<< Lecture 14 >>

Summary

- We are soon to finish the second iteration
  - We will solely focus on data structures afterwards
- Today's lecture:
  - Covariance and contravariance
  - Types as sets, methods as functions
  - Class diagrams
  - Abstraction and polymorphism, visualized

Short word on constructors

- Every constructor calls the constructor in the superclass
- If not using extends, you are automatically extending Object
  - All objects in Java are also of type Object
- First line of constructor must call super(...), otherwise super() is called anyways
- If superclass does not allow empty constructor (i.e., you defined constructor with arguments only), the subclass will not compile until you call super(...) in their constructors

Covariance and contravariance

- Consider class SuperClass with method funk:

```
interface SuperClass {
   public TypeU funk( TypeV arg );
}
```

- Which types should class SubClass implement for funk?

```
Class SubClass implements SuperClass {
   public TypeX funk( TypeY arg ) {...}
}
```

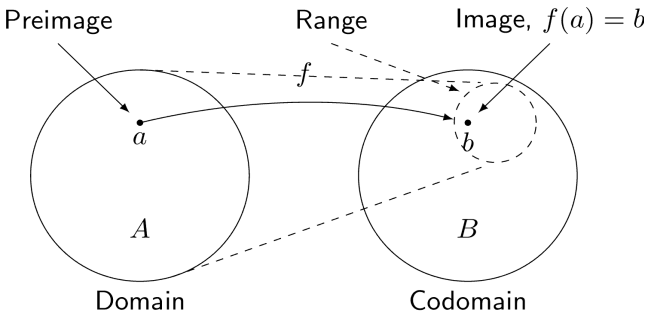
- Question: what are the relations between TypeU and TypeX, and between TypeV and TypeY?
  - **Ans**. we have that SubClass.**funk** must work in the following setting:

```
SuperClass sc = new SubClass();
// we can do the following, because, due to abstraction, we can use the
// method as promised in the type being used (SuperClass for variable sc)
TypeU u = sc.funk( new TypeV(...) );
```

    - First, we see that an object of type TypeX can be stored in an object of TypeU
        - TypeU must be supertype of TypeX (or, **TypeX *is-a* TypeU**)
    - Second, we see that SubClass.funk can accept objects of TypeV as parameter, but since it can accept objects of TypeY,
        - TypeY must be supertype of TypeV (or, **TypeV *is-a* TypeY**)
        - **Note that these subtypes relations are reversed!**
- **COVARIANCE**: subtyping in the returned type
    - *A method covering a supertype's method should subtype its return type, or use the exact same type*
- **CONTRAVARIANCE**: subtyping in the argument types
    - *A method covering a supertype's method should supertype its argument type(s), or use the exact same type(s)*
- Can do with arrays and constructors (go figure!)
    - When using **new**, we are using constructors, so, in a way, we are using covariance and contravariance—see next section for a more conceptual explanation

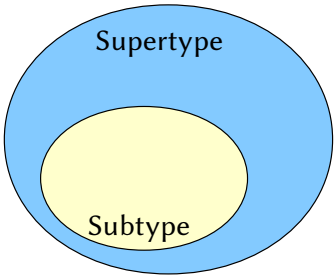
Parenthesis: types and methods as sets

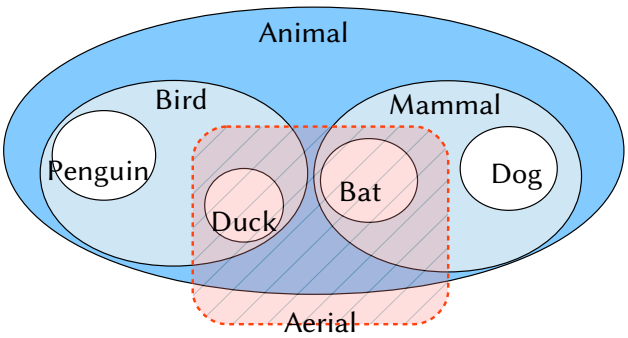
- We can think of methods as functions in the more mathematical sense:
    - They have domain, codomain, and (effective) range
        - Domain: the space of parameters/arguments
        - Codomain: the space of the images (f(x))
        - Range: actual portion of the codomain used (a.k.a., *effective range*)
            - *Lesson*: range ⊆ codomain
        - Question: for function f:X⟶Y, what are its domain, codomain, and range?
            - **Ans**. for f:X⟶Y, its domain is X, its codomain Y, and its range is $\{f(x) : x \in X\}$ or, equivalently, f(X)



A function, $f : A \to B$.

- Can you express the following in the f:X⟶Y notation?
  - static double squareIt(double x);
    - **Ans**. squareIt: double ⟶ double
    - If we consider that squareIt does x ⟶ x^2, then the effective range are the positive doubles!
  - static int maximum(int x, int y, int z);
    - **Ans**. maximum: int ⊗ int ⊗ int ⟶ int
    - I'm using ⊗ for the Cartesian product—this product just creates sets tuples, triples, etc.
  - static String reverseCase(String x);
    - **Ans**. reverseCase: String ⟶ String
  - new String();
    - **Ans**. new String: ∅ ⟶ String
  - (SomeObject)
    - **Ans**. (SomeObject): Object ⟶ SomeObject
    - Casting can be applied to an object of any type (i.e., to any Object), and the resulting type is the type specified in the parentheses (i.e., SomeObject)
  - A.length, where A is a char[]
    - **Ans**. char[].length: char[] ⟶ int
    - Indeed, a method from an object uses the type of the object itself as a parameter
      - *This is the philosophy behind's Python method definition in classes! If you remember, a method defined inside a class must first have* **self** *as a parameter*
  - B.equals(Object b), where B is of type Bacon
    - **Ans**. Bacon.equals: Bacon ⊗ Object ⟶ boolean
- We can express type relations through Venn Diagrams
  - If types/classes are sets, then values/objects are the elements of these sets

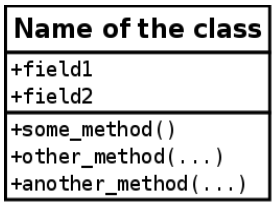


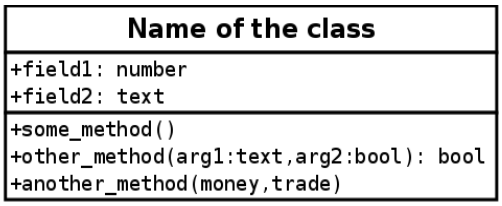


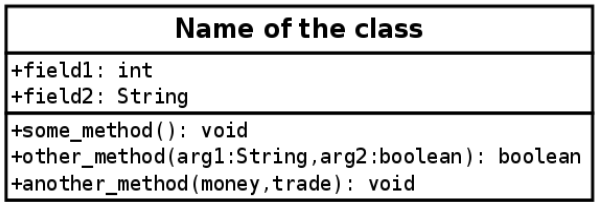
- We can explain covariance and contravariance through sets also:
  - Covariance: if f is to be valid on every element of the domain, having f^{ext} valid on the codomain + more elements meets the requirement; hence, if the domain of the new method is a supertype of the original domain, we meet the requirement (or we could use the original type)
    - Original restriction: $f(X)$ must be valid, i.e., $f(x)$ must be valid for all x in X
  - Contravariance: since the range is a subset of the codomain, we can see that using any subtype of the original returned type meets the requirement (or we could use the original type)
    - Original restriction: $f(X^*) \subseteq Y$, for Y being the original codomain
- More distraction:
  - http://www.cs.toronto.edu/~sme/presentations/cat101.pdf
  - https://en.wikipedia.org/wiki/F-algebra (looks so clean!)

Sketching classes

- Before jumping onto giving more examples of polymorphism and hierarchies, let us discuss a very useful tool in software design: class diagrams
- Diagrams? Illustrate software components, type/class hierarchies, class structure, etc.
- Often times, you can convert diagrams to proper code in one step—little functionality, but useful for prototyping!
- Helps with *domain modeling*, and *domain driven software engineering*—goal: to describe a reality through code

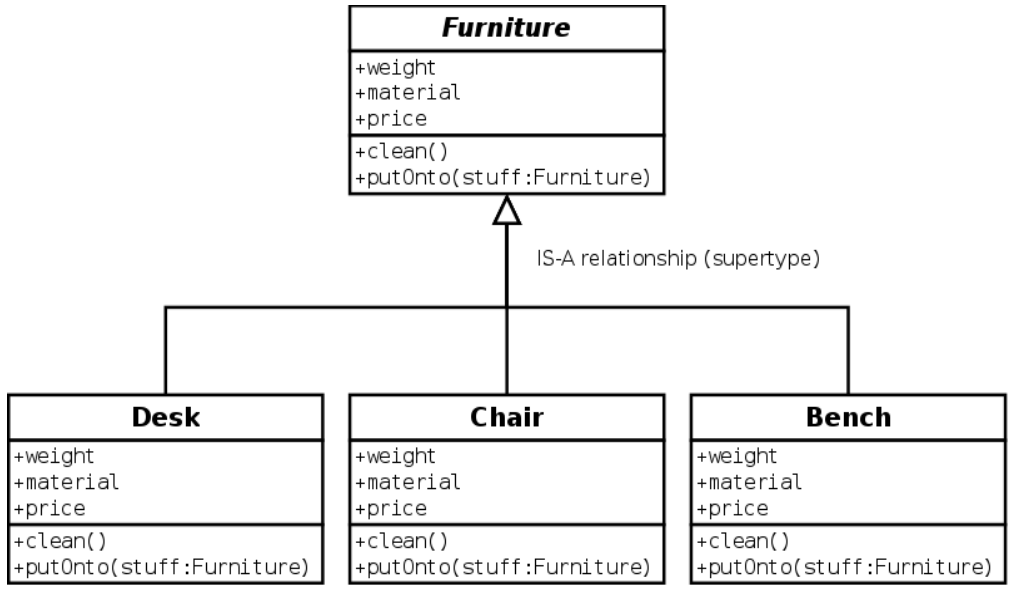






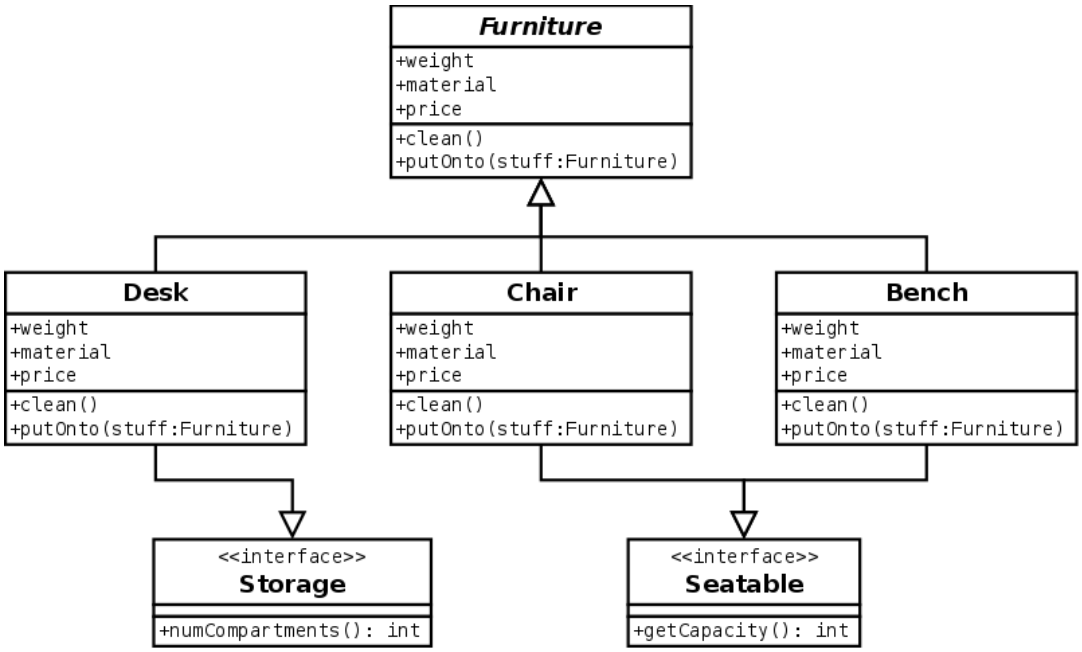
<< A class is depicted in a box of three sections: the name, the fields, and the methods >>

<< Different levels of detail/abstraction for describing the same class >>



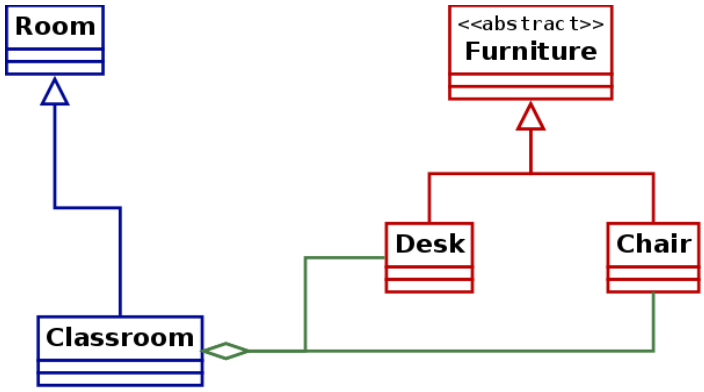
<< Class hierarchies are explicit through the IS-A relationship >>

<< The IS-A arrow has an empty head and points to the immediate supertype >>



<< The IS-A relationship is also used with interfaces >>

- Conversion to code is straightforward!
- Diagrams can say a lot more, such as composition

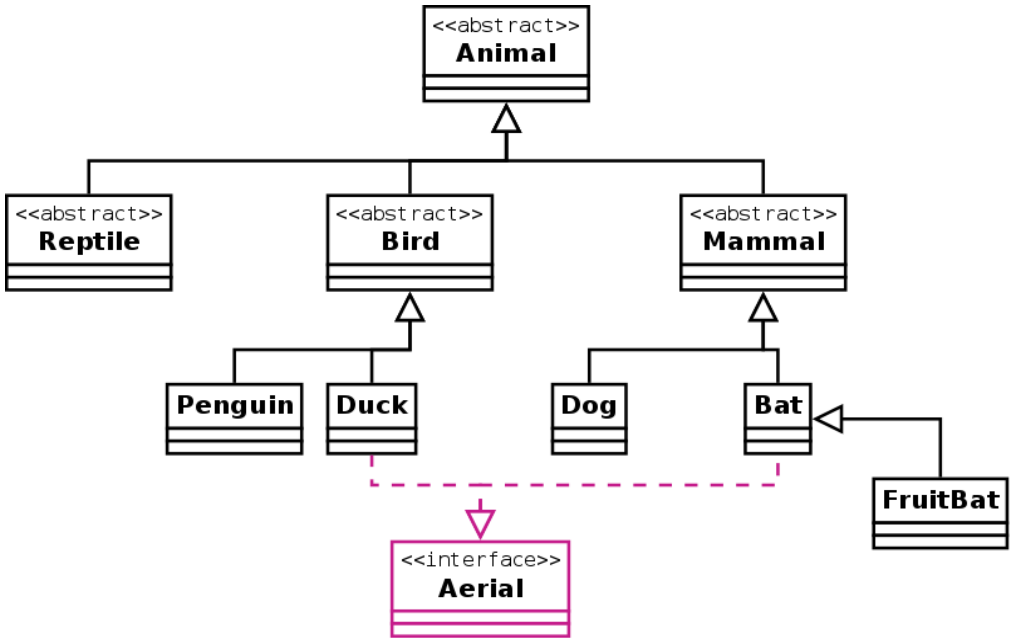


<< A Classroom is [not only] composed of Desks and Chairs >>

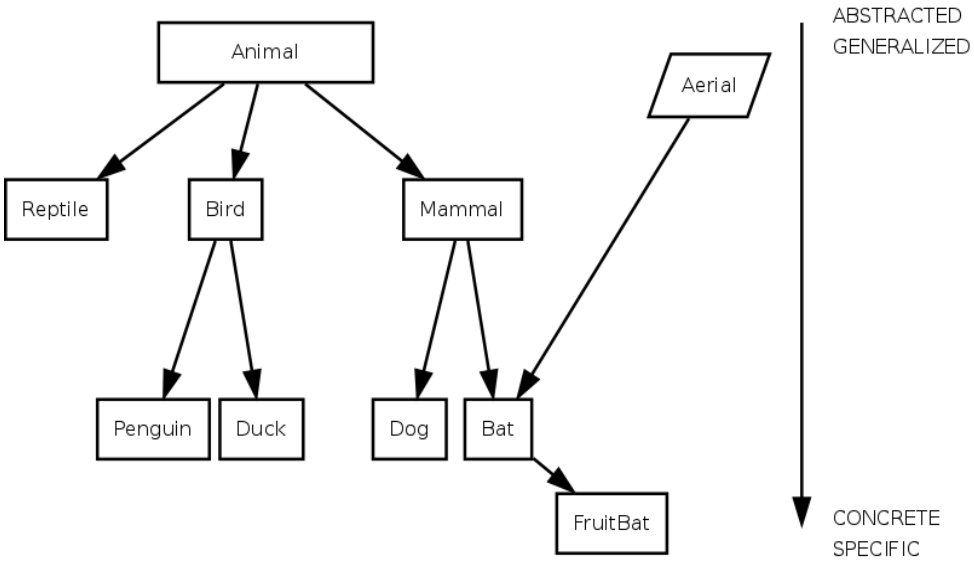
- Composition can be implemented as and array or list (we are almost there!) or other collection
- Even though proper diagrams can be converted to code, they can be an obstacle to iterative improvement of the code
  - Once classes have code, will you delete their contents just because of a slight redesign?
- Lesson: don't worry too much about formalities in diagrams, use them as sketches

POLYMORPHISM EXAMPLE: ANIMAL HIERARCHY

Remember the Animal hierarchy introduced and used in previous lectures? Well, we just drew the portion of the hierarchy we implemented so far in the diagram below. Because we wanted to illustrate the hierarchy, we drew a class diagram devoid of methods and fields.

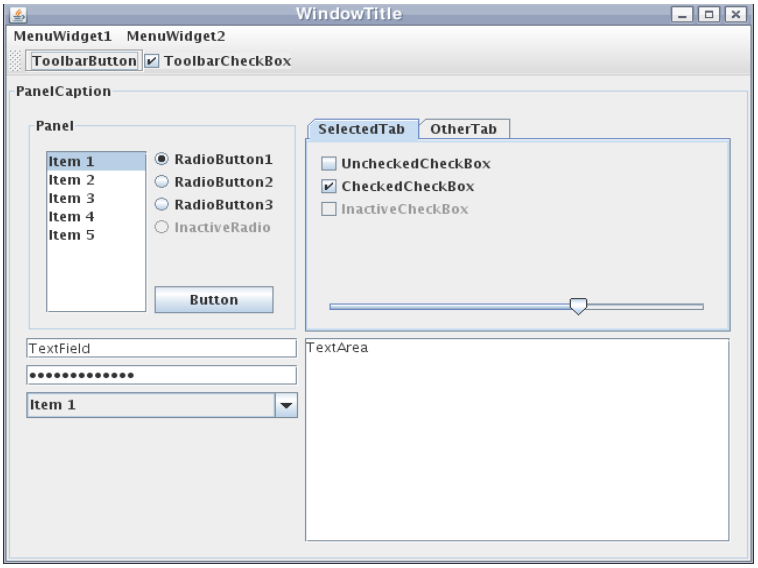


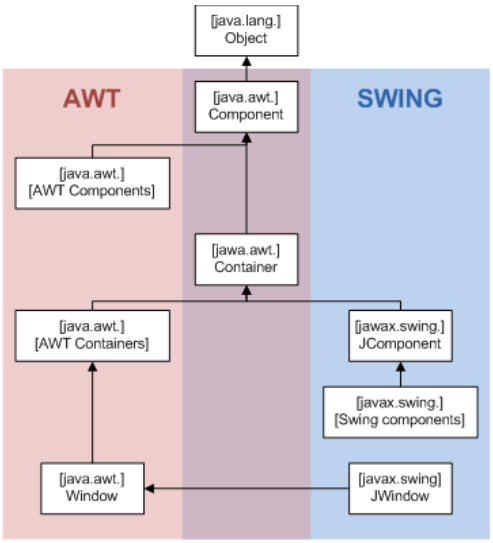
Now, the Animal hierarchy serves as an example for the concept of *abstraction*. In polymorphism, abstraction is addressed by the supertype-subtype relations. Following the path of subtyping, as in the image below, we get increasingly more specific *concepts*. Besides being more specific as concepts, their behavior is more specialized and concretely implemented. The path of supertypes, on the other hand, show how concepts are contained in more simplified or abstracted concepts, which are more general than the specific ones.



Polymorphism example: user interfaces

User interfaces are a great example of polymorphism and abstraction. They might be composed of the same elements in general, but the *components* are arranged in different ways and portray different information and behavior. But, when an event occurs, they must behave in a compatible manner.





Let us consider the case of a *mouse click event* on one the checkboxes. First, the window received the event. It checks the location of the mouse, and decides it has to be addressed by the *panel* rather than the *menu bar*. The panel looks its *layout* and realizes that the click should be handled by the subpanel with *tabs*. The tabs-panel receives the event and, by knowing which tab is active, it delegates the event to the contained *panel*. And this goes on until reaching the *checkbox*. Then, the checkbox changes its state (true to false or viceversa) and calls for the whole window to be redrawn.

All the components behave similarly with respect to handling events. That is abstraction in action. However, there are many types of components (i.e., many subtypes of *Component* or *JComponent*). That is polymorphism.

Also, note that the components are of very different nature. Checkboxes deal with boolean values. Scrollbars with integer values (or doubles). Text areas deal with Strings, and so do text fields and password fields. Radios deal with exclusive elements in a list. Lists, drop down menus, tabs, etc., also deal with elements in a list. Labels and buttons do not handle data. Et cetera. But in spite that their behavior is so complex, all components can be generalized as boxes that are drawn and that receive events from the keyboard and the mouse.

Besides abstraction, *how do GUIs address the object oriented design principles?*

<< Lecture 15 will resume from here >>