Lecture 11 — Method Overriding

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Housekeeping

- Mid-Semester Test is next week (week 7)
  - 10% formative assessment
  - Thursday April 18th, 17:00
  - Surnames A-M: Wilsmore, Surnames N-Z: Tattersall
  - (Students with alternative arrangements should have been in contact by email and have received an email telling them their alternative venue)
  - See LMS for complete info!
- Monday next week will be a normal lecture
- Thursday lecture will be a revision session
- Labs or office hours are available if you need help before then
- It will cover contents up to and including this week (week 6)
- Assessed lab worksheet will be released on or before 21/04 (week 8)

Contents

- See Chapter 7 of the textbook
- Method overriding

Previously on CITS2005

- In the previous lecture we learned about inheritance
- Today, we will start with method overriding
- This is a big part of using inheritance
- Inheritance lets you extend the functionality of classes
- This achieves code reuse: we can borrow methods and fields from the superclass
- It also achieves abstraction: any code we write for a superclass will work for its subclass
- Method overriding lets us achieve more: polymorphism
- Method overriding happens when methods are hidden by subclasses

SuperGoose

```
class Animal {
    public void talk() {
        System.out.println("*Generic animal sounds*");
    }
}

class Goose extends Animal {
    public void talk() {
        super.talk();
        System.out.println("Honk!");
    }
}

public class SuperGoose {
    public static void main(String[] args) {
        Goose a = new Goose();
        a.talk();
    }
}
```

- `Goose` subclasses `Animal`
- Recall that hiding happens when a subclass redefines a member
- The `talk` method is getting hidden
- `super.talk()` allows `Goose` to call the `Animal` version of `talk`

SuperGoose

```
class Animal {
    // ...
}

class Goose extends Animal {
    // ...
}

public class SuperGoose {
    public static void main(String[] args) {
        // What if we did this??
        Animal a = new Goose();
        a.talk();
    }
}
```

- What if the variable has the superclass type, `Animal`?
- Will it call the `Animal` version of talk because it is an `Animal` variable?
- The object is a `Goose` object, will it call the `Goose` version?

Animal Override

```
class Animal {
    public void talk() {
        System.out.println("*Generic animal sounds*");
    }
}

class Goose extends Animal {
    public void talk() {
        System.out.println("Honk!");
    }
}

class Dog extends Animal {
    public void talk() {
        System.out.println("Woof!");
    }
}

public class AnimalOverride {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.talk();
        a = new Goose();
        a.talk();
        a = new Dog();
        a.talk();
    }
}
```

Animal Override

```
public class AnimalOverride {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.talk();
        a = new Goose();
        a.talk();
        a = new Dog();
        a.talk();
    }
}
```

- The instance's version of the method is always called!
- This is called *dynamic dispatch*
- Java figures out which method to call at runtime, regardless of the type of the variable

Polymorphism

- This is all interesting, but what is the use?
- It supports polymorphism
- We can have a single method, that works in different ways depending on the subclass
- Since they all share a superclass, we can write code that works with any of them!
- The important point is that the *interface* may not change, but the *implementation* can
- Let's see some examples

LegCount

```
class Animal {
    public int numLegs() {
        return 0;
    }
}

class Dog extends Animal {
    // see full code
}

class Seagull extends Animal {
    // see full code
}

class SeagullStandingOnOneLeg extends Seagull {
    // see full code
}

public class LegCount {
    // Works for any Animal subclass!
    public static int countLegs(Animal[] animals) {
        // see full code
    }

    public static void main(String[] args) {
        Animal[] animals = {new Dog(), new Seagull(), new SeagullStandingOnOneLeg()};
        System.out.println("Total number of legs: " + countLegs(animals));
    }
}
```

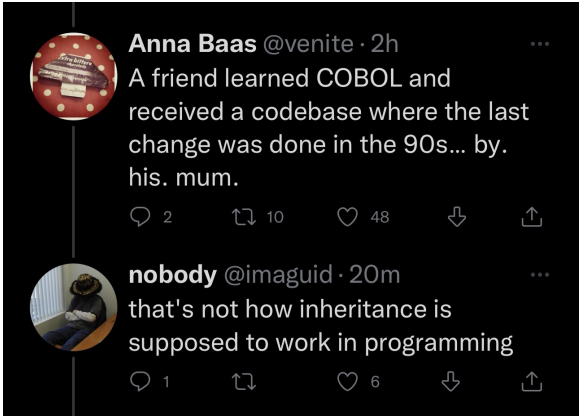
Expression Class

- We're going to make a simple expression class
- This will be a much more ambitious use of polymorphism
- It will represent simple integer arithmetic expressions
- These will contain only integer literals, addition, and multiplication
- In a way, we will be making the foundation for part of a simple programming language

Expression

```
public class Expression {
    public void describe() {
        System.out.println("unknown");
    }

    public int evaluate() {
        return 0;
    }

    public static void main(String[] args) {
        Expression expr = new Multiply(new Value(2), new Value(3));
        expr.describe();
        System.out.println(" = " + expr.evaluate());

        expr = new Add(new Value(2), new Value(3));
        expr.describe();
        System.out.println(" = " + expr.evaluate());

        expr = new Add(new Multiply(new Value(2), new Value(3)), new Value(4));
        expr.describe();
        System.out.println(" = " + expr.evaluate());

        expr = new Value(1);
        for (int i = 2; i <= 10; i++)
            expr = new Add(expr, new Value(i));
        expr.describe();
        System.out.println(" = sum of 1 to 10 = " + expr.evaluate());
    }
}
```

Mid-lecture Break



Expression

```
(2 * 3) = 6
(2 + 3) = 5
((2 * 3) + 4) = 10
(((((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10) = sum of 1 to 10 = 55
```

- `Expression` is a superclass. `Add`, `Multiply`, `Value` are all subclasses
- Lots of tools used to achieve this
- Inheritance: `Add`, `Multiply`, `Value` all need the `describe` and `evaluate` methods
- Recursion: the results of methods are computed recursively
- Method overriding: all of the methods do different things for different classes

Value

```
class Value extends Expression {
    private int value;

    public Value(int value) {
        this.value = value;
    }

    public void describe() {
        System.out.print(value);
    }

    public int evaluate() {
        return value;
    }
}
```

- Represents an integer literal value in our expression
- This is straightforward to describe and evaluate
- Note the method overriding

Add

```
class Add extends Expression {
    private Expression left;
    private Expression right;

    public Add(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public void describe() {
        System.out.print("(");
        left.describe();
        System.out.print(" + ");
        right.describe();
        System.out.print(")");
    }

    public int evaluate() {
        return left.evaluate() + right.evaluate();
    }
}
```

- Represents the addition operator
- Note the use of `Expression`, which can store any subclass
- Note how `.describe()` and `.evaluate()` can be polymorphic and are recursive!

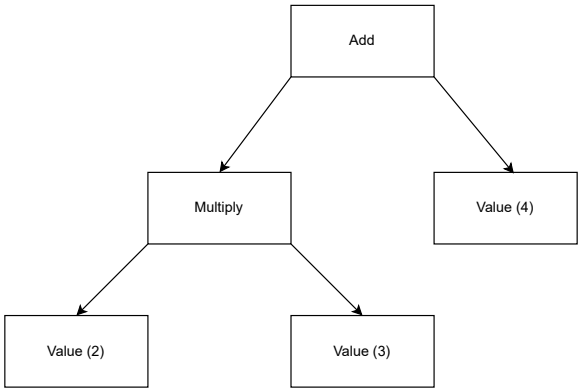
Multiply

```
class Multiply extends Expression {
    private Expression left;
    private Expression right;

    public Multiply(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public void describe() {
        System.out.print("(");
        left.describe();
        System.out.print(" * ");
        right.describe();
        System.out.print(")");
    }

    public int evaluate() {
        return left.evaluate() * right.evaluate();
    }
}
```

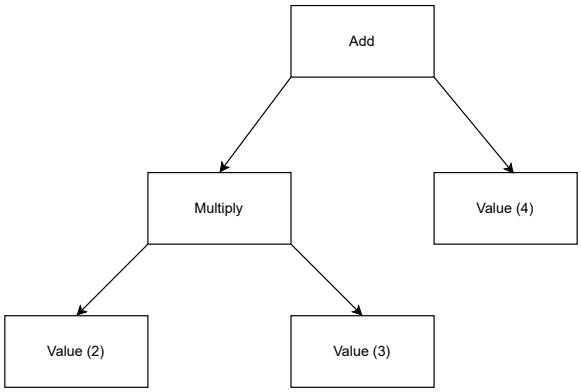
- Essentially the same as `Add`

2*3+4

```
expr = new Add(new Multiply(new Value(2), new Value(3)), new Value(4));
expr.describe();
System.out.println(" = " + expr.evaluate());
```



2*3+4



- Add calls describe/evaluate on Multiply recursively, then on Value recursively
- Multiply calls describe/evaluate on its two Values recursively
- This is possible due to method overriding and polymorphism!

@Override

- Method overriding is a common source of error
- The method must have exactly the same signature (name and parameter list), otherwise it will just be a method *overload*
- If we make a typo, the compiler will not help us!

AnimalTypo

```java
class Animal {
    public void talk() {
        System.out.println("*Generic animal sounds*");
    }
}

class Goose extends Animal {
    public void tallk() { // Oops, typo!
        System.out.println("Honk!");
    }
}

public class AnimalTypo {
    public static void main(String[] args) {
        Animal a = new Goose();
        a.talk();
    }
}
```

- Java has a tool to help us: the `@Override` annotation

@Override

```
class Goose extends Animal {
    @Override
    public void tallk () { // Oops, typo!
        System.out. println ("Honk!");
    }
}
```

- Now, compilation gives us an error
- This is an *annotation*
- These can do lots of things
- In this case, we are giving the compiler extra information telling it that this is supposed to override something
- I recommend using this where appropriate to catch errors

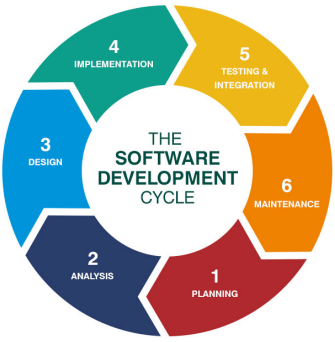
The SDLC

- Time for a philosophical diversion about software engineering in Java!
- This is the Software Development Lifecycle
- It is an idealised view of how software development works
- The cyclic nature reflects how developing software is usually iterative

The SDLC

- Java and classes fit well into the SDLC view of development
- Design involves breaking our problem up into classes
- Implementation involves writing those classes

The SDLC



- Classes give us clear "units" of code to test (see in the next lecture), making testing easier
- Data hiding and abstraction allow us to update the implementation without changing the interface making maintenance easier