Lecture 9
Inheritance

Kees Huizing
October 2012

Inheritance 1

- Remember Garden. Suppose some gardens are used for growing vergatables. We get:

```
class MarketGarden {
  Vegetable[] crops;

  double getArea() {
    ...


  double printCrops() { ... }
```

- Some data and methods are the same as in Garden
- Idea: *reuse* Garden class instead of *copy*

Inheritance 2

- Existing class is extended with new data and methods:

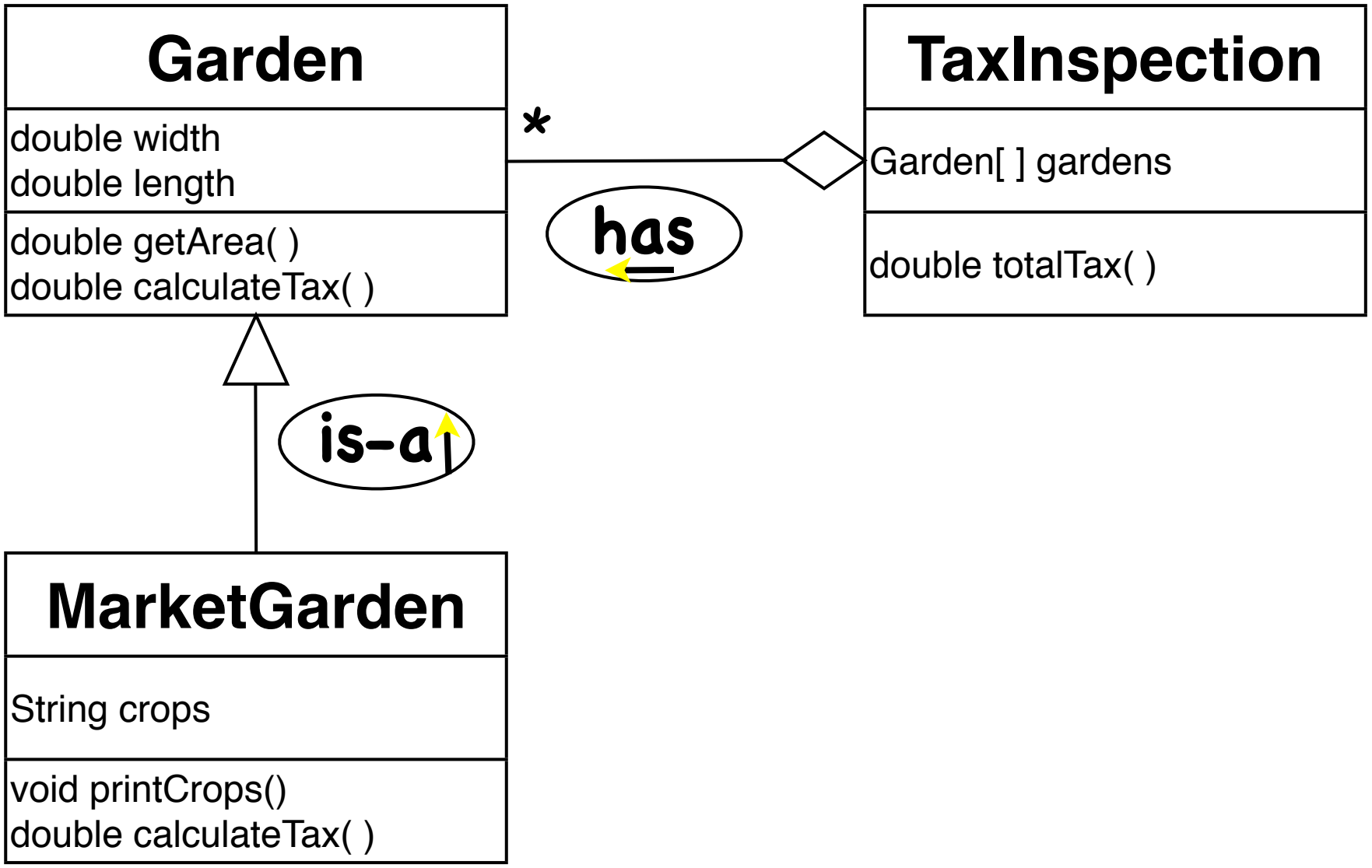
```
class MarketGarden extends Garden {
  String[] crops;

  void printCrops() { ... }
}
```

- Data (instance variables) and methods of Garden are "inherited" (≈copied)

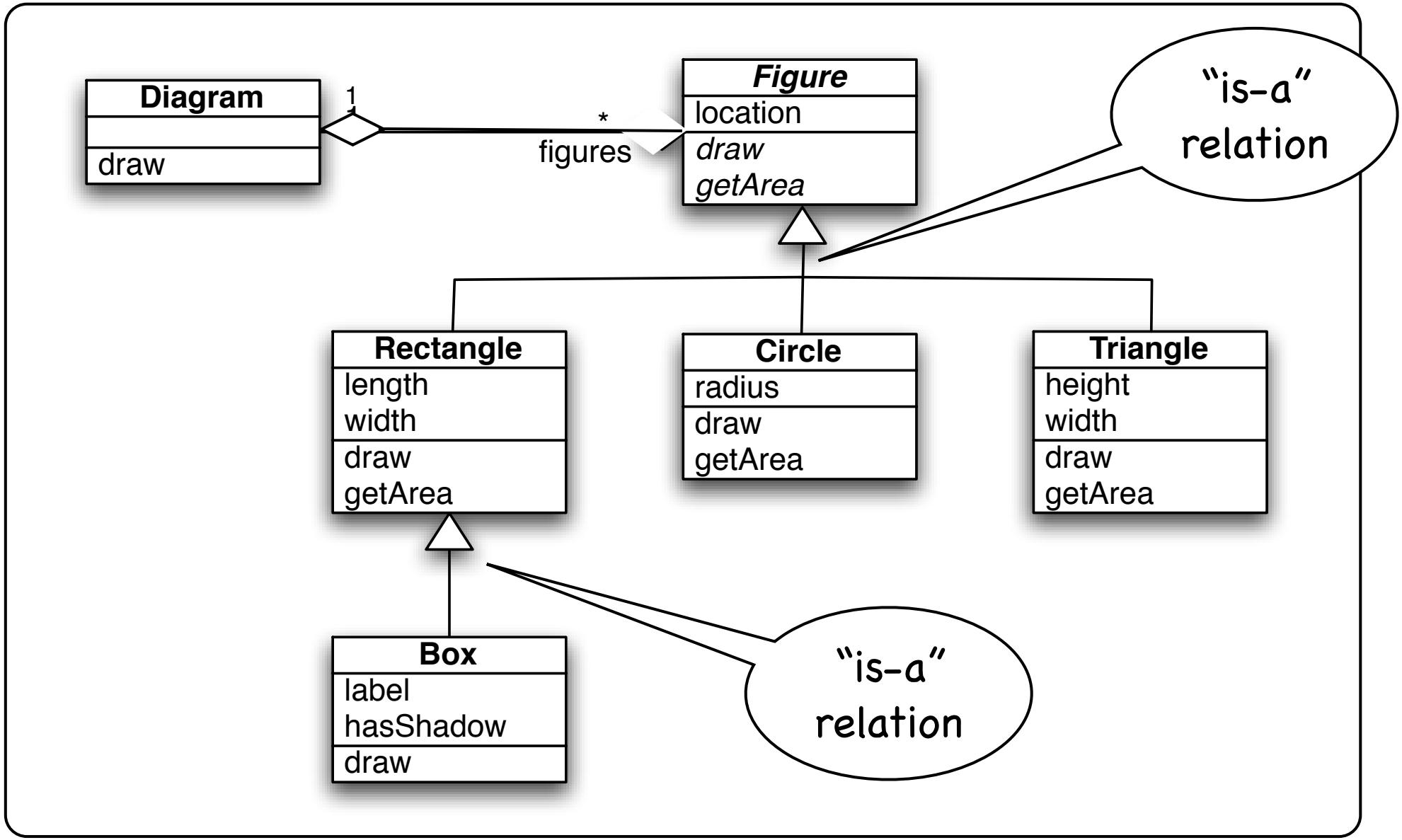
Inheritance 3

- new class is *subclass* of existing class (=*superclass*)
- methods can be *redefined (overriding)*;
- example of overriding: tax is different for an ordinary garden and a market garden; *both* classes will have a definition of `calculateTax`

Class diagram



Example Class Structure



Abstract classes

- Figure is abstract
  - you don't make objects (instances) of Figure
  - it helps organizing class hierarchy
  - cf. Mammal or MovableItem (in a game)

Inheritance and subtyping

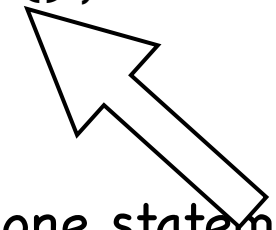
- A subclass has its own type *and* is also of the type(s) of its superclass(es). A MarketGarden is also a Garden.
- cf. a BullDog is a Dog is a Mammal
- You can declare a variable of supertype and let it refer to an object of subtype:

```
Garden g = new marketGarden();
```

Why subtyping?

- To write one code for different types: more flexibility

```
Garden[] gardens = new Garden[10];
gardens[0] = new Garden();
gardens[1] = new MarketGarden();
...
tax = 0;
for (int i=0; i<gardens.length; i++) {
  tax = tax + gardens[i].calculateTax();
}
...
```



one statement for two types! (and with two effects)

Why(2): Abstraction

Abstraction hierarchy

- soort: amorphophallus titanum
- geslacht: amorphophallus
- familie: araceae
- orde: alismatales
- clade: eenzaadlobbigen
- clade: bedektzadigen
- klasse: zaadplanten
- stam: landplanten
- rijk: planten

bron: nl.wikipedia.org 2008 (er zijn verschillende taxonomieën)

Abstraction Hierarchy
Taxonomy

- Kingdom: Plantae
- Clade: Angiosperms
- Clade: Monocots
- Order: ALismatales
- Family: Araceae
- Subfamily: Aroideae
- Tribe: Thomsonieae
- Genus: Amorphophallus
- Species: Amorphophallus titanum

Overriding

- a method defined in a subclass is said to override a method in the superclass if it has the same signature, i.e., name and parameter types)
- use overriding to
  - adapt behaviour to accommodate extensions
  - re-implement method to accommodate other data
  - implement method that is only abstract in superclass, e.g., in Figure:

```
abstract double getArea();
```

Overriding ctd.

- make sure that behavior of subclass method is the same (on right level of abstraction) or an extension
- method of superclass is accessible from overriding method with super
  e.g., in MarketGarden

```
super.calculateTax(…);
```

will use tax method of Garden

Super

- use super to reuse code from superclass instead of copying
- in method:

```
tax = super.calcTax() + crops.length * 10
```

- in constructor:

```
MarketGarden(double len, double wid, String[] cr) {
  super(len, wid);
  crops = cr;
}
```

Super ctd

- In methods:
  call method on super, similar to this
- In constructors:
  - super is call itself (takes parameters)
    - this(....) is also possible
  - always first statement of constrcutor body

remark: if first statement of constructor is not this(...) or super(...), the statement super() is filled in by the compiler