Lecture 9 — More Methods

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 6 of the textbook
- A deeper dive into methods and their keywords

Previously on CITS2005

- We have seen many methods already
- We have seen the difference between `private` and `public` (more on that today)
- We have seen constructors; special methods that are used when new objects are created
- Today we refresh all of these and dive into some new concepts such as recursion and the `static` keyword

public and private

- Classes contain fields, constructors, and methods
- Each of these can be declared as `public` or `private`
- `public` means any other class can access them
- `private` means that only this class can access them
- This is how Java supports two important software engineering principles: *data hiding* and *encapsulation*
- `private` is how we do *data hiding*
- A class bundles together code and data into easy to use objects. Data hiding insulates the user from complexity. They only need to know about the `public` parts. This is *encapsulation*
- Let's see an example where we make a `SafeArray` class that is like an array but does not throw exceptions

SafeArray

```
public class SafeArray {
    private int size;
    private int[] array;

    public SafeArray(int size) {
        // todo
    }

    private boolean isValidIndex(int index) {
        // todo
    }

    public int get(int index) {
        // todo
    }

    public void set(int index, int value) {
        // todo
    }
}
```

- Here is the class outline
- Notice what is `public` and what is `private`
- Let's complete the constructor/methods

SafeArray

```
public SafeArray(int size) {
    this.size = size;
    this.array = new int[size];
}
```

- The constructor
- Sets up the fields
- Note that `array = new int[size]` would also work
- As would `array = new int[this.size]`

SafeArray

```
private boolean isValidIndex(int index) {
    return index >= 0 && index < size;
}
```

- Checks if a specified index is in the array
- Notice that it is `private`. This is not for use outside the class!
- Users should only `get()` and `set()`

SafeArray

```
public int get(int index) {
    if (isValidIndex(index))
        return array[index];
    System.out.println("Invalid index: " + index);
    return 0;
}
```

- Gets the value at an index
- Does not throw an exception if the index is not valid
- Uses the `private isValidIndex()` method

SafeArray

```
public void set(int index, int value) {
    if (isValidIndex(index))
        array[index] = value;
    else
        System.out.println("Invalid index: " + index);
}
```

- Sets the value at an index if the index is valid
- Notice that this is a `void` method

SafeArray

- The `SafeArray` class uses encapsulation
- The `array` and methods that act on it are bundled into a class
- We can protect access to the `array` with `private`: data hiding
- `public` methods are used to provide limited access to the user and handle errors

Pass-by-reference

- In the previous lecture, we saw that methods and constructors can have parameters that are classes
- This means the values that are passed in are objects
- To understand how this works, we need to know about *pass-by-value* and *pass-by-reference*
- We have already seen that variables that store objects (or arrays) actually store references
- Variables that store primitive types store the value directly
- This is the same for parameters. Primitive values get copied (passed by value), objects get referenced (passed by reference)
- This makes sense. Primitive types are often efficient to copy (e.g., an `int`), but objects are usually much larger

copyInto

```
public void copyInto(SafeArray other) {
    if (other.size != size) {
        System.out.println("Arrays are not the same size");
        return;
    }
    for (int i = 0; i < size; i++) {
        // Notice how we can access other.array even though it is private
        other.array[i] = array[i];
    }
}
```

- A new method for `SafeArray` that copies the contents into another `SafeArray`
- Since `other` is a reference, changes to it will persist after the method call
- Note that we can access `other.array` because a `SafeArray` knows about `private` members of any `SafeArray`

copyInto

```
public static void main(String[] args) {
    SafeArray arr1 = new SafeArray(3);
    SafeArray arr2 = new SafeArray(3);
    arr1.set(0, 1);
    arr1.set(1, 2);
    arr1.set(2, 3);
    arr2.set(0, 10);
    arr2.set(1, 20);
    arr2.set(2, 30);
    System.out.println(arr2.get(0) + " " + arr2.get(1) + " " + arr2.get(2));
    arr1.copyInto(arr2);
    System.out.println(arr2.get(0) + " " + arr2.get(1) + " " + arr2.get(2));
}
```

- An example program using two SafeArrays

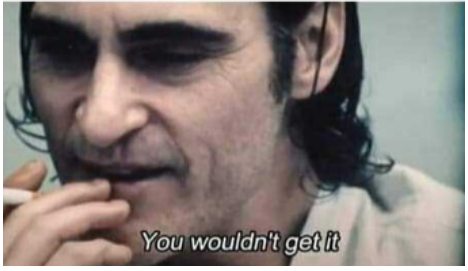
return by reference

```
public SafeArray append(int value) {
    SafeArray newArray = new SafeArray(size + 1);
    for (int i = 0; i < size; i++)
        newArray.array[i] = array[i];
    newArray.array[size] = value;
    return newArray;
}
```

- A new method that appends a value to the end of a `SafeArray`
- Works by creating a new `SafeArray` and returning it
- Notice that objects are returned by reference similarly to how they are passed by reference

Mid-lecture Break

```
1  public class Meme
2  {
3      private Joke joke;
4
5      public void setJoke(Joke newJoke)
6      {
7          this.joke = newJoke;
8      }
9  }
```



Overloading

- Multiple methods can have the same name in Java
- This is called *overloading*
- As long as they have different parameters, Java is able to determine which one you are trying to call
- Formally, they need to have a different *signature*; e.g., `myMethod(int, float)`, `myMethod(int)`, `myMethod(SafeArray, SafeArray)`, `myMethod(SafeArray)`
- A good example is `abs()` (absolute value)
- It needs to work for all numeric types (`abs(int)`, `abs(double)`, `abs(float)`, `abs(long)`, etc)
- This is convenient, since we do not need to give different names to methods that do the same thing on different types

Overloading append()

```
public void append(SafeArray other) {
    int newSize = size + other.size;
    int[] newArray = new int[newSize];
    for (int i = 0; i < size; i++)
        newArray[i] = array[i];
    for (int i = 0; i < other.size; i++)
        newArray[size + i] = other.array[i];
    array = newArray;
    size = newSize;
}
```

- This method appends an entire array in-place
- Notice how the return type is different. Overloading requires that the type signature be different. The return type can be the same, or different. It doesn't matter!

Overloading Constructors

```
public SafeArray(SafeArray other) {
    this.size = other.size;
    this.array = new int[size];
    for (int i = 0; i < size; i++)
        array[i] = other.array[i];
}
```

- Constructors can be overloaded the same way as methods
- A common reason for this is to have a “copy constructor”

Recursion

- Methods in Java can call other methods
- A Java program is essentially a single method call: `main`
- The `main` method will usually construct some objects and call other methods
- Methods can call themselves. This is called *recursion*
- Let's look at an example: computing the Fibonacci numbers: $1, 1, 2, 3, 5, 8, 13$
- $f(1) = 1$, $f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$

Fibonacci

```
public class Fibonacci {
    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        for (int i = 1; i <= 10; i++)
            System.out.println(f.fib(i));
    }

    public int fib(int n) {
        if (n <= 2)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }
}
```

The `static` keyword

- We have seen the `static` keyword a lot
- `public static void main(String[]] args)`
- What does it mean?
- Static methods belong to a class rather than an object
- If we have `ClassName variable`, we usually call a method like this `variable.myMethod(...)`
- The method executes on that object
- Static methods belong to the class rather than an object
- `ClassName.myStaticMethod(...)`

StaticFib

```
public class StaticFib {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++)
            System.out.println(StaticFib.fib(i));
    }

    public static int fib(int n) {
        if (n <= 2)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }
}
```

- Note that we could just do `System.out.println(fib(i));` here
- If we do not specify `this` or a class name, Java will look for a method in the current class

Static Methods

- These methods make sense when they do not need to use fields
- For example, the Java API's `Math` class contains many static methods
- `https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`
- For example, `abs`, `sqrt`, `max`, `min`, `sin`, `cos`, `...`

Static Fields

```
public class StaticX {
    public static String x;
}
```

- Fields can also be static
- They can be accessed the same way (`StaticX.x`)
- There is only one per class
- In a sense, all instances share the same field

Static Fields

```
public class StaticXTest {
    public static void main(String[] args) {
        StaticX instance = new StaticX();
        StaticX instance2 = new StaticX();
        instance.x = "Hello";
        instance2.x = "Goodbye";
        StaticX.x = "World";
        System.out.println(StaticX.x + " " + instance.x + " " + instance2.x);
    }
}
```