Lecture 6 — More Loops, then Classes

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapters 3 and 4 of the textbook
- More about loops
- Classes, fields, methods, and constructors

Looping back

- Last lecture, we looked at loops
- `while`, `do-while`, and `for` loops
- Now, we take a closer and look
- Loops can be nested
- Let's see this in a program that computes if numbers are prime

Nested Loops

```
import java.util.Scanner;

public class NestedLoops {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a positive number: ");
        for (;;) {
            int num = sc.nextInt();
            if (num < 1)
                break;
            boolean isPrime = true;
            for (int d = 2; d < num; ++d) {
                if (num % d == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
                System.out.println(num + " is prime");
            else
                System.out.println(num + " is not prime");
        }
    }
}
```

- There is an error in this code. What is it?

Nested Loops

```
import java.util.Scanner;

public class NestedLoops {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a positive number: ");
        for (;;) {
            int num = sc.nextInt();
            if (num < 1)
                break;
            boolean isPrime = true;
            for (int d = 2; d < num; ++d) {
                if (num % d == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
                System.out.println(num + " is prime");
            else
                System.out.println(num + " is not prime");
        }
    }
}
```

- It says 1 is a prime number! We need to deal with this special case

continue

```java
import java.util.Scanner;

public class NestedLoops2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a positive number: ");
        for (;;) {
            int num = sc.nextInt();
            if (num < 1)
                break;
            // Note the use of continue
            if (num == 1) {
                System.out.println("1 is not prime");
                continue;
            }
            boolean isPrime = true;
            for (int d = 2; d < num; ++d) {
                if (num % d == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
                System.out.println(num + " is prime");
            else
                System.out.println(num + " is not prime");
        }
    }
}
```

continue

```
public class Continue {
    public static void main(String[] args) {
        for (int i = 1; i <= 20; i++) {
            if (i % 2 == 0)
                continue;
            System.out.println(i);
        }
    }
}
```

- `continue` ends execution early and goes to the next iteration
- This is different to `break` that ends the entire loop
- Note that the condition is still evaluated, and `for` loops still execute their loop end statement

`class` and OOP

- The moment you have been waiting for?
- Object oriented programming in Java is achieved using *classes*
- We have already been using very simple classes
- These contain only one *method*, the `main` method
- Classes can contain many methods and need not have a `main` method
- Classes are a primary unit of *abstraction* in Java
- They group together data (fields) and code (methods)

class

```
class BankAccount {
    String ownerName;
    int balance;
}
```

- A simple `class` that models a bank account
- This class contains only 2 *fields*. These are like variables, but for an object
- Classes are different to objects
- A class is a specification. No BankAccounts actually exists yet
- To actually create a BankAccount, and store the fields in memory, we must create an *instance*

Creating Instances

```
public class BankExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.ownerName = "Donald Knuth";
        account.balance = 1000;
        account.balance -= 15;
        System.out.println(account.ownerName + " has $" + account.balance);
    }
}
```

- Here create an instance of the `BankAccount` class. Our first proper object!
- This is done using the `new` keyword
- Notice how `BankAccount` is a new *type* in Java (like `String` or `Scanner`)
- `account` is a variable that holds `BankAccount` objects
- Note the `object.member` syntax to access the fields

Creating Instances

```
public class BankExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.ownerName = "Donald Knuth";
        account.balance = 1000;
        account.balance -= 15;
        System.out.println(account.ownerName + " has $" + account.balance);
    }
}
```

- You may be wondering why `BankExample` knows about `BankAccount`
- It is because they are in the same folder
- If they were not, we would need to use `import` to tell Java where to find it
- We will learn more about this when we learn about `packages`
- For now, we will always put classes that refer to one another in the same folder

Creating Instances

```
public class BankExample2 {
    public static void main(String[] args) {
        BankAccount account1, account2;
        account1 = new BankAccount();
        account2 = new BankAccount();
        account1.ownerName = "Donald Knuth";
        account1.balance = 1000;
        account1.balance -= 15;
        account2.ownerName = "Alan Turing";
        account2.balance = 2000;
        account2.balance -= 77;
        account1.balance += account2.balance;
        System.out.println(account1.ownerName + " has $" + account1.balance);
        System.out.println(account2.ownerName + " has $" + account2.balance);
    }
}
```

- Their fields are separate, despite coming from the same class

Stored by Reference

```
public class BankExample3 {
    public static void main(String[] args) {
        BankAccount account1, account2;
        account1 = new BankAccount();
        account2 = account1; // Same object
        account1.ownerName = "Donald Knuth";
        account1.balance = 1000;
        account1.balance -= 15;
        System.out.println(account1.ownerName + " has $" + account1.balance);
        System.out.println(account2.ownerName + " has $" + account2.balance);
    }
}
```

- Objects are stored by *reference*
- `account1` and `account2` store a reference to the object, not the object itself
- Think of them as a handle, or the name of the right pigeonhole

Methods

```
class BankAccount2 {
    String ownerName;
    int balance;

    void depositMoney(int amount) {
        balance += amount;
    }
}
```

- A new version of `BankAccount` with a *method*
- A method is a way of grouping executable code together
- `main` is a special method where Java starts execution
- This method allows us to deposit money into an account

Methods

```
return-type methodName(parameter-list) {
    body
}
```

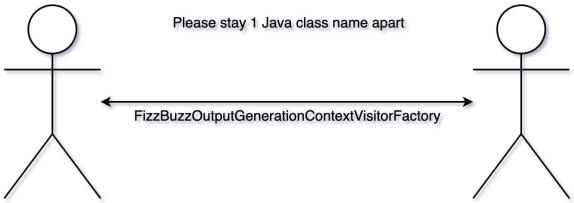
- A method has a *header* and a *body*
- When a method is called, the body is executed with the parameters defined as variables
- We will explain the return type in detail soon
- For now, `void` means the method does not return anything

Methods

```
public class BankExample4 {
    public static void main(String[] args) {
        BankAccount2 account = new BankAccount2();
        account.ownerName = "Donald Knuth";
        account.balance = 1000;
        account.depositMoney(500);
        System.out.println(account.ownerName + " has $" + account.balance);
    }
}
```

- We *call* a method using the dot notation and providing *arguments*
- Each *parameter* must be given an *argument*
- `object.methodName(arg1, arg2, ...)`
- Methods can have no parameters: `myObject.myMethod()`

Mid-lecture break



return statement

```
class BankAccount3 {
    String ownerName;
    int balance;

    void depositMoney(int amount) {
        if (amount < 0) {
            System.out.println("You can't deposit a negative amount!");
            return;
        }
        balance += amount;
    }
}
```

- The `return` statement allows us to end the execution of a method
- Analogous to `break`

return statement

```
class BankAccount4 {
    String ownerName;
    int balance;

    void depositMoney(int amount) {
        if (amount < 0) {
            System.out.println("You can't deposit a negative amount!");
            return;
        }
        balance += amount;
    }

    int withdrawMoney(int amount) {
        if (amount > balance) {
            amount = balance;
        }
        balance -= amount;
        return amount;
    }
}
```

- `return` is used to return a value for non-`void` methods
- The return value must be of the correct type (e.g., `int`)
- `return` *`expression`*`;`

return statement

```
public class BankExample5 {
    public static void main(String[] args) {
        BankAccount4 account = new BankAccount4();
        account.ownerName = "Donald Knuth";
        account.balance = 1000;
        account.depositMoney(500);
        int withdrawn = account.withdrawMoney(2000);
        System.out.println(account.ownerName + " has $" + account.balance);
        System.out.println("Withdrawn $" + withdrawn);
    }
}
```

- `return` statements allow methods to send a value back to the caller
- `return` and arguments are how methods communicate
- The returned value often indicates success or failure, or it can be some computed value (e.g., `sqrt()`)
- All non-`void` methods must return a value

Constructors

- When we do `new BankAccount()`, it is calling a *constructor*
- A *constructor* is a special method that sets up a new instance
- Java provides a default constructor
- It gives all numeric types a value of 0, boolean a value of `false`, and all objects `null`
- `null` is a special value for *reference* (object) variables that means they do not have any reference yet
- Notice how we had to set `account.ownerName = "Donald Knuth"`
- `account.ownerName` would be `null` by default
- `account.balance` would be 0 by default

Constructors

```
class BankAccount5 {
    String ownerName;
    int balance;
    BankAccount5() {
        ownerName = null;
        balance = 0;
    }
}
```

- We can replace the default constructor by adding our own constructor
- This example does the same thing as the default constructor
- Notice how, unlike a method, there is no return type (not even `void`)

Constructors

```
class BankAccount5 {
    String ownerName;
    int balance;
    BankAccount5() {
        ownerName = "No name";
        balance = 1000;
    }
}
```

- We could do something different to the default constructor if we wanted

Parameterised Constructors

```
class BankAccount6 {
    String ownerName;
    int balance;
    BankAccount6(String ownerName, int balance) {
        this.ownerName = ownerName;
        this.balance = balance;
    }
}
```

- Constructors can have parameters
- Notice the `this` keyword to distinguish between parameter and field
- `this.name` always means a field, `name` will refer to the most specific variable

Multiple Constructors

```
class BankAccount6 {
    String ownerName;
    int balance;

    BankAccount6(String ownerName, int balance) {
        this.ownerName = ownerName;
        this.balance = balance;
    }

    BankAccount6(String ownerName) {
        this.ownerName = ownerName;
        this.balance = 0;
    }
}
```

- It is possible to have multiple constructors. They must have different parameter lists

Multiple Constructors

```
public class BankExample6 {
    public static void main(String[] args) {
        BankAccount6 account1 = new BankAccount6("Donald Knuth", 1000);
        BankAccount6 account2 = new BankAccount6("Alan Turing");
        System.out.println(account1.ownerName + " has $" + account1.balance);
        System.out.println(account2.ownerName + " has $" + account2.balance);
    }
}
```

- Our new constructors can be called via `new`
- Note that the old default constructor is gone
- `new BankAccount6()` would be invalid