Java Polymorphism

Recall the classes Employee and HourlyEmployee that we created earlier. Employee extends Object and HourlyEmployee extends Employee.

new HourlyEmployee("A1", "Malik"): By polymorphism, this object is type HourlyEmployee, but it is also type Employee and type Object.

new Employee("A2", "Marlowe"): By polymorphism, this object is type Employee, but it is also type Object.

The "true type" of an object is what it is created as. The true type never changes (also called the "run time type").

The "current type" of an object is which of its many types is it acting as in a specific Java statement or operation. The current type changes often (also called the "compile time type".)

Polymorphism rules:

- All non-private methods (except constructors) are inherited by subclasses.

- Any inherited method may be overridden in the subclass.

- An object always uses the overridden method of the true type.

- Code may only call method names and parameter signatures accessible to the current type.

- All non-private fields and inner classes can be accessed by subclasses.

- There is no overriding of fields and inner classes, but subclasses can create fields or inner classes of the same name.

- Code may only use a field that is available to the current type.

\* The compiler only has access to the current type - this is why it is also called the "compile time type".

The compiler determines which fields are accessed at each line of the code.

The compiler determines if a method call is legal, but it does not determine which specific method is called.

\* The Java Virtual Machine only has access to the true type - this is why it is also called the "run time type".

The JVM uses the true type to determine which method to call.

\* The only exception concerns typecasts. Typecasts change the current type, and we can typecast an object to any of its valid polymorphism types.

However, because the compiler does not have access to the true type, it is impossible to know if a typecast is legal at compile time.

Thus, the compiler adds a short check at every explicit typecast to verify that the type being requested is valid for the object.

Example: The earnsMoreThan method in Employee:

public boolean earnsMoreThan(Employee e) {

return this.getSalary() > e.getSalary();

}

What is the current type of e? Employee. We can easily see that because it is the type of the variable.

What is the current type of this? Employee. We know that because we are in code in the Employee class.

What is the true type of e? Impossible to know. We do not know which object's address will be stored in the variable nor where that object will be created.

What is the true type of this? Also impossible to know.

Is the getSalary() method call legal Java? Yes. The current type of e and this are both Employee and Employee has a getSalary method that takes no input.

Which getSalary() method is called? That depends on the true type. If the true type of the object is Employee, it will call the one in the Employee class.

If the true type is HourlyEmployee, it will call the overridden one in the HourlyEmployee class.

Why is it bad form to use fields instead?

return this.salary > e.salary;

The field accessed is determined by the current type. In this case, both this and e are current type Employee, and so will use the field declared in the Employee class.

Even if there is a field called salary declared in HourlyEmployee, it will not be used on this line of code.

Java Memory Model, Part 1

Java (and all programming languages) split the memory available for the program into two parts: the heap and the stack.

The stack is organized memory used for method calls. In Java, the default is to set aside about 1MB for the stack (this can be changed when you launch Java).

The heap is unorganized memory. In Java, all classes and objects are stored in the heap.

For every class your program uses, a class "object" is stored in the heap. The class "object" stores:

1) The parent class of the class

2) All static fields declared in the class.

3) All methods (static and non-static) defined in the class.

4) All constructors.

For every object (instance of a class) created by your program (e.g. by the new operator), space for the instance is allocated in the heap.

The instance record stores:

1) The true type of the object.

2) All non-static fields declared in this class or a parent class.

For example, we have a program that uses our Employee and HourlyEmployee classes.

In the heap, Java will place records for the Employee and HourlyEmployee classes. There will also be a record for the Object class since that class is extended by Employee.

Note that Employee contains a list of methods including getName, getSalary, earnsMoreThan, etc.

If Employee has any static fields, they would be stored here.

Employee also contains three constructors.

Note that HourlyEmployee contains a list of methods. Only those methods defined in HourlyEmployee are stored here, not those inherited from Employee: getHours, getHourlyRate, etc.

Note that getSalary and setSalary do appear in HourlyEmployee because these methods are overridden in HourlyEmployee. getName and setName do not appear in HourlyEmployee.

If HourlyEmployee has any static fields, they would be stored here.

HourlyEmployee currently has one constructor.

What is the difference between static and non-static methods? non-static methods have "this". Java stores static and non-static methods together, but Java keeps track

of which is non-static by noting whether it has the "hidden parameter" this.

Creating an object:

Employee e = new Employee("A2", "Marlowe")

Space is allocated for an instance in the heap. The instance contains the non-static fields of Object followed by the non-static fields of Employee: number (initialized by the constructor to the address of the String "A2"),

name (initialized to the address of the String "Marlowe"), salary (initialized to the default 0).

For each variable in the code, the compiler records its "offset" in the class. That is, is it the first, second, third, etc. variable stored in the object.

HourlyEmployee e2 = new HourlyEmployee("A1", "Malik")

Space is allocated for an instance in the heap. The instance contains the non-static fields of Object followed by the non-static fields of Employee: number (initialized by the constructor to the address of the String "A1"),

name (initialized to the address of the String "Malik"), salary (initialized to the default 0); followed by the non-static fields of HourlyEmployee: hours (initialized to the default 0) and hourlyRate (initialized to the default 0.0).

Calling a method:

What if we call a method?

e.getSalary()

Now, Java does the following:

1) Evaluates the left side of the dot to get an address. (Here it is stored in e.)

2) Goes to that address in the heap (it is an object), and goes to that object's true type in the heap. (Here, it will go to the Employee class.)

3) Run's through the class's list of methods to find one that matches. (It will find getSalary in this list.)

4) If the method is not found, if goes to the class's parent class, and repeats step 3.

Another example:

e2.getSalary()

Now, Java does the following:

1) Evaluates the left side of the dot to get an address. (Here it is stored in e2.)

2) Goes to that address in the heap (it is an object), and goes to that object's true type in the heap. (Here, it will go to the HourlyEmployee class.)

3) Run's through the class's list of methods to find one that matches. (It will find getSalary in this list.)

4) Step 4 is not needed.

What if we typecast e2 first? ((Employee)e2).getSalary() ???

- it does not matter! The typecast only affects the current type, and the current type is not used after compiling.

- Java still gets the address of e2, goes to the object, and then to the object's true type to find the method to call.

Another example:

e2.getName()

Now, Java does the following:

1) Evaluates the left side of the dot to get an address. (Here it is stored in e2.)

2) Goes to that address in the heap (it is an object), and goes to that object's true type in the heap. (Here, it will go to the HourlyEmployee class.)

3) Run's through the class's list of methods to find one that matches. (It will -not- find getName in this list.)

4) Since the method getName was not found, it goes to the parent class objet and runs through its list. Here it does find getName.

Accessing an instance field:

this.salary

Java does the following:

1) Evaluates the left side of the dot to get an address.

2) Goes to the object. The compiler stored the offset of the field, and so Java accesses that field in the object.

\* Note that the offset of salary is the same in both an Employee object and an HourlyEmployee object. In both cases it is the second field.

Even if HourlyEmployee has its own salary field, if the current type is Employee, it will use the Employee salary field.