Assertions

- supports testing
  - assertions can be used for correctness checks that are run any time the code is (*), not only when specific tests are run
    - used for *invariants* – things that are always true (in correct code)
    - can be left in production code – meant to be permanent correctness-checking
    - (*) must turn on assertion checking when the code is run – it is off by default

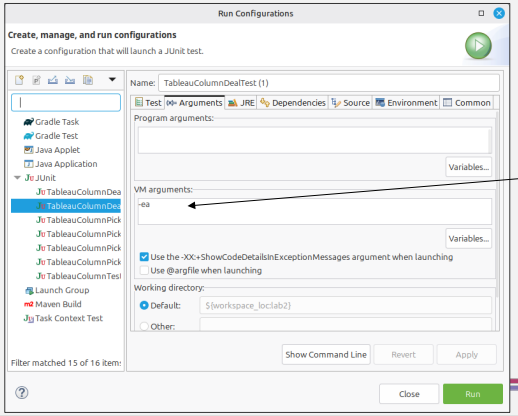
```
/**
 * If this column is non-empty and its top card is face-down, turns it
 * face-up. Also asserts, as an internal invariant check, that the column's
 * top-card-is-face-up property holds once this method returns.
 */
private void flipNewTopIfNeeded () {
  if ( !cards_.isEmpty() && !cards_.get(cards_.size() - 1).isFaceUp() ) {
    cards_.get(cards_.size() - 1).flip();
  }
  assert cards_.isEmpty() || cards_.get(cards_.size() - 1)
      .isFaceUp() : "Invariant violated: top card of a non-empty column must be face-up";
}
```

`TableauColumn` is responsible for making sure the top card is face up – checking that it is at the end of every method that modifies the column's contents helps catch bugs

(here the check is bundled into a helper to avoid repeated code)

Enabling Assertion Checking

In Eclipse: right-click the class with `main` → **Run As** → **Run Configurations…** → **Arguments** tab → add the flag to the **VM arguments** box → **Run**.



add -ea in the VM arguments in the run configuration to enable assertions

Logging

```
.level = INFO
java.util.logging.ConsoleHandler.level = INFO
```

- supports debugging
  - can leave informational messages in place and control what is seen at runtime without changing code
- Java has a logging API – `java.util.logging.Logger`

```
public class TableauColumn {

  private static final Logger LOGGER =
      Logger.getLogger(TableauColumn.class.getName());
```

typically a separate logger is associated with each class – this lets you control logging on a per-class basis

`static` so that all instances of the class share the same logger

ALL CAPS is the convention for naming constants

`TableauColumn.class` is the `Class` object associated with `TableauColumn` (every class has one) and `getName()` gets its fully qualified name (*`pkg.classname`* form) – naming the logger after the class makes it easy to keep track of them and allows log levels to be configured at the class and package level

```
public void deal(Deck deck, int count) {
    if (count <= 0) {
        throw new IllegalArgumentException("count must be positiv
    }
    for (int i = 0; i < count; i++) {
        Card dealt = deck.deal();
        cards.add(dealt);
        LOGGER.fine(() -> "Drew " + dealt + " from deck");
    }
    flipNewTopIfNeeded();
    LOGGER.info(() -> "Dealt " + count + " cards into column");
}
```

there are methods to log at each of the various log levels

the simpler version just takes a String message: `LOGGER.fine("Draw "+dealt+" from deck");`

the version shown, with lambda expressions, takes a no-parameter function that will generate the log message

Logging Levels

- logging frameworks organize log messages by *severity level*

Level	Meaning
`SEVERE`	A serious failure — something is broken
`WARNING`	Something unexpected, but not fatal
`INFO`	Routine, high-level status — "dealt 4 cards to column 2"
`CONFIG`	Configuration-related information
`FINE` / `FINER` / `FINEST`	Increasingly detailed debug-level tracing

  - the level assigned to a log message is a statement about how important it is to see, not whether it is "debug code" to be removed later
- logging output is controlled by adjusting the logging threshold and can be routed to different handlers (console, file, etc)
  - doesn't require editing and rebuilding code

Logging

Logging settings are defined in a separate configuration file.
- need to specify the log level for both loggers and handlers

- convention is to name the file `logging.properties` and put it in the top-level directory in the project

```
.level = INFO
java.util.logging.ConsoleHandler.level = INFO
```

the root logger is named "" (empty string), so `.level` sets the root logger's level (inherited by all other loggers)
– `.level = INFO` means only `INFO` or higher levels pass

`ConsoleHandler` writes log messages to the console –
`java.util.logging.ConsoleHandler.level = INFO` means that any `INFO` or higher messages will be displayed

a message is only logged if it passes both the log's level and the handler's level

- the JVM must be told the location of the `logging.properties` file at runtime
  - add `-Djava.util.logging.config.file=logging.properties` to the VM arguments
    - adjust the path if the file is named differently and/or isn't at the top level of the project directory

In Eclipse: right-click the class with `main` → **Run As** → **Run Configurations…** → **Arguments** tab → add the flag to the **VM arguments** box → **Run**.

Logging

- other handlers
  - FileHandler writes log messages to a file
  - SocketHandler sends log messages over a network socket to a remote host
  - MemoryHandler buffers log messages in memory
  - StreamHandler writes log messages to any OutputStream
  - ConsoleHandler writes log messages to System.err
- a logger can have multiple handlers at once, with the same or different log levels
- different classes can have different log levels

```
.level = INFO
java.util.logging.ConsoleHandler.level = FINE

solitaire.model.TableauColumn.level = FINE
solitaire.model.GameEngine.level = WARNING
```

```
solitaire.model.level = FINE
```

a message must pass both the logger's level and the handler's level to be displayed – so only `WARNING` level message will be displayed for `GameEngine` on the console

sets the level for every class in the `solitaire.model` package

Integration Testing

- typically done incrementally
  - add and test at most a few components at a time
- *bottom-up* tests low-level components first, then combines them into larger subsystems
  - low-level pieces are tested early and thoroughly
  - *drivers* stand in for higher-level components (something that would call the thing being tested) that aren't built yet
- *top-down* tests high-level components first, then fills in lower levels
  - overall structure and control flow is validated early
  - *stubs* stand in for lower-level components (something that would be called by the thing being tested) that aren't built yet
- *sandwich* (or *hybrid*) does bottom-up and top-down at the same time
  - practical in divide-and-conquer team environments
    - e.g. for klondike, one person works bottom-up on game engine internals while another works top-down on the overall game loop
  - requires both drivers and stubs

Mocking and Test Doubles

- top-down and sandwich approaches often mean needing to test code that depends on something unfinished
  - e.g. you are implementing the game engine while your partner works on `TableauColumn` and the other pile classes
- or the logic depends on something slow or unpredictable
  - e.g. a random shuffle

- the idea: substitute a temporary stand-in (a *test double*)
  - a *stub* returns canned, fixed answers
  - a *fake* is a simplified but working implementation
  - a *mock* also records how its operations were invoked
    - allows test to check that the calling code interacts correctly – e.g. that `save()` was called exactly once

(note this is a different context and a different meaning for "stub" – though it isn't just a coincidence of terminology since stubs in bottom-up integration testing are often implemented using canned-answer stubs)

Substituting a Double

- example – testing `TableauColumn`'s deal-the-column method

```
public void deal(Deck deck, int count) {
    for (int i = 0; i < count; i++) {
        Card dealt = deck.deal();
        cards.add(dealt);
        // ...
    }
}
```

testing this means we also need `Deck` … and testing with Deck is also a bit tricky because we are limited in the order of cards – they are either in a standard order or random

Substituting a Double

- the key is *abstraction* – the caller should depend on an *interface* rather than a concrete class...because then another version can be substituted

```
public interface Deck {
    Card deal();
    void shuffle();
    boolean isEmpty();
}
```

```
public class StandardDeck implements Deck {
    private final List<Card> cards = /* a real, shuffled 52
    @Override
    public Card deal() {
        return cards.remove(cards.size() - 1);
    }
    @Override
    public void shuffle() { ... }
    @Override
    public boolean isEmpty() { ... }
}
```

```
public class FixedOrderDeck implements Deck {
    private final List<Card> remaining;
    public FixedOrderDeck(List<Card> order) {
        this.remaining = new ArrayList<>(order);
    }
    @Override
    public Card deal() {
        return remaining.remove(0);
    }
    @Override
    public void shuffle() {
        // no-op
    }
    @Override
    public boolean isEmpty() {
        return remaining.isEmpty();
    }
}
```


Substituting a Double

- `deal` is unchanged
  - the caller creates an instance of `FixedOrderDeck` or `StandardDeck` as desired and passes it to `deal`

```
public void deal(Deck deck, int count) {
    for (int i = 0; i < count; i++) {
        Card dealt = deck.deal();
        cards.add(dealt);
        // ...
    }
}
```


Designing for Testing

- tests often need to verify an object's internal state, but –
  - adding a special testing-only getter undermines the class's contract
  - adding package-private/`protected` access just for testing breaks encapsulation
- strategies
  - utilize public methods that are already part of the class – or that reasonably could be
    - e.g. adding a `top()` method to `TableauColumn` to retrieve the top card on the column
  - prefer checking observable behavior over checking hidden internals
    - e.g. `GameEngine`'s `moveCards` can check that `from.size()+to.size()` is the same before and after the move
  - the class that owns a piece of state should be the one that checks any invariant about it
    - e.g. that the top card in a tableau column is always face up should be checked by `TableauColumn` in the methods that change the column's contents – use assertions
  - a package-private accessor is a legitimate fallback if nothing else works – but should be a fallback, not a starting point

How Many Tests is Enough?

- *test coverage* measures what fraction of the code gets executed by a set of tests
- most useful as a way to find blind spots rather than a score to maximize
  - code not covered definitely hasn't been tested
  - high coverage isn't a guarantee – 100% coverage can still miss logic bugs
    - a test may execute a line but not check everything that happens
      - e.g. only checking that two cards were returned by pickUp(2)
    - coverage only counts that a line was reached at some point – it doesn't check all inputs that could reach that line
      - e.g. "normal" tests could cover every line of code, but edge cases still break
    - the test and code can share the wrong assumptions – verification vs validation
    - coverage doesn't address interaction between components
- balance with risk
  - aim for 100% coverage of higher-risk code – but recognize that's not enough by itself

What to Test

- *black-box* tests cover the specification
  - designed based on the expected behavior without looking at the implementation
  - addresses "does this do what it is supposed to do?"
  - test-driven development focuses on black-box
  - the `TableauColumn` tests in lab 2 are black-box
- *white-box* tests cover the implementation
  - designed to cover all of the paths through the code
  - addresses "did all the code written actually get checked?"
  - e.g. make sure each branch of an if/else is covered by at least one test

What to Test

- both black-box and white-box tests are important
  - start with black-box to establish correct behavior and the expected specifications
  - after code is written, augment with white-box to ensure sufficient coverage

Writing a Test Plan

- a *test plan* is an outline of what you intend to test and why
  - doesn't need to be formal
  - tests require specific starting state, input
  - test plan identifies the broader category of behavior that each specific test represents
- a basic template
  - method/feature being tested
  - list of scenarios (normal cases, edge cases, error cases)
  - expected behavior for each scenario
- writing a test plan before starting coding has advantages
  - identify and resolve ambiguities in specifications before coding
  - facilitates designing for testing from the beginning

Writing a Test Plan

- example – `TableauColumn pickUp(n)`
  - normal
    - pick up a valid number of face-up cards from the top of the column
  - edge
    - pick up the entire column
    - pick up 0 cards
  - error/edge
    - pick up more cards than are available in the column
    - pick up more than are face-up
    - pick up starting from a face-down card
  - expected behavior
    - for legal pick up scenarios, picked-up cards are removed from the column and returned
    - what should happen when the pickup includes the last face-up card?
    - for illegal pick up scenarios, should it be the caller's responsibility to know the pick up is legal or should pick up notify of an illegal pick up attempt?