Lecture 10 — Inheritance

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 7 of the textbook
- Today we talk about one of the most interesting parts of OOP in Java: inheritance

Inheritance

- Inheritance is a major feature in Java
- It does two things: code reuse and abstraction
- You can take an existing class, and make a *subclass*
- A subclass inherits all the members (fields and methods, not constructors) of its *superclass*
- This is code reuse
- A subclass can then add new features on top of these
- Java sees a subclass as doing all the same things its superclass does, and more
- This is abstraction
- We will start by modifying a simplified `BasicSafeArray` similar to the `SafeArray` from the previous lecture

BasicSafeArray

```java
public class BasicSafeArray {
    protected int size;
    protected int[] array;

    public BasicSafeArray(int size) {
        // see full code
    }

    public int size() {
        // see full code
    }

    protected boolean isValidIndex(int index) {
        // see full code
    }

    public int get(int index) {
        // see full code
    }

    public void set(int index, int value) {
        // see full code
    }
}
```

- Note that we introduce a new `size()` getter method for convenience
- Also, `size` and `array` are not `private`. More on `protected` later

Extending `BasicSafeArray`

- Suppose we want to *extend* the `BasicSafeArray` to create a new class with the same functionality plus a new method
- We can do this using inheritance
- In Java, we use the `extends` keyword to create a subclass
- Let's look at an example where reintroduce the `append(...)` method

AppendableSafeArray

```
public class AppendableSafeArray extends BasicSafeArray {
    AppendableSafeArray(int size) {
        super(size);
    }

    public void append(int value) {
        int newSize = size + 1;
        int[] newArray = new int[newSize];
        for (int i = 0; i < size; i++)
            newArray[i] = array[i];
        newArray[size] = value;
        size = newSize;
        array = newArray;
    }

    public static void main(String[] args) {
        // see full code
    }
}
```

- The `extends` keyword essentially copies in all the fields and methods that aren't `private`
- We add a new append method
- To keep things interesting, this a mutable version of append (unlike the previous lecture)
- `super(size)` is how we call the constructor for the superclass, `BasicSafeArray`. We need to do this because constructors are not inherited

`protected`

- We have seen `private` and `public`
- `private` members can only be accessed by that class
- `public` can be accessed by any class
- This means `private` members cannot be accessed even by subclasses!
- This can be an issue if you are trying to extend to behaviour of the class, such as in our array example
- The alternative is `protected`. This is similar to `private`, but can be accessed by subclasses

`protected`

- There are some exceptions to `protected`
- We will learn about these in more detail when we cover *packages*
- The basic idea is that subclasses **and** classes in the same package can see `protected` members
- By the way, there is a fourth access modifier in addition to `private`, `protected`, and `public`
- This is the *default* access modifier
- We will talk about this too when we cover *packages*
- Let's look at another example

Thing

```java
public class Thing {
    public int x;
    protected float y;
    private String z;
}
```

ExtendedThing

```
public class ExtendedThing extends Thing {
    public void print() {
        System.out.println(x);
        System.out.println(y);
        // System.out.println(z); // Error: z is private
    }

    public static void main(String[] args) {
        ExtendedThing t = new ExtendedThing();
        t.print();
    }
}
```

Aside about Field Initialisation

- Did you notice that the fields (`x,y`) had value zero even though they weren't initialised?
- Fields work similarly to arrays. They are initialised before object construction to some default values
- All numeric types (e.g., `int, long, char, double, float, ...`) are initialised to zero
- Booleans are set to `false`
- Objects and arrays are set to `null`

Superclass Constructors

```
public class AppendableSafeArray extends BasicSafeArray {
    AppendableSafeArray(int size) {
        super(size);
    }
    ...
}
```

- The super keyword allows us to reference the super class
- Here, we use it to call the constructor of BasicSafeArray
- Sometimes, we want to write our own constructor without explicitly calling super
- For example, let's try to re-write the constructor without reference to super

Superclass Constructors

```
public class AppendableSafeArray extends BasicSafeArray {
    AppendableSafeArray(int size) {
        this.size = size;
        this.array = new int[size];
    }
    ...
}
```

- This leads to a mysterious error
- `AppendableSafeArray.java:2: error: constructor BasicSafeArray in class BasicSafeArray cannot be applied to given types;` ... etc
- This is because the superclass constructor is always actually called
- By default, the empty constructor (`BasicSafeArray()`) is called

Superclass Constructors

```java
public BasicSafeArray() {
    this(0);
}
```

- To fix this, we can add an empty constructor to `BasicSafeArray`
- Note the use of `this` to call a different constructor
- Aside: when we use `this` or `super` to call a different constructor, it needs to be the first statement of our constructor

```java
    public BasicSafeArray() {
        // int x = 10; // error
        this(0);
        boolean y = false; // ok
    }
```

More Features of `super`

- `super` can be used to access more than the superclass's constructor
- It can be used to access members
- This can be useful when a subclass hides fields or methods
- If we re declare a field or method in a subclass, it hides the member that was inherited
- In general, I recommend not doing this for fields. It usually indicates bad class design
- Let's see an example extending `Thing` from earlier in the lecture

Hiding Members

```
public class HiddenThing extends Thing {
    public int x; // Hides super.x
    protected float y; // Hides super.y

    public void setSuperTo10() {
        super.x = 10; super.y = 10;
    }

    public void setTo100() {
        x = 100; y = 100;
    }

    public void print() {
        // see full code
    }

    public void printSuper() {
        // see full code
    }

    public static void main(String[] args) {
        HiddenThing t = new HiddenThing();
        t.setTo100();
        t.setSuperTo10();
        t.print();
        t.printSuper();
    }
}
```

Inheritance Chains

- Java allows us to build chains of inheritance
- We could have a class B that extends A, and a class C that extends B
- This works how you might expect
- Class B gets all the non-private members of A, and class C gets all the non-private members of A and B
- Two tricky questions
- How do superclass constructors work?
- How does `super.member` work?

Superclass Constructors

```
class A {
    public A() {
        System.out. println ("A()");
    }
}

class B extends A {
    public B() {
        System.out. println ("B()");
    }
}

class C extends B {
    public C() {
        System.out. println ("C()");
    }
}

public class SuperclassConstructor {
    public static void main(String [] args) {
        C c = new C();
    }
}
```

- Superclass constructors are called from the “highest” class downwards
- By default, Java does the same thing as inserting `super()` at the start of any constructor

What does super refer to?

```
class A {
    public void doSomething() {
        System.out.println("A.doSomething()");
    }
}

class B extends A {
    public void doSomething() {
        System.out.println("B.doSomething()");
    }
}

class C extends B {
    public void doSomething() {
        super.doSomething();
    }
}

public class SuperclassMembers {
    public static void main(String[] args) {
        C c = new C();
        c.doSomething();
    }
}
```

- `super` always refers to the immediate superclass

Multiple Subclasses

- It is possible for a single class to be extended by multiple classes
- Not in a chain, but in parallel
- For example, class B could extend class A, and class C could also extend class A
- Let's consider an example (slightly silly) class `DeletableSafeArray`
- This will extend `BasicSafeArray`
- Recall that `AppendableSafeArray` already extends `BasicSafeArray`!

DeletableSafeArray

```java
public class BasicSafeArray {
    // see full code
}

public class AppendableSafeArray extends BasicSafeArray {
    public void append(int value) {
        // see full code
    }
}

public class DeletableSafeArray extends BasicSafeArray {
    public DeletableSafeArray(int size) {
        super(size);
    }

    public void delete(int index) {
        // todo
    }
}
```

- We can create a arbitrarily complex tree of inheritance, not just a chain!

DeletableSafeArray

```java
public void delete(int index) {
    if (isValidIndex(index)) {
        for (int i = index; i < size - 1; i++)
            array[i] = array[i + 1];
        size--;
    } else
        System.out.println("Invalid index: " + index);
}
```

- The new delete method removes an element at a specific index
- `delete(2)` with an `array` $= \{1, -2, 5, 6\}$ should result in $\{1, -2, 6\}$
- We implement this by copying elements from right to left from the deletion index
- Note the use of the inherited method `isValidIndex`

Phylogenetic Tree

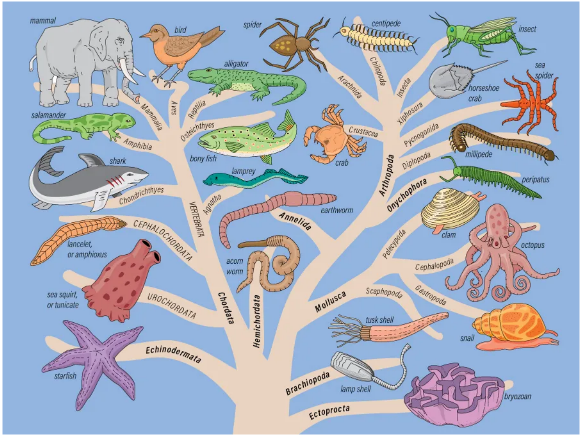

Image from `https://www.britannica.com/story/how-do-you-read-phylogenetic-trees`

- The tree of life is an example of a class hierarchy
- We are using the same principle to organise our classes

Superclass References

- Java is strongly typed. Mismatched types lead to errors
- `int x = 55.2` leads to a type error
- There are a few exceptions: automatic type conversions and casting
- We now see a powerful third exception
- Consider two different classes X and Y
- `X myObj = new Y()` would cause an error
- However, if Y extends X, then it works!
- Why is this?

Superclass References

- When we say `Y extends X` we're say that `Y` *is a* `X`
- It inherits all the same members, so it can do everything an `X` can, and (usually) more!
- This *is a* relationship is really what `extends` captures
- Java is therefore happy to store a `Y` in any variable that is expecting an `X`
- This is an example of *abstraction*
- We will be seeing a lot more of this in the next lecture, but for now, let's end with an example of the power of this

SetEverythingTo

```
public class SetEverythingTo {
    public static void setTo(BasicSafeArray array, int value) {
        for (int i = 0; i < array.size(); i++)
            array.set(i, value);
    }

    public static void printArray(BasicSafeArray array) {
        for (int i = 0; i < array.size(); i++)
            System.out.println(array.get(i));
        System.out.println();
    }

    public static void main(String[] args) {
        BasicSafeArray array = new BasicSafeArray(5);
        AppendableSafeArray append_array = new AppendableSafeArray(5);
        DeletableSafeArray delete_array = new DeletableSafeArray(5);
        setTo(array, 1);
        setTo(append_array, 2);
        setTo(delete_array, 3);
        append_array.append(11);
        delete_array.delete(3);
        printArray(array);
        printArray(append_array);
        printArray(delete_array);
    }
}
```