Debugging

- If testing has shown the presence of a fault, then we must track down the cause of the fault and repair it
  - This is debugging
- The overall process is:
  - Locate the fault
  - Design a repair
  - Carry out the repair
  - ***Re-test the program!***
  - ***Document the fault and the repair***
- Re-testing the program is important:
  - To check whether the repair worked
  - To check whether further faults have been introduced:
    ⇒ in the repaired code,
    ⇒ or in code dependent on the repaired code!
- We will focus on locating faults

Locating faults

- If the cause of a fault is *not* "obvious", then we usually need to investigate the program or components under test
- Approaches to the investigation:
  - Manually tracing the faulty test case through the code
    - clearly problematic in complex/large systems
  - Designing and running extra tests to localise/ characterize the fault
    - tests forcing or avoiding execution of parts of the code may indicate where the fault is/is not
    - tests more or less similar to the test exposing the fault may indicate the area of functionality at fault (compare: equivalence partitioning)
  - Instrumenting the code with diagnostic statements
    - allows us to see what path execution takes and what the values of variables of interest are
  - Using an interactive debugger to monitor the code
    - similar to instrumenting

Using diagnostic statements

- We can add extra statements to the code to monitor
  - What path execution takes
  - What values are computed and held in variables
- Simple approaches are "`System.out.println`" or equivalent statements for producing a diagnostic log:
  - Can produce excessive output
  - Not "controlled" enough
  - Inflexible without re-editing
  - Code needs editing (and hence un-editing!)
  - Ideally we would like to investigate the *actual unedited* code
- Could be clever:

```
private boolean debugging = false;    (or true)
...
if (debugging) System.out.println(...);
```


Interactive debugging

- Interactive debuggers allow us to run the *actual compiled code* for a program
  - Under our control: starting and stopping the program, monitoring variable values
- For this to be really useful the compiled code must retain information from the source code: Typically
  - Links to source code file names/line numbers
  - The symbolic names for variables, methods, classes used in the source code
- This allows the interactive debugger to report a program's state to us in source code terms – and not perhaps just as RAM addresses!
- Most compilers have an option to indicate to the compiler that it should retain this information, eg:

```
javac -g foo.java
```

Interactive debugging

- Many IDEs offer *interactive debugger* facilities
  - For Java: Eclipse (& Together), BlueJ, ...
- A standalone Java debugger: JSwat
  - https://github.com/nlfiedler/jswat
- Interactive debuggers typically allow:
  - Setting "breakpoints" to automatically pause the program
  - "Stepping" by single statements from breakpoints
  - Inspecting the values of variables
  - Possibly *modifying* the values of variables
  - "Watches" : monitoring nominated variables
  - Evaluating expressions using current variables
- We'll take a quick look at Eclipse...
  (it is the JDT plug-in and not Together providing these functions)

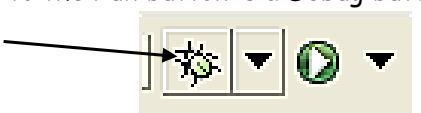
How do *breakpoints* work?

- "Traditional" scheme:
  - Programs compile to machine code
  - The debugger loads the compiled machine code into RAM
  - Before jumping to the start address of the program, the debugger *replaces* the *machine instruction* at any address where we would like to break the execution with an unconditional branch (or subroutine jump) *into its own code*
  - So the program runs to the breakpoint and then jumps to the debugger, which then interacts with the programmer
  - To resume execution of the program:
    - If the bp is to remain in effect then a copy of the replaced instruction is artificially executed within the debugger, which then jumps back to the program just after the bp.
    - If the bp has been cancelled, then the replaced instruction is restored and the debugger jumps to it.

- The JVM scheme:
  - Programs compile to bytecode
  - The JVM is a hybrid interpreter / adaptive compiler
    - "hotspots" identified may be compiled to native code
  - The JVM can be launched in a *debugging mode* in which it interprets the bytecode as normal, but "listens" for commands from a separate debugging program
    (which could even be on a remote computer!)
    Socket based communication
  - The JVM can be instructed to stop at breakpoints, fetch variable values, …
  - The JVM can stop interpretation of the bytecode at nominated locations - based on source code line numbers

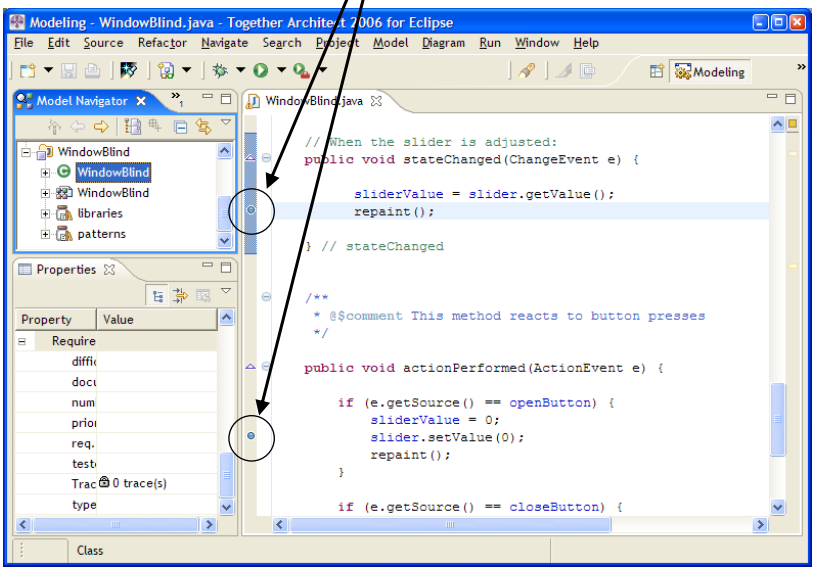
Eclipse's debugger

- Breakpoints can be set by double-clicking next to lines of code in an Editor pane (see next slide)
  - This can be done *before activating the debugger* – to get control immediately
- Next to the Run button is a Debug button:

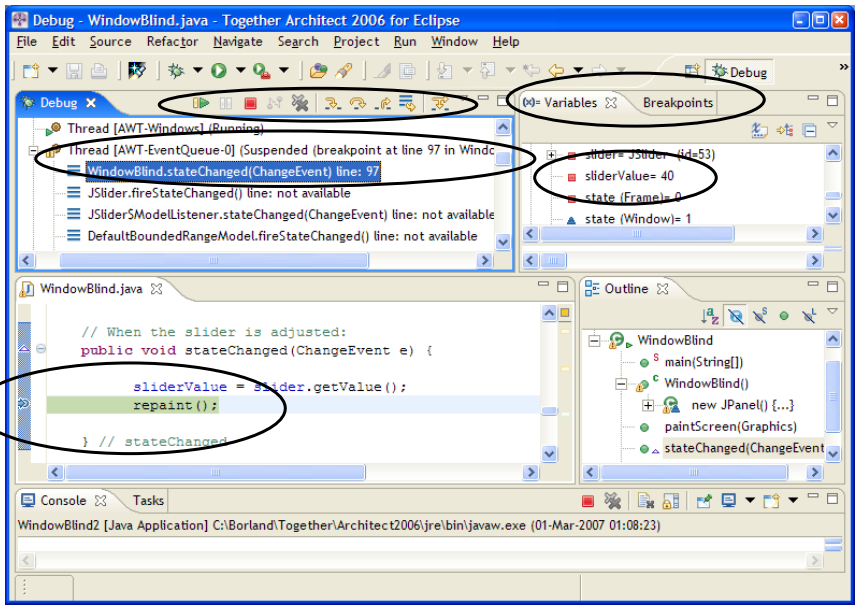


- This compiles (if necessary), and starts the application running
- The application runs until it hits a breakpoint
  - Eclipse then offers to switch to the Debugger perspective
  - The Debugger perspective offers a very detailed view of the state of the program
  - Breakpoint information, variable values, step/resume control buttons are located in the debugger sub-panels
  - (See the slide after next)

Double-click to set/unset breakpoints

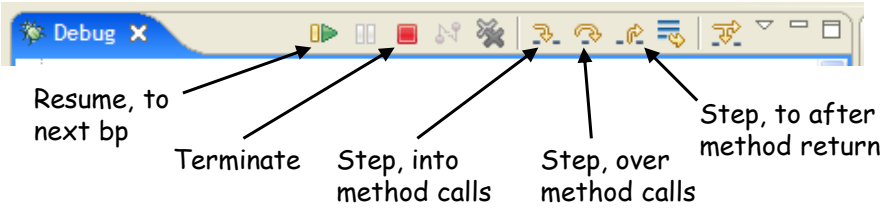


At breakpoint in **stateChanged**



Controlling the debugger

- Extra breakpoints can be set, and breakpoints can be removed, *as required during debugging* in the code editor pane
- Breakpoints can be controlled by "hit counters" or condition expressions
  - Right-click on a breakpoint in the top right pane
- Variables can be modified, or targeted as the subject of "watches"
  - Right-click on a variable in the top right pane
- The Debug toolbar has useful tools:



End of lecture