Lecture 12 — More inheritance: Object, `abstract`, `final`

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 7 of the textbook
- Previously, we learned about inheritance and method overriding
- We now see some more about inheritance
- `abstract` classes
- The `final` keyword
- The `Object` class

Animal

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

- The `talk()` method from `Animal` is awkward
- It would be good if we could have a `talk` method with no implementation

AbstractAnimal

```java
abstract class Animal {
    public abstract void talk();
}

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

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

- We can use `abstract`
- The `talk()` method is abstract
- Any class with an abstract method must also be declared as abstract

Abstract Classes and Methods

```
abstract class Animal {
    public abstract void talk();
}
```

- Any class can be declared abstract
- An abstract class cannot be instantiated (`new Animal()` leads to an error)
- Any class with an abstract method must be abstract
- Abstract methods need no body
- Abstract methods must be overridden by subclasses (unless the subclass is abstract too)

AbstractError

```java
abstract class Animal {
    public abstract void talk();
}

class Goose extends Animal {
    // Error: Did not override abstract method talk()
}

public class AbstractError {
    public static void main(String[] args) {
        Animal a = new Animal(); // Error: Cannot instantiate the type Animal
        a.talk();
    }
}
```

Expression

```java
public abstract class Expression {
    public abstract void describe();

    public abstract int evaluate();
}
```

- Another good example is the `Expression` class from the previous lecture
- It made sense for `Value`, `Add`, `Multiply` to all implement the methods
- However, it was not well defined for `Expression`

`final` Classes and Methods

- Sometimes we want the opposite of abstract
- Abstract methods must be overidden
- `final` methods can never be overidden. They are “final”
- `final` classes can never be inherited from. They are “final” too
- Note that a class with a final method does not need to be a final class!

final Classes and Methods

```java
abstract class Animal {
    public abstract int numLegs();
}

class Spider extends Animal {
    public final int numLegs() {
        // All spiders have 8 legs
        return 8;
    }
}

class SpiderWith6Legs extends Spider {
    // Error: Cannot override the final method from Spider
    // public int numLegs() {
    //    return 6;
    // }
}
```

final Classes and Methods

```java
final class Insect extends Animal {
    public int numLegs() {
        // All insects have 6 legs
        return 6;
    }
}

// class InsectWith8Legs extends Insect {
//     // Error: Cannot inherit from final Insect
// }
```

- This can be a useful tool to prevent anyone incorrectly inheriting from your class
- In this case, making the method `final` makes the most sense

final Variables

```java
public class FinalVariable {
    public static void main(String[] args) {
        final int x = 5;
        // x = 6; // Error: Cannot assign a value to final variable 'x'
        final double y;
        y = 10.5;
        // y = 1.1; // Error: Cannot assign a value to final variable 'y'
        System.out.println(x);
        System.out.println(y);
    }
}
```

- The `final` keyword can be used for variables and fields
- It has a different meaning. A final variable can only be assigned once
- It's value is “final”; it is a constant

CircleTools

```
public class CircleTools {
    public static final double PI = 3.14159;

    public static double area(double radius) {
        return PI * radius * radius;
    }

    public static double circumference(double radius) {
        return 2 * PI * radius;
    }

    public static void main(String[] args) {
        System.out.println("Area of a circle with radius 5: " + area(5));
        System.out.println("Circumference of a circle with radius 5: " + circumference(5));
    }
}
```

- `final` is often combined with `static`
- It is useful to make the Java compiler ensure nobody can ever redefine π
- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html

The `Object` Class

- Java comes with a built-in class called `Object`
- Every class inherits from `Object` (exception `Object`)
- This means every class in Java is technically in the same inheritance tree!
- `Object` has a methods, which means every class comes with these methods
- Sometimes, you will want to override these methods
- Let's take a look: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html`
- Today we will focus on `toString()` and `equals()`

The `toString()` Method

- The `.toString()` method is used to convert objects to a string representation
- The fact that this is in `Object` means all objects can be represented as a string
- When we do a string concatenation (`"a string"+myObject`), `myObject.toString()` gets called
- The same for `System.out.println(myObject)`
- Overriding `.toString()` gives us control over how our object gets printed

The toString() Method

```java
class MyClass {
    private int x;
    private int y;
    public MyClass(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "MyClass(" + x + ", " + y + ")";
    }
}

public class ToString {
    public static void main(String[] args) {
        MyClass mc = new MyClass(5, 10);
        System.out.println(mc);
        System.out.println("mc.toString() = " + mc);
    }
}
```

The `equals()` Method

- The `.equals()` method is used to check if two objects are equal (e.g., `myObject.equals(myOtherObject)`)
- This is needed because the == operator simply checks if the references are the same
- The default implementation of `.equals()` does exactly the same thing, but since we can override it, we can change this behaviour!
- For example, the `String` class overrides this, which is why you can do `string1.equals(string2)` instead of `string1 == string2`

Custom equals()

```
class StringPair {
    String first, second;
    public StringPair(String first, String second) {
        this.first = first;
        this.second = second;
    }
    @Override
    public boolean equals(Object other) {
        // Make sure that other is a StringPair
        if (other instanceof StringPair) {
            StringPair otherPair = (StringPair) other; // Cast to StringPair
            // Use String .equals() to compare the two Strings
            return first.equals(otherPair.first) && second.equals(otherPair.second);
        }
        return false;
    }
}
```

- Note the use of `instanceof`
- Doing a `instanceof B` checks if an object a is an *instance* of a *subclass* of B. (It can also do other things, but those aren't important here)
- Why is it useful for `Object` to have this method? It means equality will always at least be defined for *any* object

Custom equals()

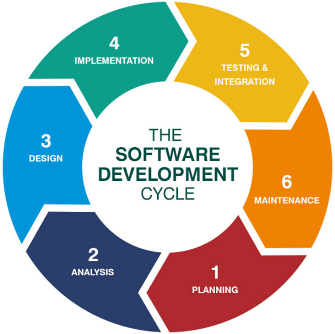
```java
public class Equals {
    // Notice how this will work regardless of which classes are really in the array
    public static boolean allEqual(Object[] objects) {
        if (objects.length == 0) {
            return true;
        }
        Object first = objects[0];
        for (int i = 1; i < objects.length; i++) {
            if (!first.equals(objects[i])) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        StringPair[] pairs = new StringPair[3];
        pairs[0] = new StringPair("Hello", "World");
        pairs[1] = new StringPair("Hello", "World");
        pairs[2] = new StringPair("Hello", "World");
        System.out.println(allEqual(pairs));
    }
}
```

- The `allEqual` method is quite generic. It works for any array of objects regardless of their class

Mid-lecture Break

how many layers of abstraction are you on
like,, maybe 5, or 6 right now my dude
this was supposed to be legs
you are like a little baby
watch this
IBaseEnterpriseVisitorOutputPrintableContextGeneratorSingletonMiddlewareStrategyConfiguratorFactoryReturner.class
EnterpriseAbstractIntegerInputGlobalSanitizerSQLServiceRunnerManagerAgileImpl.class
AdvancedLoopStateManipulationResultRetrievalViaExternalBusinessClientPayloadRunner.class
Java
Java

The SDLC and Testing



- Time for another software engineering interlude
- So far, we have mostly been focused on skills for stages 3, 4, and 6.
- Let's talk a little about stage 5: testing (this will eventually have to do with `abstract` and `final`)

Unit Testing

- Unit testing are automated tests created during software development
- Often, they are the only good way to convince people your code works
- Sometimes, people follow a processes called *test-driven development*
- First you write tests, then you write code that passes those tests
- While the Software Development Lifecycle may seem abstract, **unit testing is very practical**

Unit Testing

- JUnit was one of the early major unit testing frameworks
- It was made for Java, and is very powerful
- We can test each class individually in OOP!
- There are many such frameworks for different languages
- Learning a framework is beyond the scope of this course, although you should be aware of their existence and use
- We will do a basic introduction to unit testing with our own (simple) framework and use `abstract` and `final` along the way

SimpleUnitTest

```java
public abstract class SimpleUnitTest {
    public static final void assertTrue(boolean condition) {
        // See full code for details
        // Error if condition is false
    }

    public abstract void runAllTests();

    public static final void main(String[] args) {
        // See full code for details
        // Look for a class named arg[0]
        // If it is a subclass of SimpleUnitTest, call runAllTests()
    }
}
```

- `assertTrue()` and `main()` are `final` because they need never be altered
- `runAllTests()` is `abstract` because it must be overidden
- Let's test the `Expression` class from earlier

ExpressionTest

```
public class ExpressionTest extends SimpleUnitTest {
    void testAddition () {
        assertTrue (new Add(new Value(1), new Value(1)).evaluate() == 2);
        assertTrue (new Add(new Value(1), new Value(2)).evaluate() == 3);
        assertTrue (new Add(new Value(3), new Value(−3)).evaluate() == 0);
    }

    void testMultiplication () {
        assertTrue (new Multiply(new Value(1), new Value(1)).evaluate () == 1);
        assertTrue (new Multiply(new Value(1), new Value(2)).evaluate () == 2);
        assertTrue (new Multiply(new Value(3), new Value(−3)).evaluate() == −9);
    }

    void testMixed() {
        assertTrue (new Add(new Multiply(new Value(1), new Value(1)), new Multiply(new Value(1), new Value(2))).evaluate () == 3);
        assertTrue (new Multiply(new Add(new Value(1), new Value(2)), new Add(new Value(3), new Value(4))).evaluate() == 21);
    }

    @Override
    public void runAllTests () {
        testAddition ();
         testMultiplication ();
        testMixed();
    }
}
```

ExpressionTest

```
public class ExpressionTest extends SimpleUnitTest {
    void testAddition () {
        // see above
    }

    void testMultiplication () {
        // see above
    }

    void testMixed() {
        // see above
    }

    @Override
    public void runAllTests () {
        // see above
    }
}
```

- Compile the test: `javac ExpressionTest.java`
- This will compile any dependencies, such as `SimpleUnitTest.java`
- Run the test with `java SimpleUnitTest ExpressionTest`