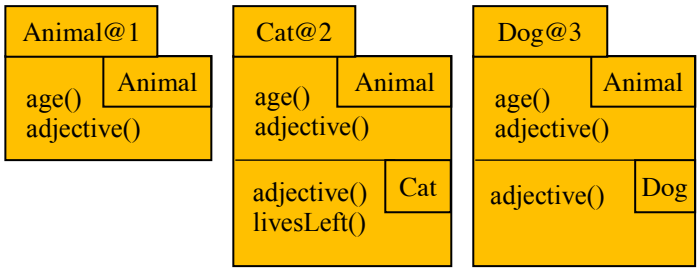
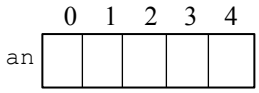
Subtype Polymorphism

Consider a class `Animal`, with subclasses `Cat` and `Dog`. All animals have an age, so class `Animal` has method `age`(), which returns the animal's age. Cats are said to have nine lives, so class `Cat` has a method `livesLeft`(), which returns the number of lives left. Different kinds of animals have an adjective that is another way of identifying the kind. Dogs are *canine*, cats are *feline*, and animals are *zoic*. Therefore, all three classes have a method `adjective`() that returns the adjective (return type `String`). Methods `adjective` in `Cat` and `Dog` override method `adjective` in `Animal`.



It would be nice to have an array of different animals —cats, dogs, and any other kinds of animal we might have implemented. But does the following declaration do it? (We also draw the array to the right.)

```
Animal[] an= new Animal[5];
```



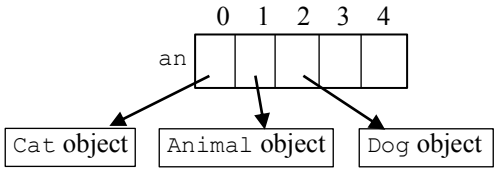
It looks like the array elements can contain only pointers to objects of class `Animal`, since each array element has type `Animal`. What if we wanted a list of a family's pets? We would need an array for the dogs, another array for the cats, and later, as the family got more pets, separate arrays for the fish, the minnows, the parrots, the iguanas, …. This would be extremely awkward and cumbersome.

But *subtype polymorphism* saves the day. Here are two rules about Java:

- Subclasses are subtypes. For example, `Cat` and `Dog` are subtypes of type `Animal`.
- Suppose `S` is a subtype of `T`. Anywhere a value of type `T` is required, a value of type `S` may be used.

Thus, at runtime, execution of the following assignments changes the array as shown to the right below:

```
an[0]= new Cat(…);
an[1]= new Animal(…);
an[2]= new Dog(…);
```



Why is this polymorphism? Type `Animal` comes in different forms: type `Animal` itself, subtype `Dog`, and subtype `Cat`, and perhaps others. All of them are treated the same way, as `Animals`, but they exhibit differences.

This polymorphisms brings up two questions for us. First, at compile-time, what method calls should be allowed? Second, at runtime, which methods are actually called?

A problem with missing methods at runtime

Consider the method calls shown to the right, given array `an` as shown above, with pointers to objects in `an`[0], `an`[1], and `an`[2]. The first call appears to be OK; since the `Cat` object contains method `livesLeft`(). But the second call is not OK, since the `Animal` object doesn't have a method `livesLeft`() to call. If this were allowed at runtime, an exception would be thrown.

```
an[0].livesLeft()  —seems OK
an[1].livesLeft()  —not OK
```

The designers of Java decided that checking at runtime whether a method existed was not a good idea. Instead a program will be compiled —it is syntactically legal— only if it is *guaranteed* that a method exists at runtime for every method call. Thus, the call `an`[1].`livesLeft`() —in fact any call `an`[`k`].`livesLeft`(), will not compile.

Below, we will use the term *static* type for *type*, to stress the fact that the type of a variable or expression is a property of the program text; *static type* has nothing to do with runtime. We will then use the term *dynamic type* for the type of the value of an expression at runtime; *dynamic type* has nothing to do with compile-time. The dynamic type of an expression can change as values are assigned to variables.

Subtype Polymorphism

Compile-time: guaranteeing that a method exists

The static type of each element of array `an` is `Animal`. The simplest way to guarantee that a method exists at runtime for each call `an[i].m(…)` is to require that `m` be declared in or are inherited by class `Animal`. That is what Java does:

> **Compile-time reference rule.** For a variable `p` declared as
>
> `C p;`
>
> (where `C` is a class or interface), the method call `p.m(...)` (or variable reference `p.v`) is syntactically correct and can be compiled only if `m` (or variable `v`) is declared in `P` or is inherited by `P`.

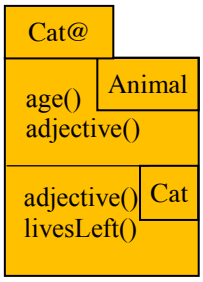
Casting about

An expression that yields (a pointer to) an object, like **`new`** `Cat()` , can be cast to other classes, e.g.

`(Animal) (new Cat())` and `(Cat) an[i]`

At this point, look at the entry for casts in JavaHyperText and read the pdf file found at item 3, *Casting with classes*. Stop on the second page, before the section on testing whether a downward class will work.

Runtime: which method is called



Suppose `an[0]` contains a pointer to the `Cat` object shown to the right. The object contains two `adjective` methods. Which one is called when `an[0].adjective()` is executed? This depends on the *dynamic type* of the expression `an[0]`.

The *dynamic type* of the expression is the type of the value of the expression. At this point, the dynamic type of expression `an[0]` is `Cat`. Therefore, method `adjective` in partition `Cat` is called. There are actually three possibilities, using our `Animal-Cat-Dog` classes:

If `an[0]` points at an object with dynamic type `Animal`, `an[0].adjective()` returns "zoic".

If `an[0]` points at an object with dynamic type `Cat`, `an[0].adjective()` returns "feline".

If `an[0]` points at an object with dynamic type `Dog`, `an[0].adjective()` returns "canine".

You may have already known this because we discussed it in terms of the *overriding rule*. Because of the way we draw objects, to find the method to call, start at the bottom of the object —the partition for the dynamic type— and search upward until the method is found.

Defining feature of OOP

At runtime, the method to call is determined by the dynamic type of the object being pointed to, and of course the object being pointed to can change often during execution, as variables are assigned different values. This is a most important feature of OOP. Without it, OOP would not be as effective and would not be so widespread today.

Instanceof and getClass()

At this point, look at the entry for instanceof in JavaHyperText and read the pdf file found in item 1. *Full explanation*.