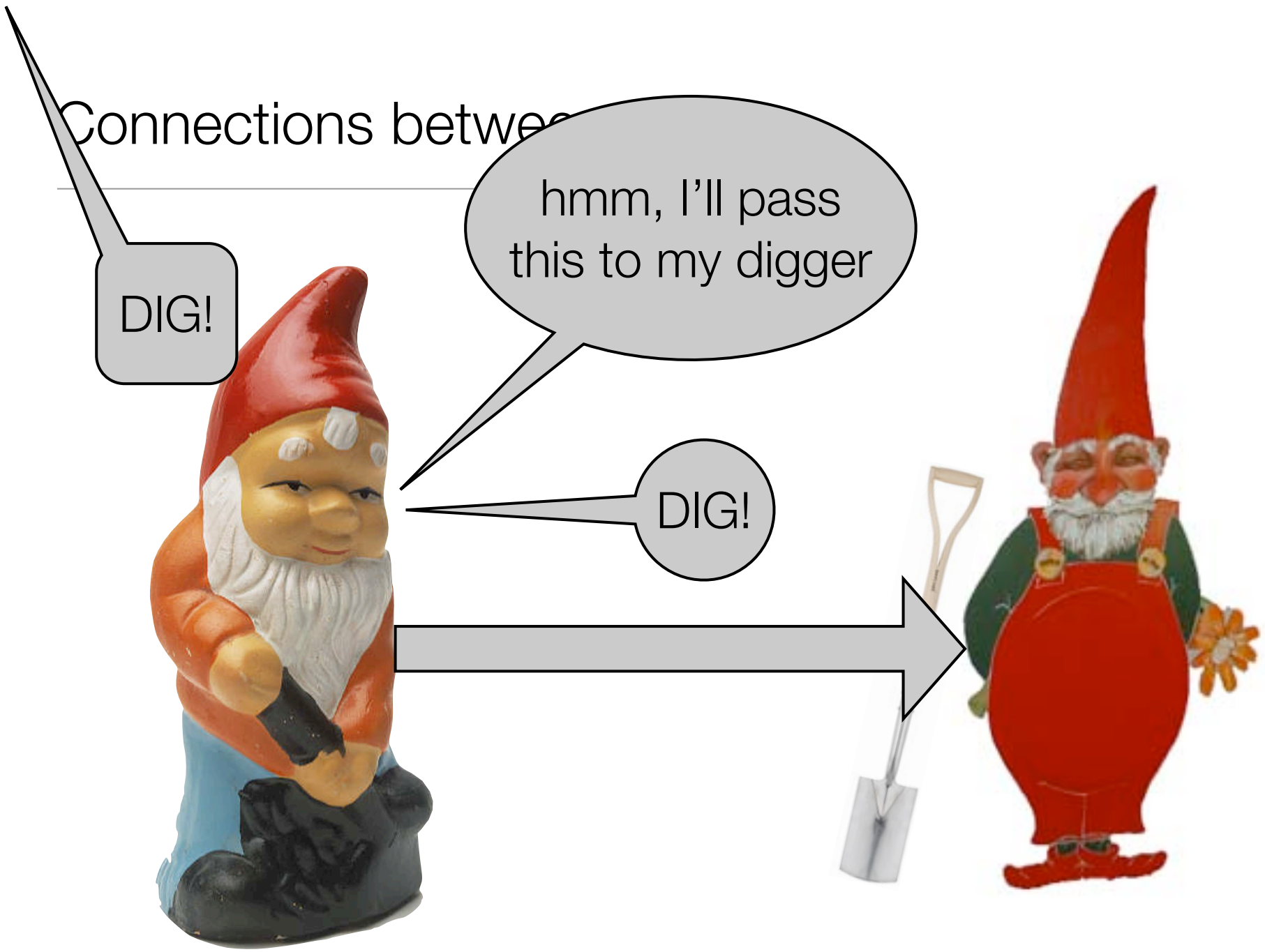
an object:

- stores data (in *instance variables*)
  - may contain connections to other objects
- offers services on this data (by *methods*)

a class:

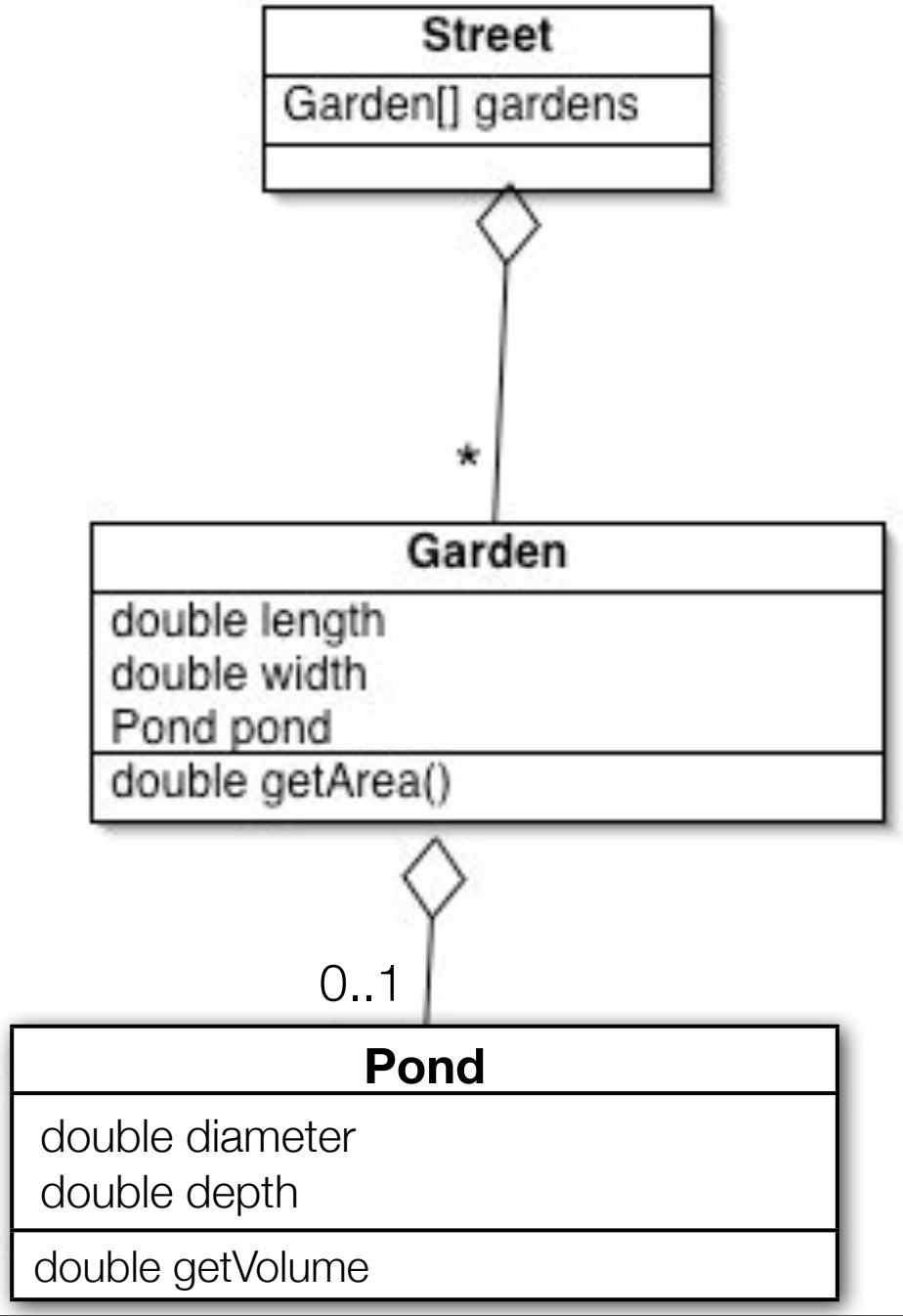
- describes a set of similar objects

Connections betwee

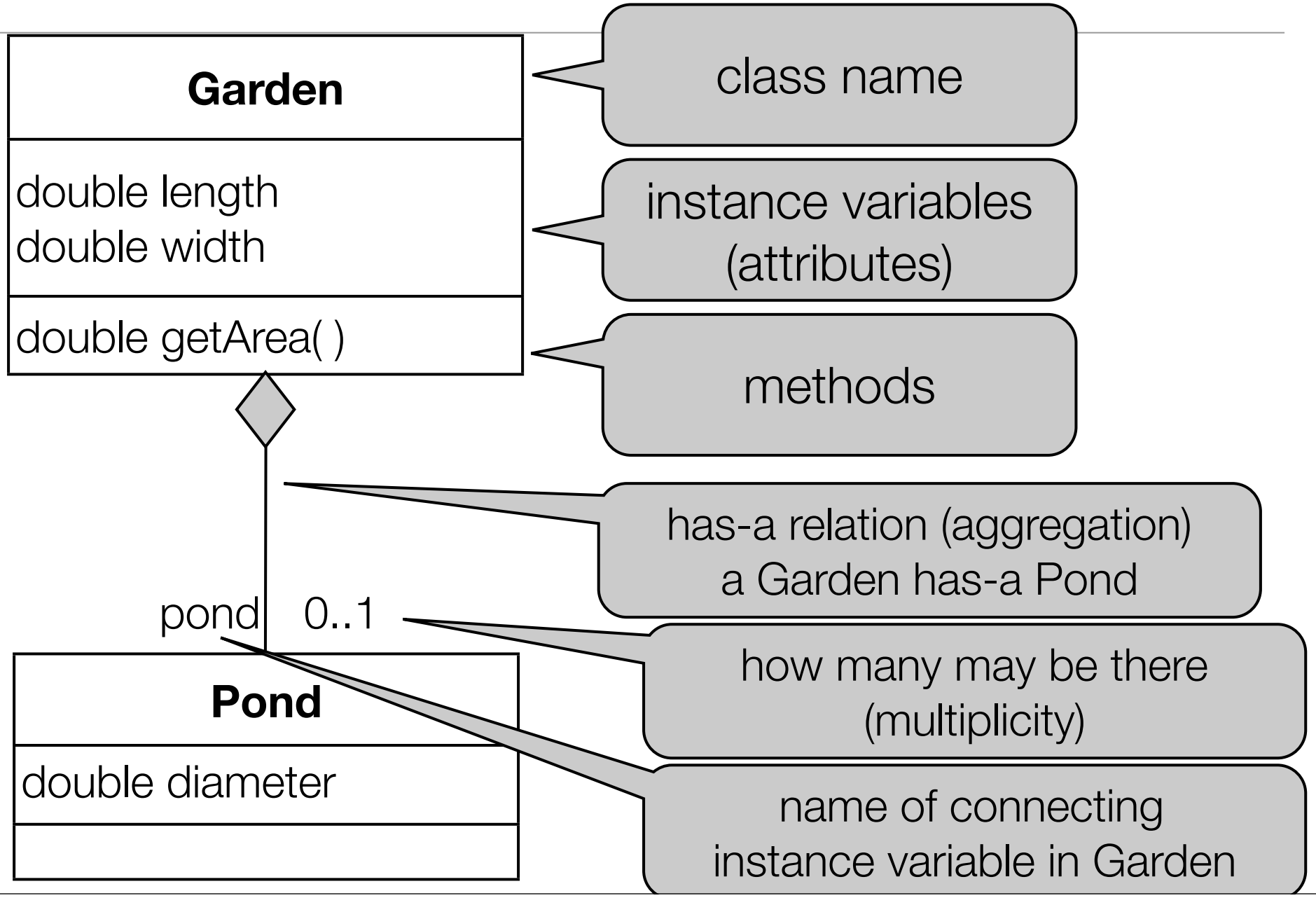


```
class Street {
    Garden[] gardens;
}

class Garden {
    double length;
    double width;
    Pond pond;

    double getArea() {
        return length*width;
    }
}

class Pond {
    double diameter;
    double depth;

    double getVolume() {
        double radius = diameter/2;
        return Math.PI *
            radius*radius*depth;
    }
}
```

Class diagram



Class diagram



Establishing relations between objects

```
Garden g = new Garden();
Pond p = new Pond();
g.pond = p;
```
etc.

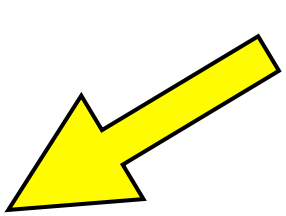
- Not very structured
- Idea: add methods to classes to organize this
- Proper moment: when objects are created, i.e. at *construction*

Constructors, references

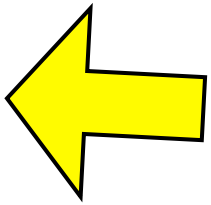
Constructors

- Goal: an easy and compact way to initialize objects

```
class Garden {
  double length;
  double width;

  Garden(double len, double wid) {
    this.length = len;
    this.width  = wid;
  }
  ...
}
...
Garden g = new Garden(12,10);
```



constructor



result is a Garden object with length 12 and width 10

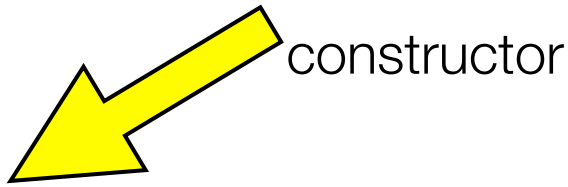
20

Constructors

- Convenient way to establish object relations

- and initialize other instance variables

```
class Garden {
  double length;
  double width;
  Pond pond;

  Garden(double len, double wid, Pond p) {
    this.length = len;
    this.width  = wid;
    this.pond = p;
  }
  ...
}
...
Pond nicePond = new Pond(1, 0.5);
Garden g = new Garden(12, 10, nicePond);
```



21

Constructors 2

- Differences between method and constructor:

1. a constructor has ***no return type,*** not even void
2. a constructor has ***the same name*** as the ***class***
3. a constructor can ***only*** be called at construction
4. a constructor can not be inherited (see later)

22

Constructors 3

- In one class there can be several constructors, as long as they have different parameters (as there can be methods with the same name and different parameters)
- The constructor with *no parameters* is called the *default constructor*
- The default constructor is automatically available when there is no constructor defined explicitly
- So it is not available anymore when you add another constructor
  - if you still want to use `new Blah()`, add a constructor with no parameters

23

References

- each Java object has a unique id, ca reference
- you don't see them, you can or em in variables and copy them

```
Garden g = new Gard
Garden h = g;
```

- g and h are different variables me value (reference)
- both point to the same objec

24

References

- each Java object has a unique id, called reference
- you don't see them, you can only store them in variables and copy them

```
Garden g = new Garden(4,5);
Garden h = g;
```

- g and h are different variables with the same value (reference)
- both point to the same object, hence...

25

Aliasing

```
Garden g = new Garden(5,3);
Garden h = g;

g.setLength(10);
```

- `System.out.println( g.getArea() );` prints...

  30

- `System.out.println( h.getArea() );` prints...

  30

  - both g and h contain the same reference, point to the same object, hence changes to g are sensed also by h

  the object is aliased (known by two "names")

26

Alisasing: why?

Why would you use aliasing?

- two parts of your code share an object
  - one part deals with the values, other part with visual appearance
- objects often represent real world things, e.g., an employee (record)
  - same employee object is shared by salary administration and by teaching schedule

27

null references

What happens when the variable pond is not initialized?

- It will have the special value `null`
- `null` refers to no object at all
- you can not call a method on `null`
- you can test on equality to `null`:

```
if (p != null) {
  x = p.getVolume();
}
```


28

Arrays of objects

Three steps:

1. declaration: `Garden[] gardens;`

2. creation of array: `gardens = new Garden[10];`

3. creation of objects:

```
for (int i=0; i<gardens.length; i++) {
    gardens[i] = new Garden(...);
}
```


29

Two fundamental kinds of types

1. **primitive types: int, double, ...**

   don't have to be created

   - are defined by their value
   - equality by `x == y` (`!=`)

2. **object types: anyhting else (including Strings)**

   - have to be created with `new` (Strings also by "...")
   - equality of contents by `x.equals(y)`
   - have own identity: same contents can have different identities
   - equality of identity (reference) by `x == y`
   - sharing (aliasing) is possible!

30

More animations