CITS 1200 Java Programming

A Discipline of Programming
(with apologies to E.W. Dijkstra)

An Introduction to Programming by Contract and
UWA Java Tools for BlueJ
Complexity

The Boeing-777 commercial aircraft is a mix of proven equipment, many new technologies and some new features. Altogether the digital aircraft contains over $5 \times 10^6$ lines of code. Report AGARD-AR-343, Advisory Group for Aerospace R&D 1996
CITS1200 projects are approx

- 250 lines of code and require around 25 person hours of effort
- So the B-777 code is the size of 20,000 CITS1200 projects taking 57 person years
- Writing real code requires discipline and standards to ensure that your code interfaces correctly with the whole system
Lecture Outline

• Programming in the large
• Java syntax and the Java compiler
• Public interface of a class
• Method signatures
• Private fields and methods
• How to read Javadoc documentation
• How to read a JUnit4 test cases
• Checkstyle: see lab sheets and
UWA Java Tools

- The standard release of BlueJ runs only the outdated JUnit3 system
- Patrick Doran-Wu UWA has developed a JUnit4 version in consultation with the BlueJ team at Kent.
- We will be using his BlueJ 99.0.1junit4 in this unit
- BlueJ with JUnit4 is already installed in the CSSE labs

- You can download it (for home) from


- This is a new release, so please email rachel with any bugs discovered
Java Compiler

Programs must conform to Java syntax in order to be executable.
Some of the most common errors are forgetting a semi-colon or a bracket.

Use Edit/Auto-layout in BlueJ to make it easier to spot these errors.
Missing semi-colon syntax error

```java
/**
 * Class BankAccount - a simple model of a bank account.
 * designed to illustrate the concepts of classes, objects, and methods
 *
 * @author Rachel Cardell-Oliver based on version by Gordon Royle
 * @version Feb 2011
 */

class BankAccount {
    private int balance
    private String accountName;
    private int valueDeposits;
    private int valueWithdrawals;
    private int maximumBalance;
    private int minimumBalance;
}
```

'.' expected
Missing bracket syntax error
Public Interface of a Class

```java
public class BankAccount {
    private int balance;
    private String accountName;
    private int valueDeposits;
    private int valueWithdrawals;
    private int maximumBalance;
    private int minimumBalance;

    /**
     * construct a bank account with given initial balance
     * @param accountName unique identifying string
     * @param balance opening balance of account, assumed non-negative
     */
    public BankAccount(String accountName, int balance) {
        this.accountName = accountName;
        this.balance = balance;
        this.valueDeposits = balance;
        this.valueWithdrawals = 0;
        this.maximumBalance = balance;
        this.minimumBalance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public String getAccountName() {
    }
}
```
Javadoc view of the class

Class BankAccount

```java
java.lang.Object
    BankAccount
```

```java
public class BankAccount extends java.lang.Object

Class BankAccount - a simple model of a bank account designed to illustrate the concepts of classes, objects, and methods
```

Version:
Feb 2011
Author:
Rachel Cardell-Oliver based on version by Gordon Royle

**Constructor Summary**

BankAccount(java.lang.String accountName, int balance)

construct a bank account with given initial balance

**Method Summary**

```java
void applyInterest(double rate)
    calculate and update interest under the condition that account balance is negative and rate is positive
```

```java
void deposit(int amount)
    deposit amount by adding to balance
```
Method Signatures

/**
 * construct a bank account with given initial balance
 * @param accountName unique identifying string
 * @param balance opening balance of account, assumed to be non-negative
 */

public BankAccount(String accountName, int balance)

Your implementation of all public methods must match the specification exactly – that is the contract that allows Java programs to be built from collections of classes
More Method Signatures

/**
 * deposit amount by adding to balance
 * @param amount must be non-negative
 */
public void deposit (int amount) ...

/**
 * withdraw amount by decreasing balance
 * @param amount must be non-negative
 */
public void withdraw (int amount) ...
Private Fields and Methods

- Java classes operate a “need to know” policy
- They offer a public signature of services that other classes can use.
- Everything else is hidden to avoid unintended alterations.
- For example, if a bank account balance field is public, then any other class can alter the balance.
- Fields are always hidden by using the `private` key word
- Methods that are only required within a class, and not offered as a service to other classes, are also hidden
- These private methods are called helpers
- Exercise: look for all the private declarations in sample code.
How do you know what to implement?

- A method signature tells you the name, return type and parameters.
- Javadoc gives a prose description of the intended purpose.
- But how do you know exactly what the method should do?
- For example, if the amount parameter of the deposit method is -100, what should the resulting balance be? Isn’t that a withdrawal rather than a deposit?

- Answer: JUnit test cases specify the intended behaviours of a method by giving examples of the expected result for each of the different cases of the method.
How to read JUnit test cases

```java
/**
 * deposit method increases balance by given amount
 */
@Test public void testBalanceAfterDeposit() {
    normalAccount.deposit(200);
    assertEquals(1200, normalAccount.getBalance());
}

/**
 * deposit has no effect if amount is negative
 */
@Test public void testBalanceAfterNegDeposit() {
    normalAccount.deposit(-200);
    assertEquals(1000, normalAccount.getBalance());
}
```
When your code fails a test
then read the test code to see what is missing
What’s in a test case?

Unit test cases have the following form:

1. describe the method’s inputs
2. describe the expected outputs
3. call the method and observe actual output
4. compare expected with actual and report:
   ```java
   assertEquals( comment, expected, actual);
   ```
Before every test case

```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

/**
 * The test class BankAccountBasicTest.
 * test for basic functionality of the BankAccount class
 *
 * @author Rachel Cardell-Oliver
 * @version February 2011
 */
public class BankAccountBasicTest {
    private BankAccount normalAccount, emptyAccount;

    /**
     * set up sample accounts to use for the tests
     */
    @Before public void setUp() {
        normalAccount = new BankAccount("John", 1000);
        emptyAccount = new BankAccount("Jane", 0);
    }
}
How a JUnit test suite is executed

1. `setUp` method is executed
2. the first test method in the class is executed
3. `setUp` method is executed
4. the second test method in the class is executed

Continue until all test cases are executed.

**IMPORTANT** each test case is *independent* of previous tests
How Test Cases are Chosen

• Typical cases
  – Sanity check that method code works as expected, but try to choose tricky cases anyway!

• Special cases and Boundary cases
  – Most errors occur for boundary values, eg empty or full array, -1,0,1 etc.

• Exceptional cases
  – Illegal input, divide by 0, un-initialized parameters
Heads up: method signature errors

Solution: check the deposit method in your class, it probably has the wrong signature e.g. makeDeposit or wrong parameter list.
A goal of any software developer should be to write consistently clear, high quality, maintainable code. This is not always easy and requires a certain amount of discipline at the best of times. One way to help achieve high quality code is by using coding standards.


- CS-naming
- CS-blocks
- CS-coding
- CS-hiding
- CS-complexity
- CS-javadoc
Summary

In CITS1200 you will be programming by contract in all your lab exercises and projects.

In order to do this, you need to take account of information provided by all of the following:

- Java compiler
- Class and method signatures
- Javadoc descriptions of methods
- JUnit test cases for methods
- Checkstyle coding standards
  - (the minimal UWA Java Tools ones)
Summary

In CITS1200 you will be programming by contract in all your lab exercises and projects.

In order to do this, you need to take account of information provided by *all* of the following:

- Java compiler
- Class and method signatures
- Javadoc descriptions of methods
- JUnit test cases for methods
- Checkstyle coding standards
  - (the minimal UWA Java Tools ones)
Lab Sheets and Submissions

http://undergraduate.csse.uwa.edu.au/units/CITS1200/Laboratories/

The lab sheets are posted on this web page in the week before the week in which the lab is to be done. Labs must be submitted by the due date shown on that page and marks will be available from csmarks. Following submission there will be a 1-week window for resubmission to allow you to improve your code or correct errors. *Trial for 2011*