## Polymorphism (continued)

Last lecture I introduced the idea of a "class descriptor" which is an object of the `Class` class. Today I will continue that discussion and elaborate on the details.

When you run a Java program, you are running the `main` method of some class. A stack frame for the `main` method is put on the call stack. As the instructions in the `main` method are executed, objects may be created. The `main` method may also call other methods. Each time another method is called, a new stack frame goes on the call stack, and when the method returns, the frame of that method gets popped from the call stack. I have discussed this idea several times throughout the course so you should be familiar with it.

Each method including `main` can have local variables. When a method is called, a copy of the local variables for that method are in the stack frame for that method call. I say "a copy" because there may be multiple stack frames for that method, such as in the case of recursion. Each call to a method requires local variables for that call.

The stack frame for a method call also keeps a copy of the parameters that are passed to it. Each stack frame also keeps track of the `this` "variable" which references the object that called the method. (As we will see next lecture, some methods are `static` and aren't called by an object, so the stack frame of these `static` methods don't have a `this`.)

Finally, all of the variables in the call stack can be either primitive types or they can be reference types. In the former case, the values of the variable are written in the stack frame itself. In the latter case, the reference value i.e. `Object.hashCode()` is stored in the stack frame and the object that is referenced is stored somewhere else, namely some "object area" space. All you need to know about this latter space is that:

- each object occupies a consecutive sequence of bytes in that space, and the size of each object depends on the class that this object is an instance of;
- the bytes are numbered and we refer to the numbers as "addresses"; each address is a 32 bit number;
- each object starts at some address and when we use the `==` operator on two objects, we are testing if the starting addresses are the same.

## Example

In the slides, I walked through a simple example. Suppose we have a `TestDog` class which has a `main` method with the following instructions:

```
Dog    myDog = new Beagle();
       myDog.bark()
       myDog.getOwner()
```

See the illustrations on the slides of what happens when these instructions are executed. In particular, what happens on the call stack? The first stack frame is `main` and it has a local variable `myDog` that is initialized to `null`. Then the `Beagle` constructor is called which creates a new object. The variable `myDog` references this new object, and the `Beagle` stack frame is popped.

The `main` method continues and the `bark()` method is called by `myDog`, so the `bark` stack frame is pushed on the stack. But which `bark` method is called? Note that the answer must be determined before the `bark` stack frame is put on the call stack. The answer is determined by "asking" the object referenced `myDog`: which class do you belong to. The answer is `Beagle`. One (the JVM) then checks if `Beagle`'s class descriptor contains a `bark()` method; if not, it proceeds to the `Beagle`'s superclass and upwards until the `bark()` method is found. Since the `Beagle` class descriptor does indeed have a `bark()` method, the JVM uses it.

In the next instruction, `myDog` calls `getOwner()` which simply returns a person object.[1] The point here is that this method is in the `Dog` class and so this method is inherited by the `Beagle` class. So, when the method `getOwner` is invoked, the JVM finds this method by first checking the `Beagle` class descriptor (method not found) and then proceeds to the `Dog` class descriptor (method found). A stack frame for the method `getOwner()` can go onto the call stack and the method can be executed.

I mentioned that local variables in a method (including parameters passed to the method) are stored in the stack frame. What about the variables that are fields in the class (such as `owner` or the `name` of a `Dog`). Where are these fields stored?

When a new `Beagle` object is created as in the above example, it has fields that are defined in the `Beagle` class (if there are any) and fields that are inherited from the `Dog` class, such as `owner` and `name`. Since the values of these variables are specific to the object (rather than to the class, the superclass,or particular method that is being run), the values are stored in the Dog object. When I say "stored" here, I just mean that these reference variables are in the object. The objects that they reference such as the `String` object that `name` references or the `Person` object referenced by `owner` would be separate objects. My figures in the slides did not show these objects.

One final note about where things are located: the Java instructions for each method are in the form of compiled "byte code". This code is located in the class descriptor for the class that defines the method. (Although each object has a set of methods that it can invoke, do not think of these methods as part of the object. It would be wasteful to repeat the same instructions in each object – imagine you had thousands of instances !)

## Garbage collection – the "mark and sweep" algorithm

It often happens that we create objects and use them, but at some point we no longer need them and indeed we no longer reference them. The example I gave in the lecture was:

```
Dog myDog = new Beagle(Bob);
    myDog = new Terrier(Tim);
```

In this case, there will be no way to access the beagle Bob in the program. We say that Bob the beagle is *garbage*. We would like to be able to reuse the space that this object is taking up and write another object in that space. To do so, we must free up that space and remove the object.

[1] I had mentioned earlier that all dogs have owners and so it makes sense to have a `Person owner` field in the `Dog` class. You can imagine all domestic dogs need a license from the city and the owner has to be listed on the license.)

In the slides, I illustrate that the JVM needs to keep track of all the objects in a program. For simplicity I used a linked list data structure of all the objects, but there is nothing particularly list-like about the object collection and we could use any collection data structure. I use a linked list because it is easy to illustrate.

Garbage collection is done by the JVM when the object space fills up.[2] To remove the objects that are garbage, they must be identified. An object is garbage if there is not way to reference that object from the program. This means that there are no references to the object from variables in the call stack, and there are no references to the object from objects that are referenced from variables in the call stack, and there are no references to the object from objects that are referenced by objects that are referenced by variables in the call stack, etc.

The JVM defines a garbage object in terms of a *graph* whose vertices are one of the following: (1) a variable in the call stack or (2) a variable in an object or (3) an object itself. The edges in the graph are of the form (reference variable, object) where the reference variable vertex can either be in the call stack or in an object. See the slides for an illustration of these edges.

Garbage collection is done by finding all objects that are *not* garbage, that is, objects that are reachable in the graph starting at vertices in the call stack. To find such reachable object, the JVM builds a graph described above and then traverses it from each vertex that corresponds to a reference variable in the call stack. We say that reached objects are *marked* as *live*. The JVM then removes objects that are not marked. This removal is called the *sweep* step. The basic idea for sweep is to just remove the corresonding node(s) in the linked list of live objects. See the illustrations in the slides.

In the slides, I defined a second type of list which keeps track of the gaps between the objects. When an object is removed, it leaves a gap and if that object was adjacent a gap then when the object is removed it increases the size of the gap. As the slides show, theJVM keeps track of a list of the gaps as well as the objects, i.e. two different lists.[3] After garbage collection, when the program resumes running, new object instances can be created and if a big enough gap is found, the new object can be put into the gap.

Garbage collection takes time. If a program is running and needs garbage collection because there is not enough space left for new objects, then the program will need to pause while garbage colletion occurs. To avoid that this disruption takes a long time, garbage collection should be done relatively frequently. (Of course, it has to be done efficiently too. Details omitted for lack of time – and because these "systems" issues are not the main point here.)

[2]As far as I know, the Java language does not specify exactly when garbage collection should be done, so I won't discuss that here

[3]In class, students asked for details on how this is done, such as: does the JVM rearrange the objects in memory so that it has a small number of bigger gaps? As far as I know, there is no specification of this, and it depends on the implementation.