Book: Head First

Chapter-7:
Better Living in Objectville

Upcode Software
Engineer Team

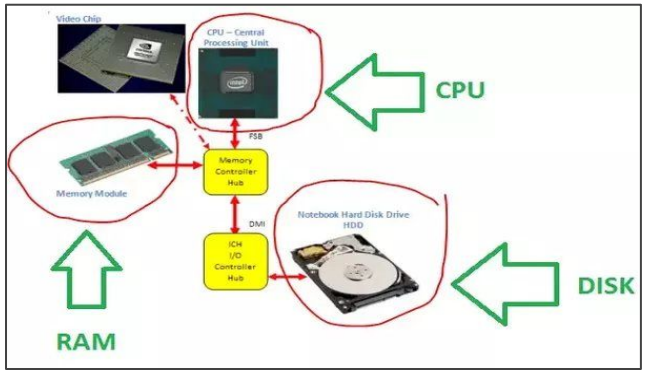
-2023-

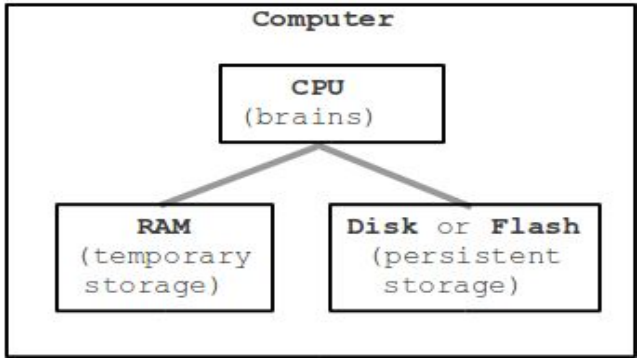
CONTENT

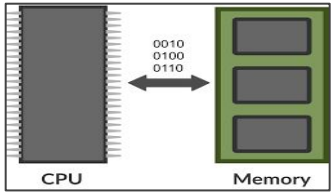
1. Java Memory Model
2. What is OOP ?
3. Why Understanding Inheritance ?
4. Using IS-A and HAS-A?
5. What is a polymorphism ?
6. Summary

The Internal Java Memory Model(1/n)

- ***CPU* percentage**: to show how much percentage of **processor is** being used.
- ***Memory* percentage**: to show how much percentage of inside memory (**RAM**) is being used.
- ***Disk* percentage**: to show how much percentage of **storage** (HDD, SSD) is being used.

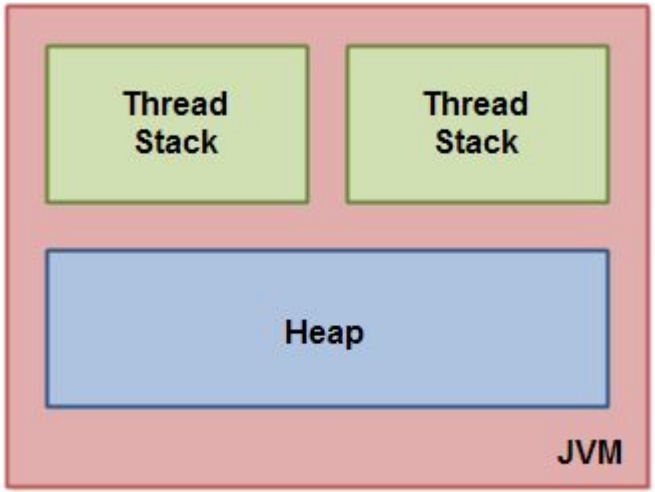






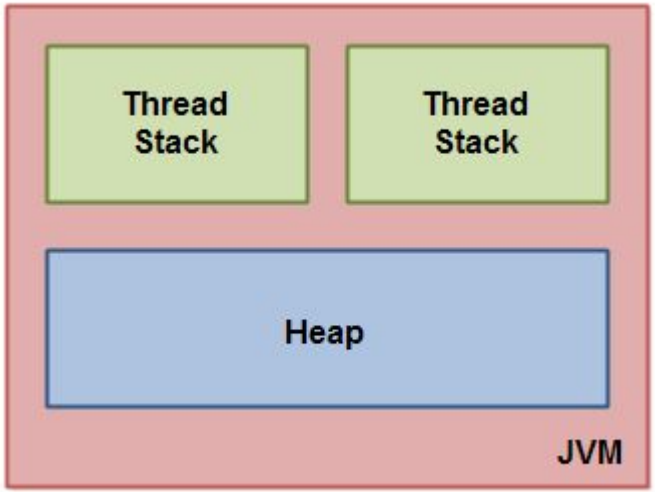
The Internal Java Memory Model(2/n)

- The Java memory model used internally in the JVM divides memory between **thread stacks and the heap**
- Each thread running in the **Java virtual machine has its own thread stack**.
- The thread stack contains information about what **methods the thread has called to reach the current point of execution.**



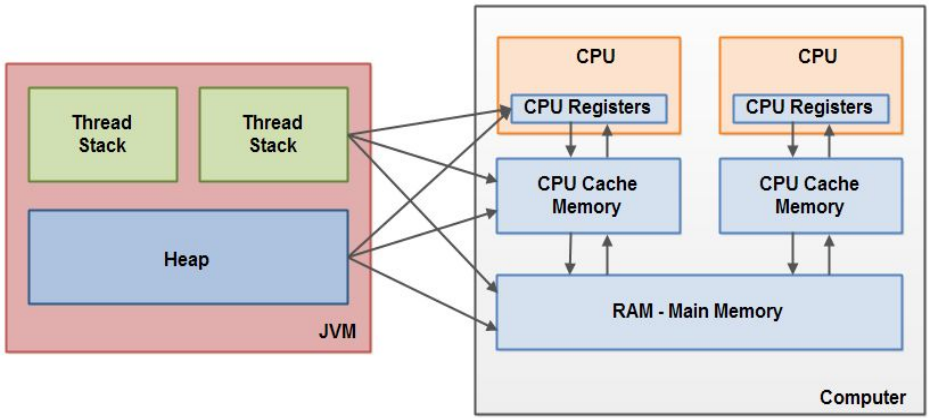
The Internal Java Memory Model(3/n)

- The heap contains all objects created in your Java application, regardless of what thread created the object.
- This includes the object versions of the primitive types (e.g. `Byte`, `Integer`, `Long` etc.).
- It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.



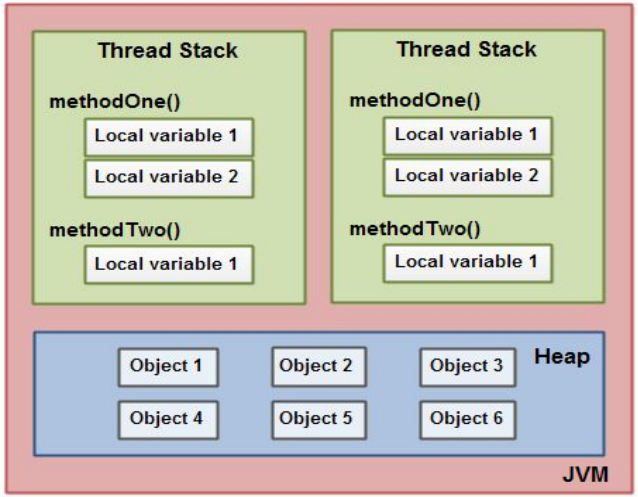
The Internal Java Memory Model(4/n)

- The thread **stack** also contains **all local variables for each method being executed** (all methods on the call stack).
- **A thread can only access it's own thread stack**.
- **Local variables created** by a thread are invisible to all other threads than the thread who created it.\
- Even if **two threads** are executing the exact same code, the **two threads will still create the local** variables of that code in each their own **thread stack.**
- Each thread has its own version of each **local variable**.



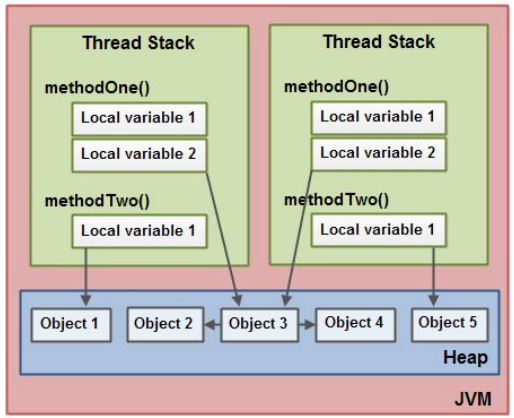
The Internal Java Memory Model(4/n)

- All **local variables of primitive types** (`boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`) are fully stored on the **thread stack** and are thus not visible to other threads.
- One thread may pass a copy of a primitive variable to another thread, but it cannot share the primitive local variable itself.
- The **heap contains all objects created in your Java application**, regardless of what thread created the object.



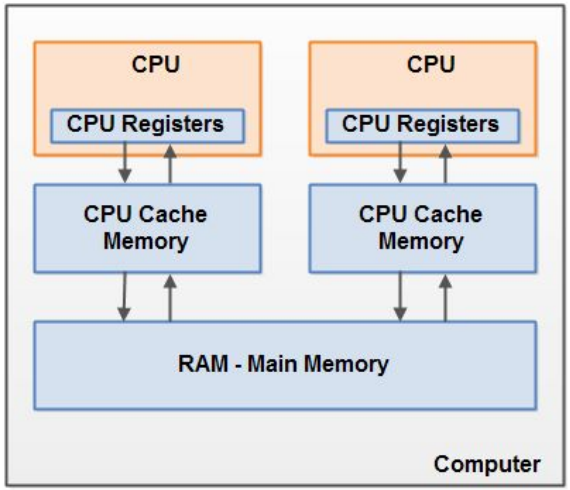
The Internal Java Memory Model(5/n)

- This includes the object versions of the primitive types (e.g. `Byte`, `Integer`, `Long` etc.). It does not matter if an object was created and assigned to **a local variable, or created as a member variable of another object,** the object is still stored on the heap.
- A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on **the thread stack, but the object itself if stored on the heap.**
- **An object may contain methods and these methods may contain local variables.** These local variables are also stored on **the thread stack, even if the object the method belongs to is stored on the heap.**

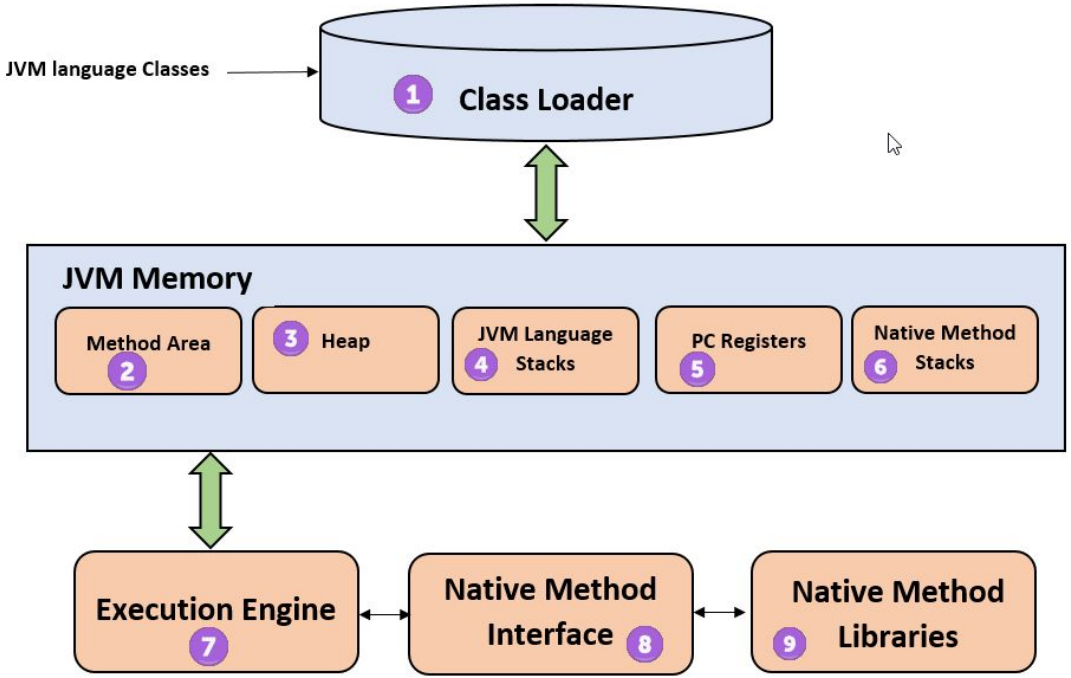


The Internal Java Memory Model(6/n)

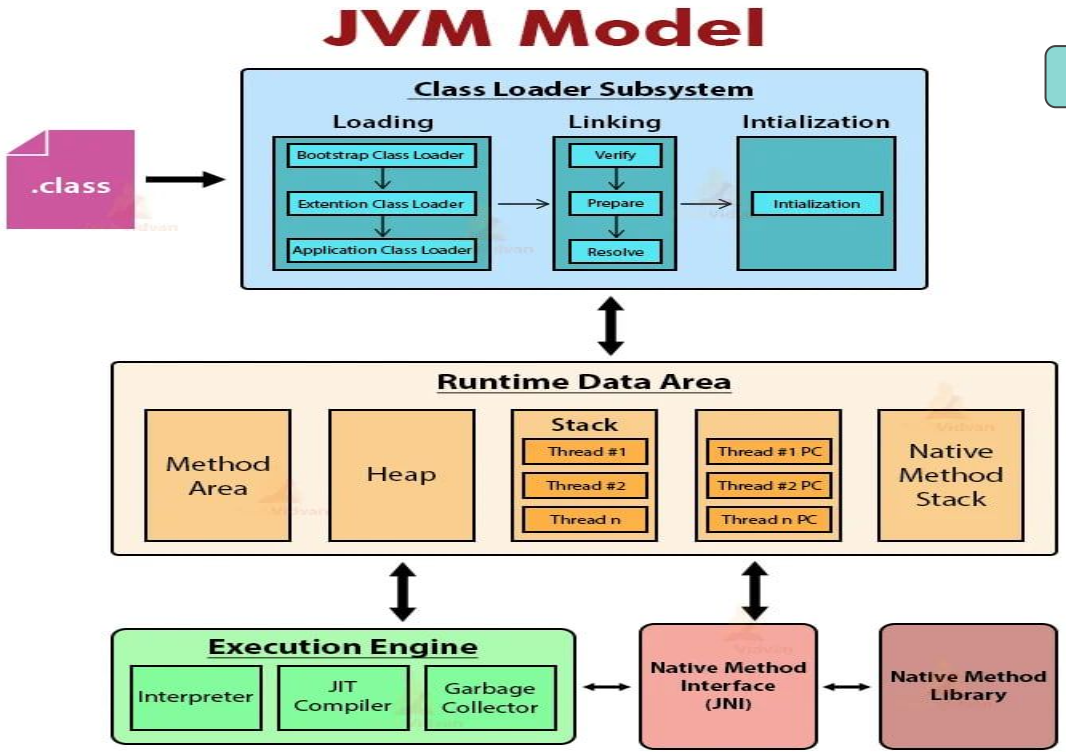
- **Some of these CPUs may have multiple cores**.
- The point is, that on a modern computer with 2 or more CPUs it is possible to have more than one **thread running simultaneously.**
- Each **CPU is capable of running one thread at any given time**. That means that if your Java application is multithreaded, one thread per CPU may be running simultaneously (concurrently) inside your Java application.



The Internal Java Memory Model(7/n)

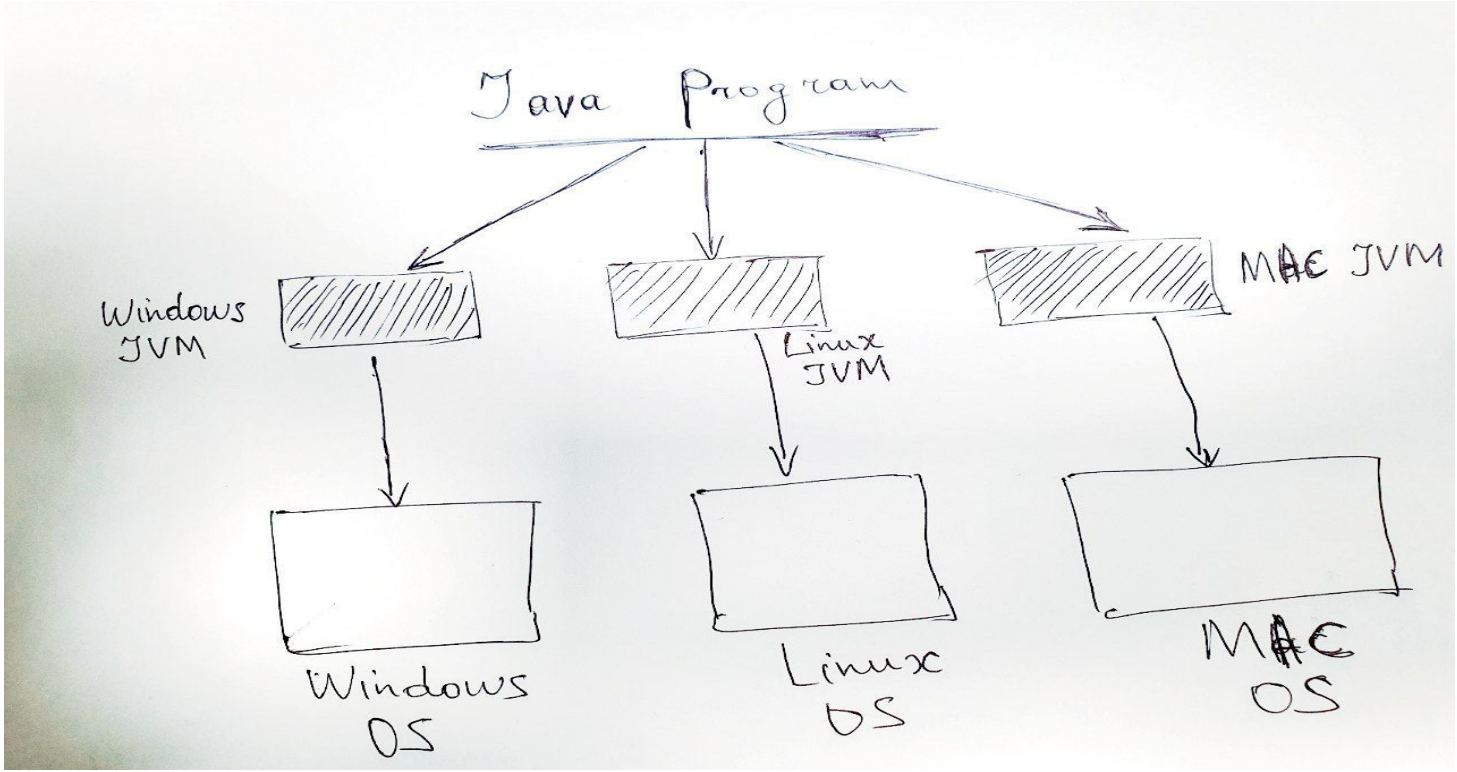


The Internal Java Memory Model(8/n)



1

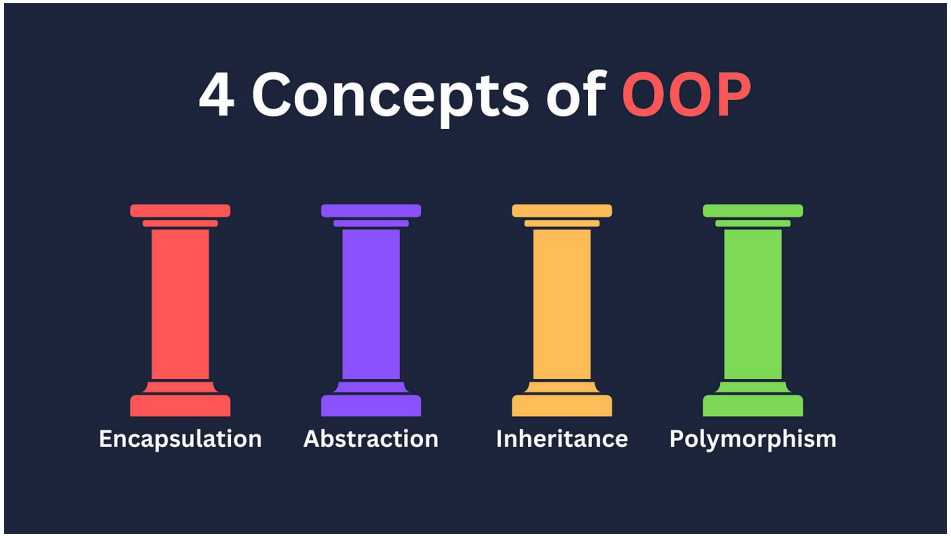
The Internal Java Memory Model(9/n)



What is OOP?

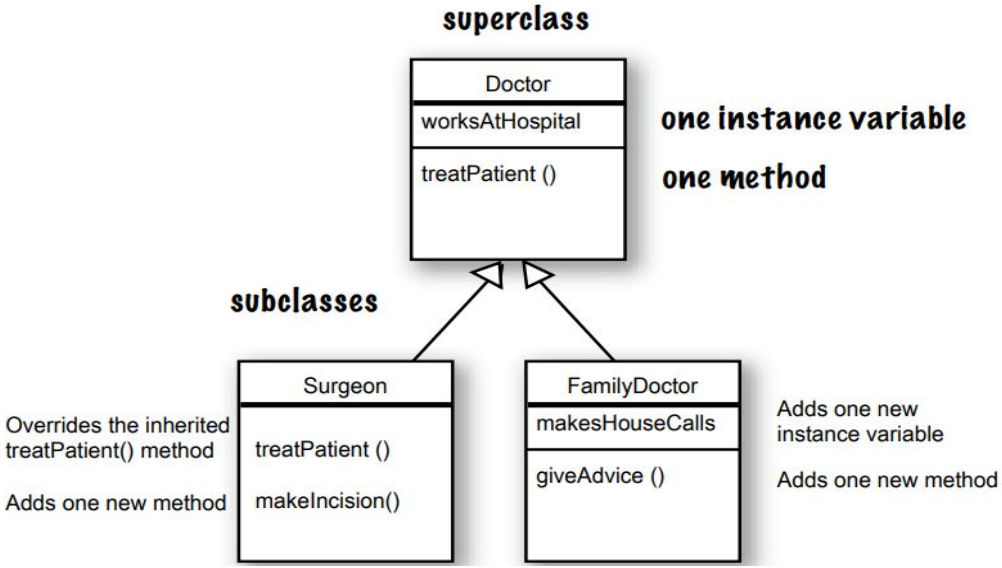
Start today, and we'll throw in an extra level of abstraction!

- **Object-oriented programming (OOP)** is a computer programming model that organizes software design around data, or objects, rather than functions and logic.



What is the power of inheritance ?

- Instance variables are not overridden because they don’t need to be.
- They don’t define any special behavior, so a subclass can give an inherited instance variable any value it chooses



Using IS-A and HAS-A?

- Remember that when **one class inherits from another**, we say that the subclass extends the superclass.
- When you want to know if one thing should extend another, apply the **IS-A test.**
- **If class B extends class A, class B IS-A class A. This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.**

Canine extends Animal

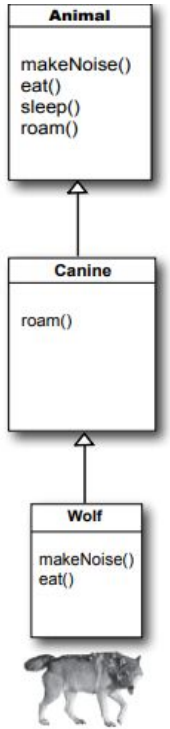
Wolf extends Canine

Wolf extends Animal

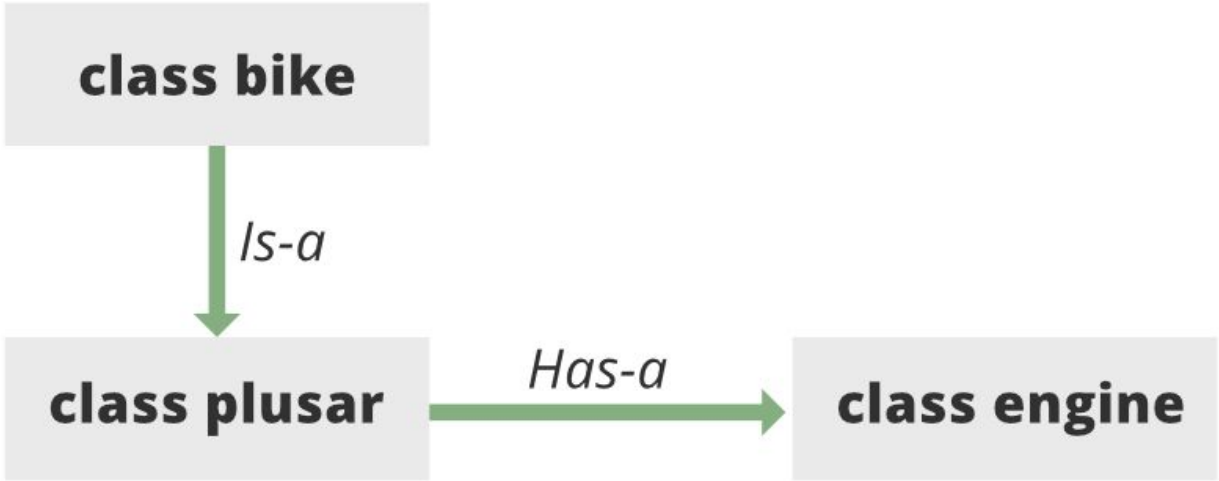
Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



Using IS-A and HAS-A?



Q&A

Q: So we see how a subclass gets to inherit a superclass method, but what if the superclass wants to use the subclass version of the method?

A: A superclass won't necessarily *know* about any of its subclasses. You might write a class and much later someone else comes along and extends it. But even if the superclass creator does know about (and wants to use) a subclass version of a method, there's no sort of *reverse* or *backwards* inheritance. Think about it, children inherit from parents, not the other way around.

Q: In a subclass, what if I want to use BOTH the superclass version and my overriding subclass version of a method? In other words, I don't want to completely *replace* the superclass version, I just want to add more stuff to it.

A: You can do this! And it's an important design feature. Think of the word "extends" as meaning, "I want to *extend* the functionality of the superclass".

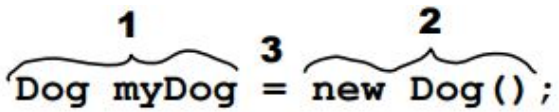
```
public void roam() {
    super.roam();
     // my own roam stuff
}
```

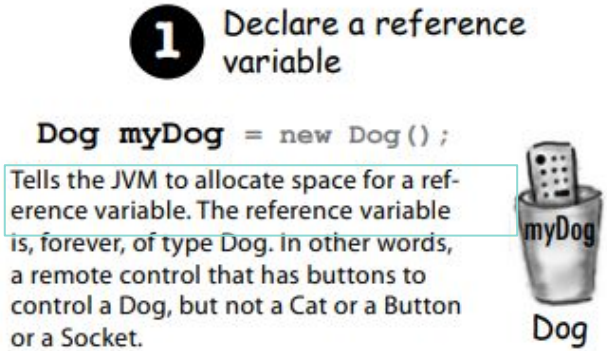
this calls the inherited version of roam(), then comes back to do your own subclass-specific code

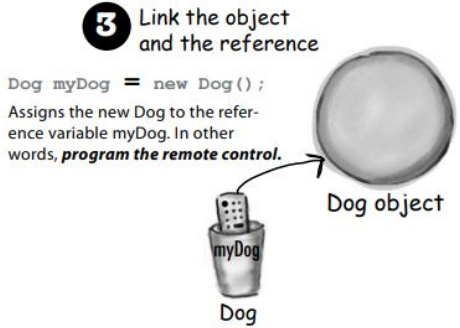
You can design your superclass methods in such a way that they contain method implementations that will work for any subclass, even though the subclasses may still need to 'append' more code. In your subclass overriding method, you can call the superclass version using the keyword **`super`**. It's like saying, "first go run the superclass version, then come back and finish with my own code..."

What is a polymorphism ?

- how polymorphism works, we have to step back and look at the way we normally declare a reference and create an object...
- The 3 steps of object declaration and assignment



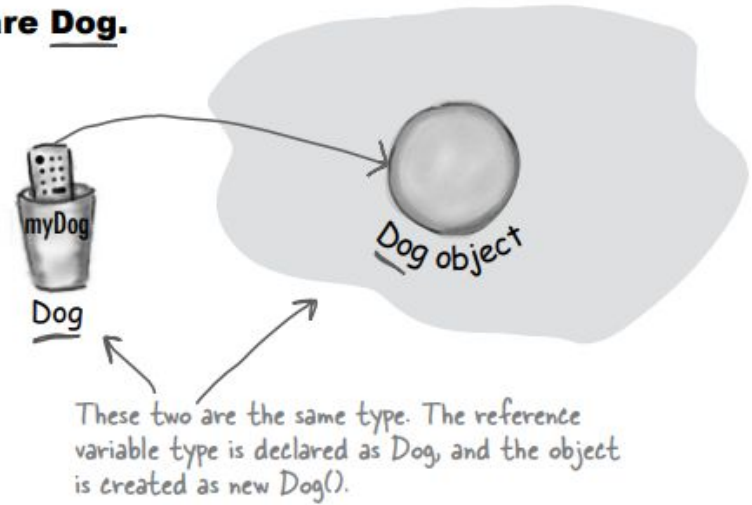




What is a polymorphism ?

The important point is that the reference type AND the object type are the same.

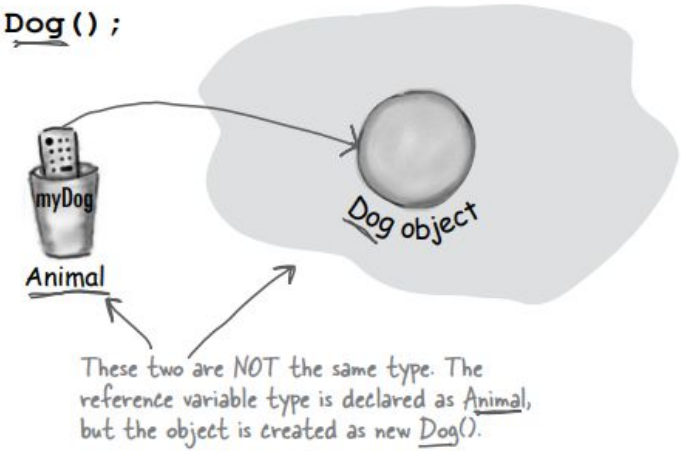
In this example, both are Dog.



What is a polymorphism ?

But with polymorphism, the reference and the object can be *different*.

```
Animal myDog = new Dog();
```



Q&A

Q: Are there any practical limits on the levels of subclassing? How deep can you go?

A: If you look in the Java API, you'll see that most inheritance hierarchies are wide but not deep. Most are no more than one or two levels deep, although there are exceptions (especially in the GUI classes). You'll come to realize that it usually makes more sense to keep your inheritance trees shallow, but there isn't a hard limit (well, not one that you'd ever run into).

Q: Hey, I just thought of something... if you don't have access to the source code for a class, but you want to change the way a method of that class works, could you use subclassing to do that? To extend the "bad" class and override the method with your own better code?

A: Yep. That's one cool feature of OO, and sometimes it saves you from having to rewrite the class from scratch, or track down the programmer who hid the source code.

Q: Can you extend *any* class? Or is it like class members where if the class is private you can't inherit it...

A: There's no such thing as a private class, except in a very special case called an *inner* class, that we haven't looked at yet. But there *are* three things that can prevent a class from being subclassed.

The first is access control. Even though a class *can't* be marked `private`, a class *can* be non-public (what you get if you don't declare the class as `public`). A non-public class can be subclassed only by classes in the same package as the class. Classes in a different package won't be able to subclass (or even *use*, for that matter) the non-public class.

The second thing that stops a class from being subclassed is the keyword modifier `final`. A final class means that it's the end of the inheritance line. Nobody, ever, can extend a final class.

The third issue is that if a class has only `private` constructors (we'll look at constructors in chapter 9), it can't be subclassed.

Q: Why would you ever want to make a final class? What advantage would there be in preventing a class from being subclassed?

A: Typically, you won't make your classes final. But if you need security — the security of knowing that the methods will always work the way that you wrote them (because they can't be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The String class, for example, is final because, well, imagine the havoc if somebody came along and changed the way Strings behave!

Q: Can you make a *method* final, without making the whole *class* final?

A: If you want to protect a specific method from being overridden, mark the *method* with the `final` modifier. Mark the whole *class* as final if you want to guarantee that *none* of the methods in that class will ever be overridden.

Summary

1. Keep in mind that the inheritance IS-A relationship works in only one direction!
2. With polymorphism, the reference type can be a superclass of the actual object type

Reference Resources?

1. **Head First (book)**
2. Java Memory Management
3. Resource Java Memory

Thank you!

Presented by

Hamdamboy Urunov

(hamdamboy.urunov@gmail.com)