TDD and friends

#09 Workshop

IContext

```
public interface IContext {
    void writeLexeme(IToken token, IWriter writer)
        throws ContextException;

    void writeNewLine(IWriter writer) throws
ContextException;

    void writeIndent(IWriter writer) throws ContextException;

    void incrementIndent();
    void decrementIndent();
}
```

Lexer

```
while (lexer.hasMoreTokens()) {
    IToken token = lexer.readToken();
    String lexeme = token.getLexeme();
    if (lexeme.equals("{")) {
        //...
    } else if (lexeme.equals("}")) {
        //...
```

Final State as `null`

```
while (reader.hasMoreChars() && state != null) {
    val char = reader.readChar()
    val command = commands.getCommand(state, char)
    command.execute(char, context)
    state = transitions.nextState(state, char)
}
```

Final State as flag

```
while (reader.hasMoreChars() && !state.isFinal()) {
    val char = reader.readChar()
    val command = commands.getCommand(state, char)
    command.execute(char, context)
    state = transitions.nextState(state, char)
}
```

Final State as type

```
while (reader.hasMoreChars() && !(state is FinalState)) {
    val char = reader.readChar()
    val command = commands.getCommand(state, char)
    command.execute(char, context)
    state = transitions.nextState(state, char)
}
```

Two Key Maps

```
interface ILexerStateTransitions {
    fun nextState(state: State, char: Char): State?
}
interface ILexerCommandsRepository {
    fun getCommand(state: State, char: Char): ICommand
}
interface IFormatterStateTransitions {
    fun nextState(state: State, token: IToken): State?
}
interface IFormatterCommandsRepository {
    fun getCommand(state: State, token: IToken): ICommand
}
```

Two Key Maps

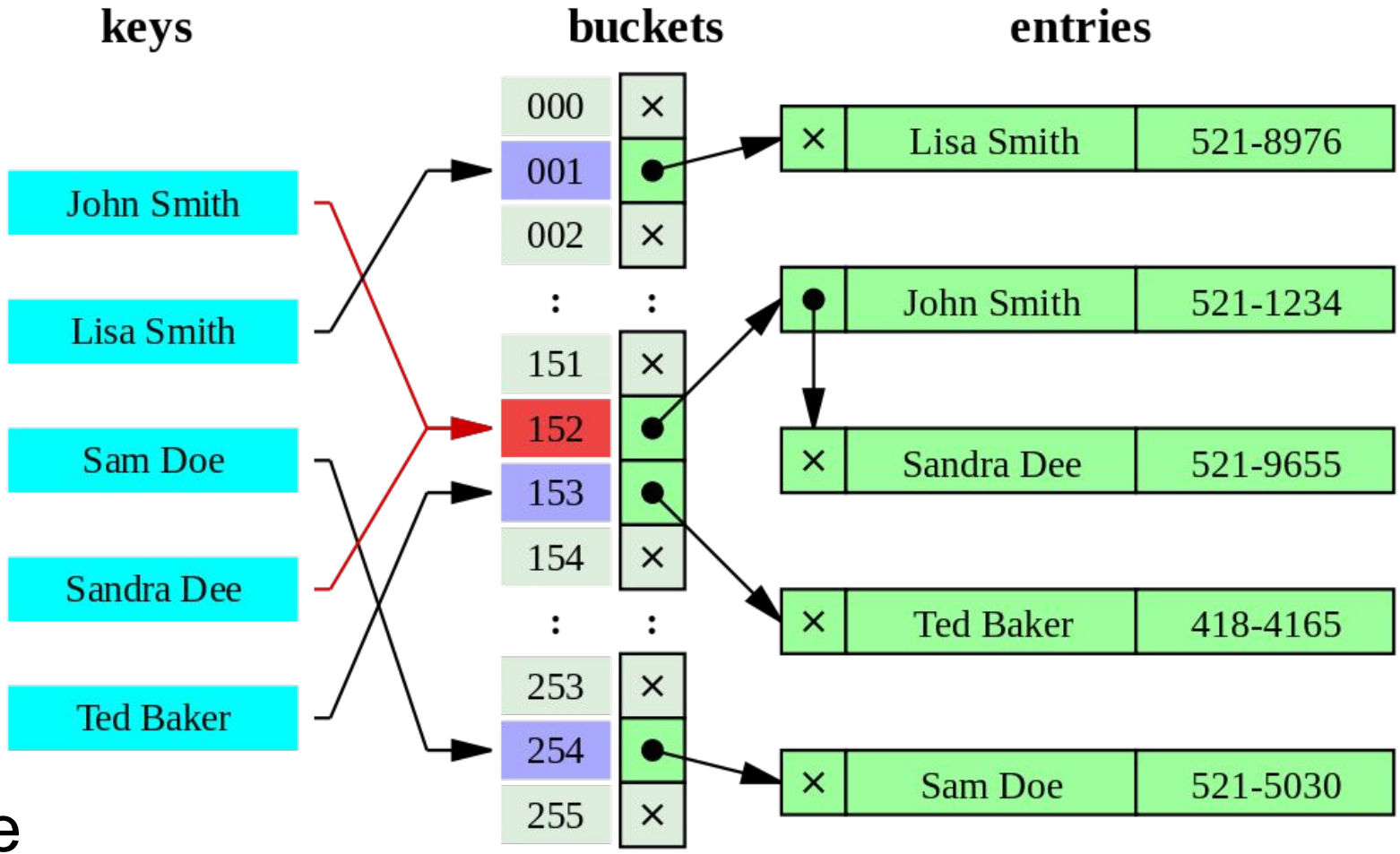
```
Map<Pair<State, Char>, State?>

Map<Pair<State, Char>, ICommand>

Map<Pair<State, IToken>, State?>

Map<Pair<State, IToken>, ICommand>
```

java.util.Map

```
Map<Integer, String> map;
map.put(5, "five");             // add value by key
map.size();                     // count values
map.containsKey(5);             // check key
map.containsValue("five");      // check value
map.get(5);                     // get value by key
for (Map.Entry<Integer, String> entry :
          map.entrySet()) {
   // loop over keys and values
   entry.getKey();      // key
   entry.getValue();    // value
}
```

Map implementations

```
Map<Integer, String> hashMap =
        new HashMap<Integer, String>();

Map<Integer, String> treeMap =
        new TreeMap<Integer, String>();
```


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Hashtable

hashCode()

public int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java 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(java.lang.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.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

equals()

public boolean equals(Object obj)

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equalscomparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

equals()

```
public final class Pair<T, U> {
    private final T first;
    private final U second;
    public boolean equals(final Object o) {
        if (this == o) { return true; }
        if (!(o instanceof Pair)) { return false; }
        Pair<?, ?> pair = (Pair<?, ?>) o;
        if (first != null ? !first.equals(pair.first) :
            pair.first != null) { return false; }
        return second != null ? second.equals(pair.second) :
           pair.second == null;
    }
```

hashCode()

```
public final class Pair<T, U> {
    private final T first;
    private final U second;
    @Override
    public int hashCode() {
        int result = first != null ? first.hashCode() : 0;
        result = 31 * result +
            (second != null ? second.hashCode() : 0);
        return result;
    }
```

Token

```
public class Token implements IToken {
    public String getName() {
        return name;
    }
    public String getLexeme() {
       return lexeme;
    }
    public boolean equals(Object o) {
        //???
    }
    public int hashCode() {
        //???
    }
}
```

Mocks

REAL SYSTEM

Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST

Green = class in focus
Yellow = mocks for the unit test

Mockito

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.12.0</version>
    <scope>test</scope>
</dependency>
```

Create mock

```
import static org.mockito.Mockito.*;


IReader mockReader = mock(IReader.class);
```

Define mock behavior

```
// return value
when(mockReader.read())
        .thenReturn('a', 'b', 'c');

// throw exception
doThrow(Exception.class)
        .when(mockReader).close();
```

More complex behavior

```
doAnswer(new Answer() {
    public Object answer(InvocationOnMock invocationOnMock)
            throws Throwable {
        char[] chars =
            (char[]) invocationOnMock.getArguments()[0];
        chars[0] = 'a'; chars[1] = 'b'; chars[2] = 'c';
        return null;
    }
}).when(mockReader).readArray(any(char[].class));
```

Call mock methods

```
assertEquals('a', mockReader.read());
assertEquals('b', mockReader.read());
assertEquals('c', mockReader.read());

char[] chars = new char[3];
mockReader.readArray(chars);
assertArrayEquals(new char[] {'a', 'b', 'c'}, chars);

mockReader.close();
```

Verify mock calls

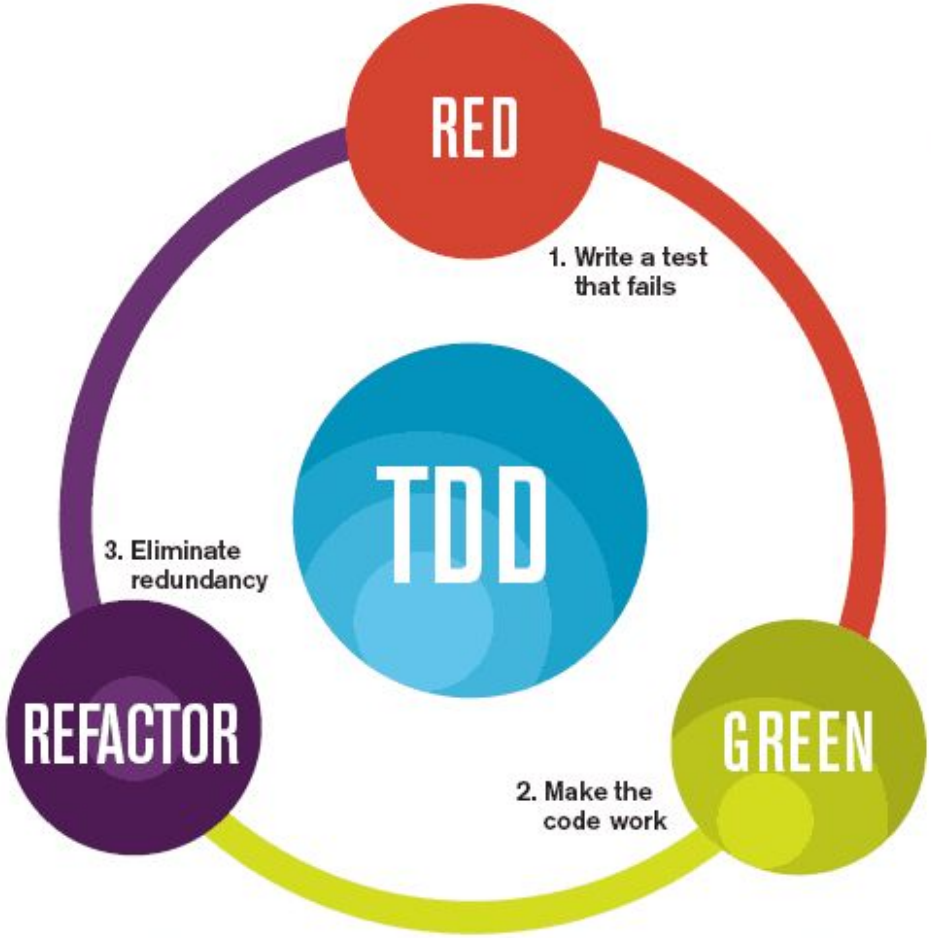
```
// read() was called 3 times
verify(mockReader, times(3)).read();

// readArray was called once with some char[]
verify(mockReader).readArray(any(char[].class));
```

Mock usage steps

```
// create mock
IReader mockReader = mock(IReader.class);
// define behavior (if necessary)
when(mockReader.read()).thenReturn('a');
// use it
assertEquals('a', mockReader.read());
// verify calls (if necessary)
verify(mockReader).read();
```

ALL CODE IS GUILTY
UNTIL PROVEN INNOCENT

CODESMACK



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

TDD buzzwords

keep it simple, stupid - KISS

you ain’t gonna need it - YAGNI

fake it till you make it

Checklist #1

1. Correct project structure, packages, .gitignore
2. IReader, IWriter interfaces
3. String IO implementation
4. Formatter uses IReader, IWriter
5. >3 JUnit tests of Formatter
6. <10 CheckStyle warnings
7. File IO Implementation, IClosable
8. `main()` uses File IO, opens and closes files
9. Maven builds the project
10. Runs successfully with `java -jar`

Checklist #2

11. Lexer, tests for Lexer
12. Formatter working with Lexer, tests
13. Lexer State Machine, tests
14. Formatter State Machine, tests

Homework

- Complete both checklists
- Need state machine
  - For Lexer
  - For Formatter

https://en.wikipedia.org/wiki/Finite-state_machine
https://en.wikipedia.org/wiki/Command_pattern
https://en.wikipedia.org/wiki/Abstract_factory_pattern