Plan for Today

• Assignment 1
• Testing
• *Effective Java* Items
• Liskov Chapter 3
• In-Class Exercise
Software Process

- Requirements
- Design
- Implementation
- Testing
- Maintenance
Testing
Why Test?

• Meeting specifications/requirements
• Find bugs
Brooks’ rule of thumb for scheduling a software project

• 1/3 design
• 1/6 coding
• 1/2 testing
  - 1/4 component testing
  - 1/4 system testing
• Black-box testing
  - Based on specifications/requirements or what the software should do

• White-box testing
  - Based on knowledge of requirements, design, and implementation
Types of Black-Box Testing

• Specifications/Requirements
• Equivalence Partitioning
• Boundary Testing
Test Case Information

• Unique Identifier
• Description with preconditions and input
• Expected output from the program or program unit
• Actual results of running the test case

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestName (Test Author)</td>
<td>Preconditions</td>
<td>Test outputs</td>
<td>Actual outputs</td>
</tr>
<tr>
<td>Test Type:</td>
<td>Test Inputs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types of White Box Testing

• Statement Coverage
• Branch Coverage
• Path Coverage
• Equivalence Partitioning
• Boundary Value
```java
int computeFine(int daysLate, boolean printOn) {
    int MAX_FINE_PERIOD = 21; fine = 0;
    if (daysLate <= MAX_FINE_PERIOD) {
        fine = daysLate * DAILY_FINE;
    }
    logFine(fine);
    if (printOn == TRUE) {
        printFine(fine);
    }
    return fine;
}
```

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>dayLate</th>
<th>printOn</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>TRUE</td>
<td>1-2-3-4-5-6-7</td>
</tr>
</tbody>
</table>
```java
int computeFine(int daysLate, boolean printOn) {
    int MAX_FINE_PERIOD = 21; fine = MAX_FINE;
    if (daysLate <= MAX_FINE_PERIOD) {
        fine = daysLate * DAILY_FINE;
    }
    logFine(fine);
    if (printOn == TRUE) {
        printFine(fine);
    }
    return fine;
}
```
Path Coverage

• All distinct code paths are executed by at least one test case

• Complete set of sequences of branches and loop traversals

• Subsumes statement and branch coverage
Related *Effective Java* Items

- Item 8: Obey the general contract when overriding equals (pp.33-44)
- Item 9: Always override hashCode when you override equals (pp.45-50)
- Item 10: Always override toString (pp.51-53)
- Item 13: Minimize the accessibility of classes and members (pp.67-70)
- Item 14: In public classes, use accessor methods, not public fields (pp.71- 72)
- Item 15: Minimize mutability (pp.73-80)
- Item 45: Minimize the scope of local variables (pp.209-211)
- Item 56: Adhere to generally accepted naming conventions (pp.237-240)
Item 8: Obey the general contract when overriding `equals`

[Bloch]

You want to leave the default method if:

• Each instance of the class is inherently unique.

• You don’t care whether the class provides a “logical equality” test.

• A superclass has already overridden `equals`, and the superclass behavior is appropriate for this class.

• The class is private or package-private, and you are certain its `equals` method will never be invoked.
Item 8: Obey the general contract when overriding equals [Bloch]

The equals method implements an equivalence relation. It is:

- Reflexive
- Symmetric
- Transitive
- Consistent
- For any non-null reference value x, x.equals(null) must return false.
Item 8: Obey the general contract when overriding equals

[Bloch]

A recipe for a high-quality equals method:

• Use the == operator to check if the argument is a reference to this object.

• Use the instanceof operator to check if the argument has the correct type.

• Cast the argument to the correct type.

• For each “significant” field in the class, check if that field of the argument matches the corresponding field of this object.

• When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?
Item 9: Always override `hashCode` when you override `equals` [Bloch, JavaSE6]

- Whenever it is invoked on the same object more than once during an execution of an application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.
Item 10: Always override `toString` [Bloch]
Item 13: Minimize the accessibility of classes and members

[Bloch]
Item 45: Minimize the scope of local variables

[Bloch]
Item 14: In public classes, use accessor methods, not public fields

[Bloch]
Item 15: Minimize mutability

[Bloch]
Item 56: Adhere to generally accepted naming conventions

[Bloch]
Benefits of Abstraction

[Liskov]

• Abstraction by parameterization:
  - irrelevant: identity of the actual data
  - relevant: presence, number, and type of the actuals

• Abstraction by specification:
  - irrelevant: “how” is done
  - relevant: “what” is done
• locality
• modifiability
Specifications

• Formal
• Informal
Procedures

• Total

• Partial
Public static methods (of the Abc class):

- `defg : Abc x int --> int`
- `hijk : Abc x int --> Abc`
- `lmno : Abc x int --> Abc`
- `pqrs : int --> Abc`
- `tuvw : Abc --> int`

Algebraic Specification:

```plaintext
Abc.defg (Abc.lmno (u, k), n) = Abc.defg (u, n) if n < Abc.tuvw (u)
Abc.defg (Abc.lmno (u, k), n) = k if n == Abc.tuvw (u)
Abc.defg (Abc.lmno (u, k), n) = n if n > Abc.tuvw (u)
Abc.defg (Abc.pqrs (k), n) = 3
Abc.hijk (Abc.lmno (u, k), n) = Abc.lmno (Abc.hijk (u, n), k) if n < Abc.tuvw (u)
Abc.hijk (Abc.lmno (u, k), n) = Abc.lmno (u, n + 1) if n == Abc.tuvw (u)
Abc.hijk (Abc.lmno (u, k), n) = u if n > Abc.tuvw (u)
Abc.hijk (Abc.pqrs (k), n) = Abc.lmno (Abc.pqrs (0), k)
Abc.tuvw (Abc.lmno (u, k)) = 1 + Abc.tuvw (u)
Abc.tuvw (Abc.pqrs (k)) = 0
```