

# JUnit exercise

Given a `Date` class with the following methods:

```
- public Date(int year, int month, int day)
- public Date() // today
- public int getDay(), getMonth(), getYear()
- public void addDays(int days) // advances by days
- public int daysInMonth()
- public String dayOfWeek() // e.g. "Sunday"
- public boolean equals(Object o)
- public boolean isLeapYear()
- public void nextDay() // advances by 1 day
- public String toString()
```

- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

# Test-Driven Development (TDD)

- Imagine that we'd like to add a method `subtractWeeks` to our `Date` class, that shifts this `Date` backward in time by the given number of weeks.
- Write code to test this method *before* it has been written.
  - This way, once we do implement the method, we'll know whether it works.

# Black and white box testing

What is the difference between black- and white-box testing?

- **black-box** (procedural) **test**: Written without knowledge of how the class under test is implemented.
  - focuses on input/output of each component or call
- **white-box** (structural) **test**: Written with knowledge of the implementation of the code under test.
  - focuses on internal states of objects and code
  - focuses on trying to cover all code paths/statements
  - requires internal knowledge of the component to craft input
    - example: knowing that the internal data structure for a spreadsheet uses 256 rows/columns, test with 255 or 257

# Black-box testing

- black-box is based on requirements and functionality, not code
- tester may have actually seen the code before ("gray box")
  - but doesn't look at it while constructing the tests
- often done from the end user or OO client's perspective
- emphasis on parameters, inputs/outputs (and their validity)

# Types of black-box

- requirements based
- positive/negative
  - checks both good/bad results
- boundary value analysis
- decision tables
- equivalence partitioning
  - group related inputs/outputs
- state-based
  - based on object state diagrams
- compatibility testing
- user documentation testing
- domain testing

# Boundary and Equivalence testing

- **boundary value analysis:** Testing conditions on bounds between classes of inputs.
- **equivalence partitioning:**
  - A black-box test technique to reduce # of required test cases.

# Example

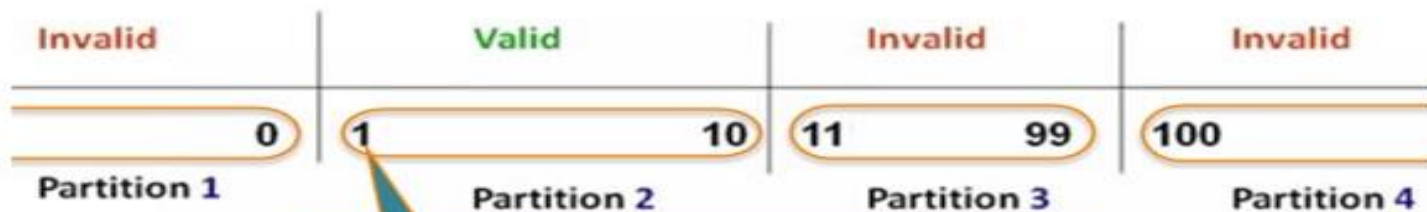
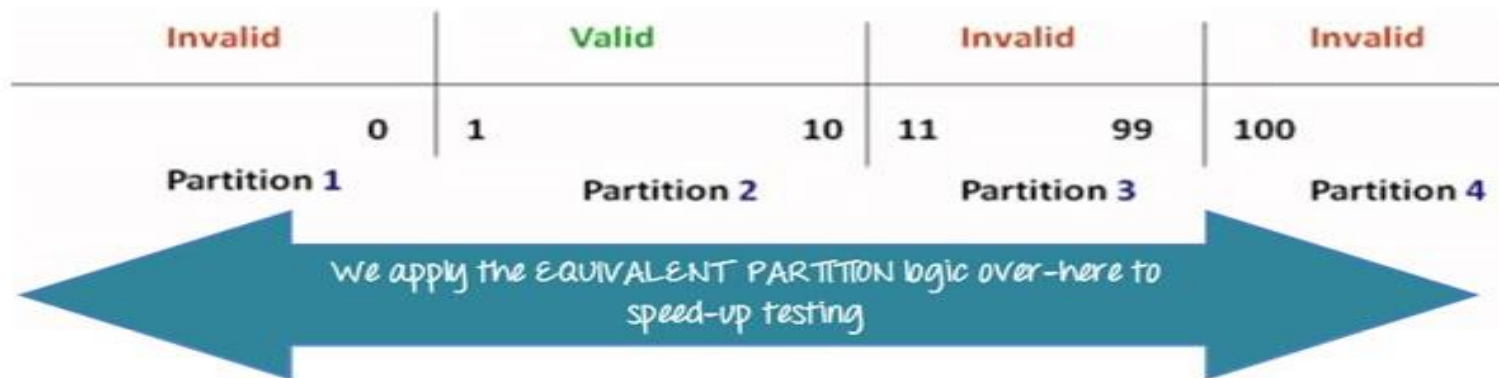
- Let's consider the behavior of Order Pizza Text Box Below
- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, **"Only 10 Pizza can be ordered"**

Order Pizza:

Submit

Here is the test condition

1. Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

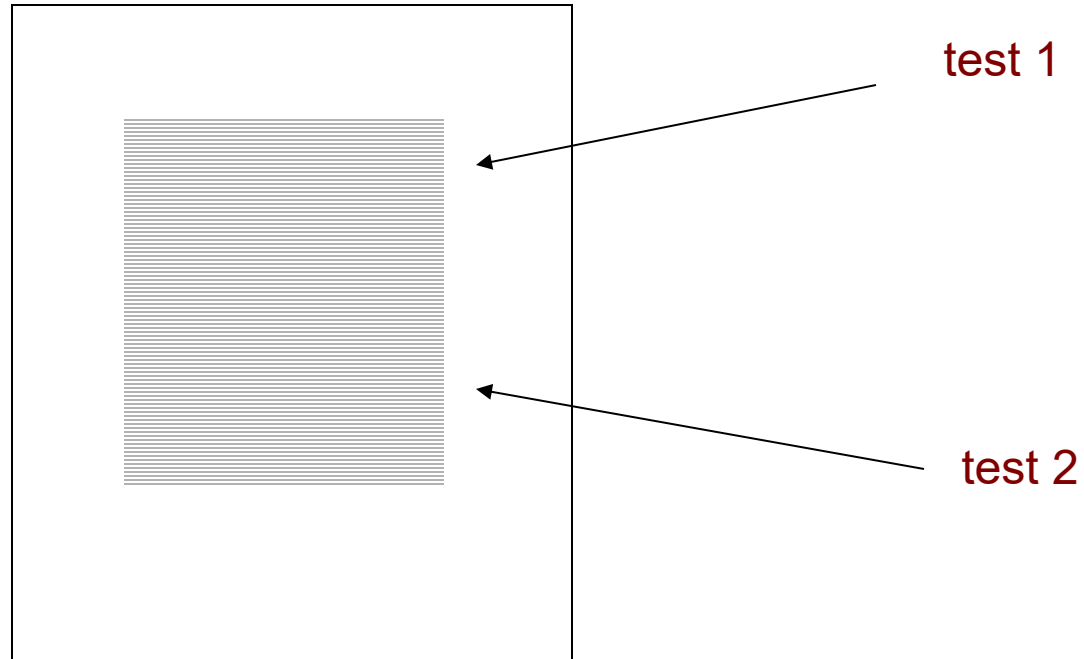


In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100



# White box tests

Based on code



# White-box testing

Some kinds of white box testing don't involve unit tests:

- "static testing"
  - code walkthroughs, inspections, code reviews
  - static analysis tools
    - Lint (and variants) JiveLint, JLint, [PMD](#), CheckR, JSLint, `php -l`
    - CheckStyle <http://checkstyle.sourceforge.net/>
    - FindBugs <http://findbugs.sourceforge.net/>
  - code complexity analysis tools
    - PMD, CheckStyle, etc.

# Static analysis example

The screenshot shows the FindBugs: Critters application interface. The left pane displays a tree view of bugs, with 'Bad practice (17)' expanded to show 'Method ignores exceptional return value (2)'. The right pane shows the source code of 'ClassUtils.java in', with line 218 highlighted: `new File(classFileName).renameTo(new File(classFileName + className + CLASS_EXTENSION));`. The bottom pane displays the bug report: 'ClassUtils.writeAndLoadClass(String, String, boolean) ignores exceptional return value of java.io.File.renameTo(File) At ClassUtils.java:[line 218]'. Below this, a detailed description explains that the method returns a value that is not checked, and that the `File.delete()` method returns false if the file could not be successfully deleted.

**FindBugs: Critters**

File Edit View Navigation Designation Help

Class search strings:

Category Bug Kind Bug Pattern ↔ Bug Rank

Bugs (56)

- Bad practice (17)
  - Bad use of return value from method (2)
    - Method ignores exceptional return value (2)
      - ClassUtils.writeAndLoadClass(String, String, boolean)
      - ClassUtils.writeAndLoadClass(String, String, boolean)
  - Dropped or ignored exception (4)
  - Incorrect definition of Serializable class (4)

Classify: unclassified

ClassUtils.java in View in browser

```
208     if (useTempFolder) {
209         javaFileName = System.getProperties().getProperty("java.io.tmpdir")
210             + File.separatorChar + "temp" + File.separatorChar + "temp";
211     }
212     writeEntireFile(fileText, javaFileName);
213
214     String classFileName = compile(javaFileName);
215     new File(javaFileName).delete();
216
217     // move class to current directory
218     new File(classFileName).renameTo(new File(classFileName + className + CLASS_EXTENSION));
219
220
221     return loadClass(classFileName);
```

Find Next Previous

ClassUtils.writeAndLoadClass(String, String, boolean) ignores exceptional return value of java.io.File.renameTo(File)  
At ClassUtils.java:[line 218]

**Method ignores exceptional return value**

This method returns a value that is not checked. The return value should be checked since it can indicate an unusual or unexpected function execution. For example, the `File.delete()` method returns false if the file could not be successfully deleted (rather than throwing an Exception). If you don't check the result, you won't notice if the method invocation signals unexpected behavior by returning an atypical return value.

<http://findbugs.sourceforge.net>

UNIVERSITY OF MARYLAND

# Complexity analysis

The screenshot displays the PMD Eclipse plugin interface. The top window shows the source code of `TTTGame.java` in the `ttt.model` package. The code includes package declarations, imports, and a class definition for `TTTGame` extending `Observable`. The bottom window shows the `Violations Overview` table, which lists various violations found in the code.

**Violations Overview Table:**

Element	# Violations	# Violations/...	# Violati...
TTTGame.java	48	355.6 / 1000	4.0
SystemPrintln	1	7.4 / 1000	0.0
ShortVariable	2	14.8 / 1000	0.1
MethodArgumentCouldBeFinal	(max) 5	37.0 / 1000	0.4
DataflowAnomalyAnalysis	2	14.8 / 1000	0.1
ConstructorCallsOverridableMei	2	14.8 / 1000	0.1
LocalVariableCouldBeFinal	(max) 5	37.0 / 1000	0.4

# Testing measures (white box)

- Code coverage – individual modules
- Path coverage – sequence diagrams
- Code coverage based on complexity – test of the risks, tricky part of code (e.g., Unix “you are not expected to understand this” code)

# Code coverage testing

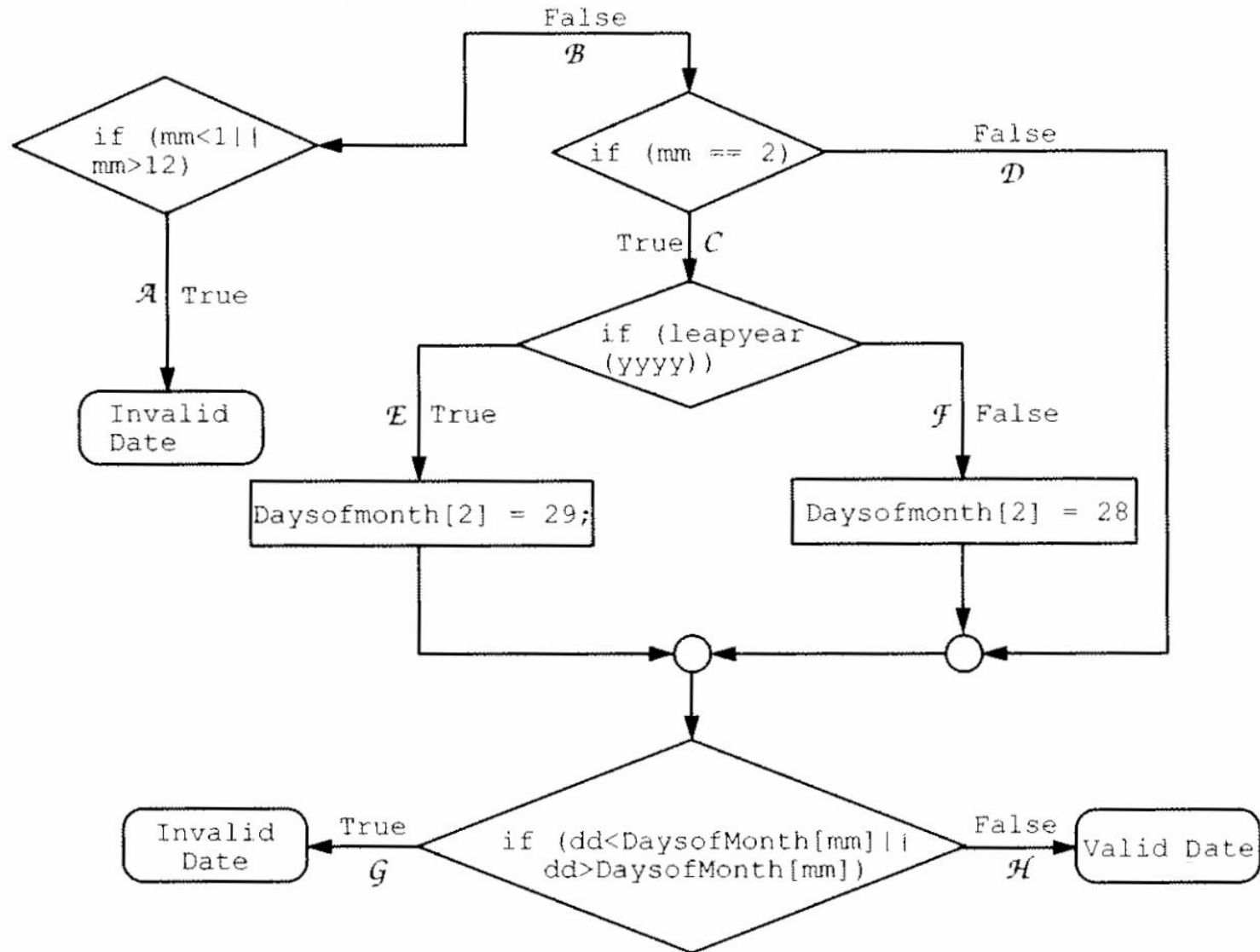
- **code coverage testing:** Examines what fraction of the code under test is reached by existing unit tests.
  - statement coverage      - tries to reach every line (impractical)
  - path coverage            - follow every distinct branch through code
  - condition coverage      - every condition that leads to a branch
  - function coverage        - treat every behavior / end goal separately
- Several nice tools exist for checking code coverage
  - EclEmma, Cobertura, Hansel, NoUnit, CoView ...

# Path testing

- **path testing:** an attempt to use test input that will pass once over each path in the code
  - path testing is *white* box
  - What would be path testing for `daysInMonth(month, year)`?  
some ideas:
    - error input: `year < 1`, `month < 1`, `month > 12`
    - one month from `[1, 3, 5, 7, 10, 12]`
    - one month from `[4, 6, 9, 11]`
    - month 2
      - in a leap year, not in a leap year



# Path coverage example





# White box testing is hard

- Developers can't easily spot flaws in their own code.
- Test cases that are too focused on code may not be thinking about how the class is actually going to be used.
- Code coverage tools can give a false sense of security.
  - Just because code is "covered" doesn't mean it is free of bugs.
- Code complexity can be misleading.
  - Complex code is not always bad code.
  - Complexity analysis tools can be overly picky or cumbersome.