INFO1113 Object-Oriented Programming

Week 11A: JDB, Annotations

Slides developed by Tyson Thomas

Final Exam

- Date: 1st of December 2020
- Time: 9:00 AM Sydney time
- Duration: 130 Minutes
  - Reading time: 10 Minutes
  - Writing time: 120 Minutes
- New Canvas site
  - Final Exam for: INFO1113
  - Access no later than 7 days before the exam
- Everyone starts the exam at the same time
  - Only one attempt allowed
  - No late submission
- Exam adjustment is done by the exam office
  - Notification no later than 3 days before the exam

Question Type:

- MCQs
  - Determine the correct output
  - True/False
  - Fill in the blanks
  - Single/Multiple choice
- Essay Type
  - Identify errors
  - Explain functionality
  - Write code
- Practice exam available in Canvas

Any Questions?

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Topics

- Debugging at runtime (s. 4)
- Annotations (s. 26)
  - Getting more from your compiler

JDB

Logic errors are much harder to detect than syntax errors. Within Java, the compiler can display when we have incorrectly written a statement, but it cannot outline if there is an error is our logic, we just see incorrect output.

So, how do we break down what is going on?

JDB

The java debugger provides functionality

- Inspect the current values of local variables
- Stop execution and step through the code
- Inspect object and class instances

These simple tools allow you to inspect the state of your program of your choosing and understand where an error has occurred.

JDB

We can set up **breakpoints** that allows us to pause execution and inspect the current state of the program.

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to replicate.

JDB

We can set up **breakpoints** that allows us to paus
inspect the current state of the program.

We can specify a breakpoint using the commands **stop at** or **stop in.**

Example: stop at Program:53

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to replicate.

JDB

We can set up **breakpoints** that allows us to paus inspect the current state of the program.

We can specify a breakpoint using the commands **stop at** or **stop in.**

Example: stop at Program:53

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to replicate.

Once it is paused, we can call **locals** and inspect all local variables to the method

JDB

We can set up **breakpoints** that allows us to paus inspect the current state of the program.

We can specify a breakpoint using the commands **stop at** or **stop in.**

Example: stop at Program:53

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a specific value.

Overriding values allows us to check scenarios which may not be easy to rep

Similar to locals but for extracting information from an **object**. For methods that manipulate properties of an object we can inspect all properties.

Once it is paused, we can call **locals** and inspect all local variables to the method

JDB

We can set up **breakpoints** that allows us to paus

inspect the current state of the program.

We can specify a breakpoint using the commands **stop at** or **stop in.**

Example: stop at Program:53

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a

Similar to **dump** command but for specific variable, typically calling toString().

Overriding values allows us to check scenarios which may not be easy to rep

Similar to locals but for extracting information from an **object**. For methods that manipulate properties of an object we can inspect all properties.

Once it is paused, we can call **locals** and inspect all local variables to the method

JDB

We can set up **breakpoints** that allows us to paus

inspect the current state of the program.

We can specify a breakpoint using the commands **stop at** or **stop in.**

Example: stop at Program:53

When the program is paused, we can inspect **local** variables, **dump** object information, simply **print** an object value or **set** a variable to a

Similar to **dump** command but for specific variable, typically calling toString().

For any heavy control flow within the program, we can set the variable's value.

Overriding values allows us to check scenarios which may not be easy to rep

Similar to locals but for extracting information from an **object**. For methods that manipulate properties of an object we can inspect all properties.

Once it is paused, we can call **locals** and inspect all local variables to the method

JDB

Typical logic errors:

- Off by one error
- Miscalculation of a field
- Infinite loops
- Usage of the wrong variable
- Using an incorrect key or index
- Incorrect assumption

Although compilers are becoming more complex and including rules to prevent certain class of logic errors, we still have to be vigilant to ensure the program’s logic is correct.

How can we debug our programs?

JDB

So we can accurately see what bytecode symbols map to our source code we will need to ensure your program is compiled with debugging symbols.

```
> javac -g MyProgram.java
```

You can use jdb with a program compiled without debugging symbols but without this information we cannot easily inspect variables.

JDB

Once compiled, we can start a JDB session and inspect our program.

```
> jdb MyProgram
```

This starts a session, waiting for the user to set up necessary checks and run the program.

JDB

Once ready, we can set up breakpoints and then

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32                op.execute("My String!");

main[1]
```

JDB

Once ready, we can set up breakpoints and then

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32                op.execute("My String!");

main[1]
```

Setting a breakpoint on this line

JDB

Once ready, we can set up breakpoints and then

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32                op.execute("My String!");

main[1]
```

Setting a breakpoint on this line

Running the program

JDB

Once ready, we can set up breakpoints and then

```
> jdb MyProgram
Initializing jdb ...
> stop at MyProgram:32
Deferring breakpoint MyProgram:32.
It will be set after the class is loaded.
> run
run MyProgram
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint MyProgram:32

Breakpoint hit: "thread=main", MyProgram.main(), line=32 bci=6
32                  op.execute("My String!");

main[1]
```

Setting a breakpoint on this line

Running the program

Hit breakpoint, ready to inspect the program state.

Let’s debug a program

JDB

Applications typically run for as long as they can (Webservers) so allow for functionality that allows us to inspect a currently running application gives this the opportunity to debug live programs.

This kind of interactions is what allows the programmer to debug programs requiring user input.

JDB

When running your program, we can set the virtual machine to allow debug and open a port for JDB to connect to.

```
> java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n MyProgram
```

Open up another window and use the following command. JDB will connect to the open session and interact with this.

```
> jdb –attach 8000
```

JDB

When running your program, we can set the virtual machine to allow debug and open a port for JDB to connect to.

```
> java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n MyProgram
```

Open up another window and use the following command. JDB will connect to the open session and interact with this.

```
> jdb -attach 8000
```

We have specified the socket the jdwp server is listening on and the port jdb will connect to.

How do we get more from the compiler?

Annotations

Java includes a few built in annotations that the programmer can use. These allow the compiler to assert user defined constructs to ensure a constraint is adhered to.

Annotations can be used at run time and used with java’s reflection api to dynamically load classes and fields.

Annotations

Some scenarios where want to utilise annotations without our program to ensure that it does not break.

- Ensure that a method is overriding an inherited method
- Interface remains a functional interface (so it does not break our existing lambda methods)
- Mark methods as being deprecated to warn other programmers not to use your method

Annotations

Common annotations built into java.

@FunctionalInterface

This asserts that the interface only contains one abstract method. It will create a warning if it is not the case.

@Override

Asserts that the method implementation is overriding a method from the parent class.

Annotations

Common annotations built into java.

@Deprecated

Warns programmers not to utilise the method as it will eventually be removed.

@SuppressWarnings

For any warnings that the programmer can safely remove.

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out.println(o);
                op.execute("My String!");
                op.execute(Integer.valueOf(5));
        }
}
```

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out.println(o);
                op.execute("My String!");
                op.execute(Integer.valueOf(5));
        }
}
```

Specified an annotation on the interface, declaring that the interface must only have one abstract method.

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out.println(o);
                op.execute("My String!");
                op.execute(Integer.valueOf(5));
        }
}
```

Specified an annotation on the interface, declaring that the interface must only have one abstract method.

The interface only contains one and will not throw a compilation error.

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out.println(o);
                op.execute("My String!");
```

Specified an annotation on the interface, declaring that the interface must only have one abstract method.

The interface only contains one and will not throw a compilation error.

This guarantees and communicates clearly to other programmers that the interface will be used with lambda expressions and method references.

```
> java FunctionExample
My String!
5
<Program End>
```

What if we had two abstract methods?

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
        void run(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out.println(o);
```

Not only does it violate the lambda expression, it violates the annotation.

```
> javac FunctionExample.java
FunctionalExample.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Operation is not a functional interface
        multiple non-overriding abstract methods found in interface
Operation
1 error
```

Annotations

Example of functional interface annotation

```
@FunctionalInterface
interface Operation {
        void execute(Object data);
        void run(Object data);
}

public class FunctionalExample {
        public static void main(String[] args) {
                Operation op = (o) -> System.out
```

Not only does it violate the lambda expression, it violates the annotation.

Attempting to compile the code will highlight the constraint being violated.

```
> javac FunctionExample.java
FunctionalExample.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Operation is not a functional interface
        multiple non-overriding abstract methods found in interface
Operation
1 error
```

Annotations

Example of method override annotation

```
class Cat {
        protected String name;
        public Cat(String name) {
                this.name = name;
        }

        public String roar() { return "meow"; }
}

class Lion extends Cat {
        public Lion(String name) { super(name); }

        @Override
        public String roar() { return "roar!"; }
}


public class OverrideExample {

        public static void main(String[] args) {
                Cat felix = new Cat("Felix");
                Cat lion = new Lion("Simba");
                felix.roar();
                lion.roar();
        }
}
```

Annotations

Example of method override annotation

```
class Cat {
        protected String name;
        public Cat(String name) {
                this.name = name;
        }

        public String roar() { return "meow"; }
}

class Lion extends Cat {
        public Lion(String name) { super(name); }

        @Override
        public String roar() { return "roar!"; }
}


public class OverrideExample {

        public static void main(String[] args) {
                Cat felix = new Cat("Felix");
                Cat lion = new Lion("Simba");
                felix.roar();
                lion.roar();
        }
}
```

Specified an annotation to assert that the method **roar** will return a string and correctly override a method from the parent

Annotations

Example of method override annotation

```
class Cat {
        protected String name;
        public Cat(String name) {
                this.name = name;
        }

        public String roar() { return "meow"; }
}

class Lion extends Cat {
        public Lion(String name) { super(name); }

        @Override
```

Specified an annotation to assert that the method **roar** will return a string and correctly override a method from the parent

The source can be compiled as no constraints have been violated.

```
> javac OverrideExample.java
```

Annotations

Example of method override annotation

```
class Cat {
        protected String name;
        public Cat(String name) {
                this.name = name;
        }

        public String roar() { return "meow"; }
}

class Lion extends Cat {
        public Lion(String name) { super(name); }

        @Override
        public Object roar() { return "roar!"; }
}


public class OverrideExample {

        public static void main(String[] args) {
                Cat felix = new Cat("Felix");
                Cat lion = new Lion("Simba");
                felix.roar();
                lion.roar();
        }
}
```

However, but simply changing the return type of the subclass method, the method violates the annotation guarantee.

Annotations

Example of method override annotation

```
class Cat {
        protected String name;
        public Cat(String name) {
                this.name = name;
        }

        public String roar() { return "meow"; }
}

class Lion extends Cat {
        public Lion(String name) { super(name); }

        @Override
```

However, but simply changing the return type of the subclass method, the method violates the annotation guarantee.

```
OverrideExample.java:14: error: roar() in Lion cannot override roar() in Cat
    public Object roar() { return "roar!"; }
                  ^
  return type Object is not compatible with String
OverrideExample.java:13: error: method does not override or implement a method
from a supertype
    @Override
    ^
2 errors
```

Quick demo!

See you next time