# Testing, Debugging & Development

CSCU9A3
Data Structures, Objects and Algorithms

David Cairns

## Debugging

- Debugging is a fact of programming life.
- Arises for many reasons (typing mistakes, conceptual errors).
- It is important to work on your debugging skills and use the tools that are available to help you. The less time you have to spend debugging, the less frustrated you will be.
- What can we do to make it easier?
  - Logging
  - Assertions
  - Run time debugger

## To debug use Logging

- When debugging we can insert print statements into the program and print out values of variables.
  - This informs the programmer what is happening inside the program as it runs.
  - However, you then need to delete or comment out all the print statements which can be tedious.
- Logging is a better way
  - The *java.util.logging* package provides the logging capabilities via the Logger class.
  - It is common practice to use the fully qualified name of each class whose activity is being logged as a message category because this allows developers to fine-tune log settings for each class.



## Levels of Logging

You can set the level of logging information that you wish to receive. We will just use 3 levels to start.

- INFO
- WARNING
- SEVERE

We can display

- INFO, WARNING, SEVERE
- WARNING, SEVERE
- SEVERE

# Logging: An Example

```
import java.util.logging.Logger;
import java.util.logging.Level;

public class LogExample
{

    private Logger   logger;
    private int      id;

    public static void main(String[] args)
    {
        LogExample logTest1 = new LogExample(1);
        LogExample logTest2 = new LogExample(2);

        logTest1.go();
        logTest2.go();
    }
```



# Logging: An Example

```
    // A constructor for LogExample
    public LogExample(int i)
    {
        // Take a note of our id
        id = i;

        // Set the logger up with the id for this object
        // Each object could have a unique id or name
        // so that you know which object produced the log output.
        logger = Logger.getLogger("Logging." + id);
    }
```

# Logging: An Example

```
// A method that does something useful and logs the output
public void doImportantThings()
{
     System.out.println("Log level: " + logger.getLevel());
     logger.log(Level.INFO,"An info entry");
     logger.log(Level.WARNING,"A warning log entry");
     logger.log(Level.SEVERE,"A severe log entry");
     System.out.println("");
}

// This method is the same as the above but uses
// shortcut methods to log information at different levels.
public void doMoreImportantThings()
{
     System.out.println("Log level: " + logger.getLevel());
     logger.info("An info entry");
     logger.warning("A warning log entry");
     logger.severe("A severe log entry");
     System.out.println("");
}
```



# Logging: An Example

```
// The real start point for our code...
public void go()
{
     System.out.println("logger name:" + logger.getName());

     logger.setLevel(Level.INFO);
     doMoreImportantThings();

     logger.setLevel(Level.WARNING);
     doImportantThings();

     logger.setLevel(Level.SEVERE);
     doImportantThings();

     logger.setLevel(Level.OFF);
     doImportantThings();
}

}
```

## Debugging: Summary

- Try not to use printing to debug code – printing is for printing.
- Create a logger for each class similar to the example
- When you are debugging turn the logger level to "ALL"
- When you have finished debugging turn the logger level to "OFF"



## Assertions / Unit Testing

- An *assertion* is a statement in Java that enables you to test your assumptions about your program (data).
  - For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.
  - Assertion tests are used to confirm that your code is operating as expected and that values are within the correct limits
- As you add more code to your class or use the class with other classes, you rerun the assertion tests to check that everything still works as expected.

# Assert

- Each assertion contains a Boolean expression that you believe will be true when the assertion executes.
  - the assertion confirms your assumptions about the behaviour of your program
- Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs
  - The assertion test code is in a separate class from your main code and exists as a set of tests. You repeatedly run the tests to check that all is well.
  - Integrated Development Environments have direct support for assertion tests and will report the number of tests that failed and where they failed.
- If you work as a team, you will use these tests on a daily basis to confirm that code that is developed by different teams will properly integrate.



# Assertion Example : Car Class

```
public class Car {

     // Some attributes of a Car
     private String make;
     private String model;
     private double maxSpeed;
     private double currentSpeed;

     // A constructor for Car
     public Car(String mk, String mod, double max)
     {
          make = mk;
          model = mod;
          maxSpeed = max;
          currentSpeed = 0;
     }
```

## Assertion Example : Car Class

```
    public void accelerate(double v)
    {
        currentSpeed = currentSpeed + v;
    }

    public void stop()
    {
        currentSpeed = 0;
    }

    public double getCurrentSpeed()
    {
        return currentSpeed;
    }

    public double getMaxSpeed()
    {
        return maxSpeed;
    }
}
```



## Assertion Example : Test Class

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Tests {

    // A constructor
    public Tests()
    {
        // Put general initialisation code here
    }
```

# Assertion Example : Test Class

```
@Test
public void CarTest1()
{
     Car enzo = new Car("Ferrari","Red",221);

     enzo.accelerate(50);

     assertTrue("Current speed mismatch",enzo.getCurrentSpeed() == 50);
     assertTrue("Current speed > max speed ",
                         enzo.getCurrentSpeed() < enzo.getMaxSpeed());
     assertTrue("Current speed is negative? ",enzo.getCurrentSpeed() >= 0);

     enzo.stop();
     assertTrue("Car not stopped ",enzo.getCurrentSpeed() == 0);
}
```



# Assertion Example : Test Class

```
@Test
public void CarTest2()
{
     Car enzo = new Car("Ferrari","Red",221);

     enzo.accelerate(50);

     assertTrue("Current speed mismatch",enzo.getCurrentSpeed() == 50);
     assertTrue("Current speed > max speed ",
                         enzo.getCurrentSpeed() < enzo.getMaxSpeed());

     enzo.accelerate(-100);
     assertTrue("Current speed is negative? ",enzo.getCurrentSpeed() >= 0);
}
}
```

## Unit Tests

- Verify that a class works correctly in isolation, outside a complete program
  - Enables you to test parts of your code long before all the pieces are available or complete
- To test a class, use an environment for interactive testing, or write a tester class
  - Typically carries out the following steps:
    - Construct one or more objects of the class that is being tested
    - Invoke one or more methods
    - Print out one or more results
    - Compare the expected result with the actual result



## Testing & The Development Process

- You should design from the top down but implement and test from the bottom up
  - When implementing code, add the smallest part you can at a time and then check it works as expected
    - Unit tests allow you to automatically run these checks
  - They will also catch cases where new code breaks older code
    - The previously successful assertion will fail
  - If you repeatedly test, you know the error is always something to do with the last part you added.
  - Do not type all your code in at once and hope for the best
    - it may work on very small problems but it becomes a complete headache for any significant problem

## Do **not** catch Assertion Exceptions

- You should not encounter any assertion errors through normal execution of a properly written program.
  - Such errors should only indicate bugs in the implementation.
  - As a result, you should never catch an AssertionError.



## Benefit of using Assertion in Java

- Assertions are great for **data validation**.
- Assertion in Java guarantees that at a certain point your **assumption** is correct, this makes debugging in Java a lot easier.
- Using Java assertions helps to detect bugs early in the development cycle which is very **cost effective**.
- Writing code with assertions will help you to be a better programmer and **improve the quality of your code**.
- Assertions in Java gives you a lot of **confidence while maintaining or refactoring code**.
- It is absolutely crucial to working as part of a team on a larger development project with millions of lines of code. Tests are repeatedly re-run (often over night) to check that the latest additions to code have not broken anything.

## Exceptions

- Exceptions are the customary way in Java (and many other languages) to indicate to a calling method that an abnormal condition has occurred
- You must position *catchers* (the exception handlers) strategically, so your program will catch and handle all exceptions from which you want your program to recover.
  - If your code is responsible for flying an aircraft, would you rather it had an exception handler that caught an unexpected condition or just gave up?



## Assert or Exception

- Assertions should be used to check something that should never happen, while an exception should be used to check something that might happen.
  - Throwing an Exception is a way to deal with external problems (wrong sort of data coming in).
  - Assertions enable you to check the state of data yourself
  - You can enable and disable assertions

## Using Exceptions

- When checking parameters passed to public or protected methods and constructors
- When interacting with the user or when you expect the client code to recover from an exceptional situation
- Very common in file input/output when you are not in complete control of the situation (e.g. a networked file server may disconnect).



## Using Assertions…

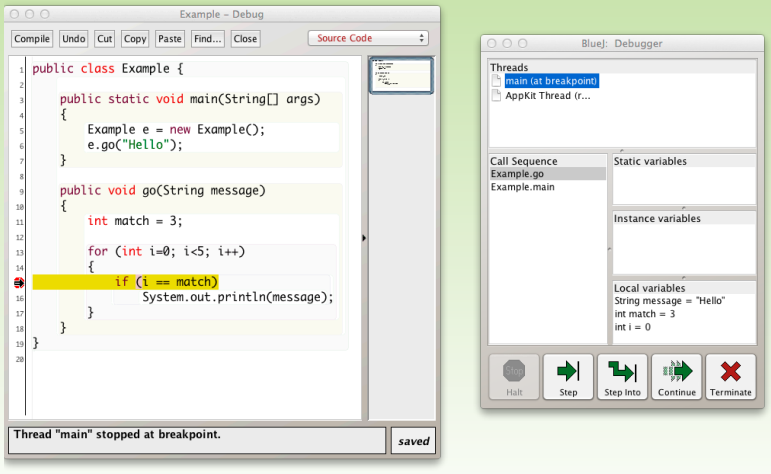
- Primarily used for pre-release testing
  - Checking pre-conditions, post-conditions and invariants (values that should not change) of private/internal code
  - Provide feedback to yourself or your developer team
  - Checking for things that are very unlikely to happen otherwise it means that there is a serious flaw in your application
  - Stating things that you (supposedly) know to be true
- Always remember Assertions do not replace Exceptions but compliment them.

# Run Time Debugger

- Professional Integrated Development Environments (IDEs) contain Run Time Debuggers
  - They allow you to run a program, step through code and watch the values of variables change as each line executes
  - Very powerful method of finding out where code is going wrong
    - A lot better than filling your code with println statements…
  - Very good learning tool if you are not sure what code is actually doing
  - They can look a bit intimidating but are well worth the effort of learning to understand and use



# Run Time Debugger - BlueJ



# Run Time Debugger - Eclipse