2IPC0 Programming Methods

From Small to Large Programs

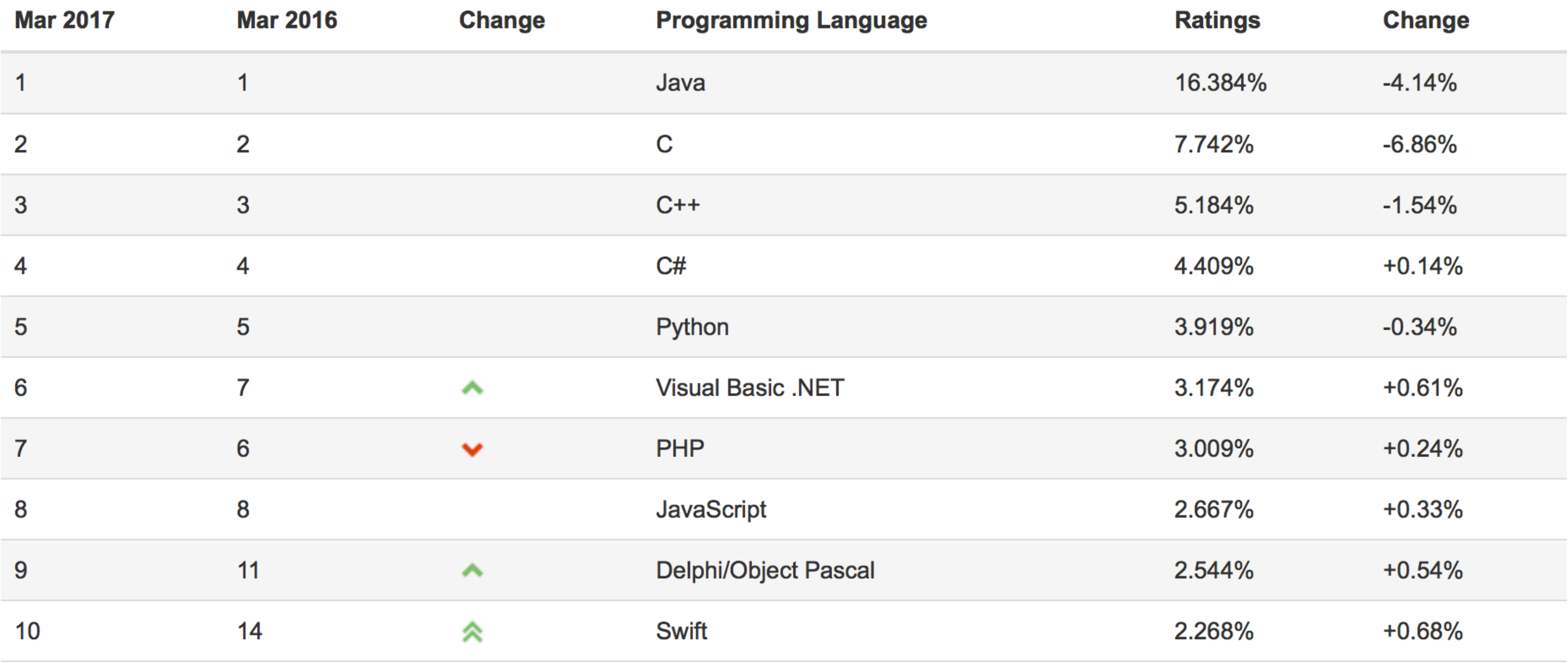
Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Professional Use of Java
- Performance profiling
- Looking back
- Looking ahead
- An anecdote about design methodology

Popularity of Java: 1st on `TIOBE.com` index



Mar 2017	Mar 2016	Change	Programming Language	Ratings	Change
1	1		Java	16.384%	-4.14%
2	2		C	7.742%	-6.86%
3	3		C++	5.184%	-1.54%
4	4		C#	4.409%	+0.14%
5	5		Python	3.919%	-0.34%
6	7	▲	Visual Basic .NET	3.174%	+0.61%
7	6	▼	PHP	3.009%	+0.24%
8	8		JavaScript	2.667%	+0.33%
9	11	▲	Delphi/Object Pascal	2.544%	+0.54%
10	14	▲▲	Swift	2.268%	+0.68%

Popularity of Java: 1st on TIOBE.com index

TIOBE is an Eindhoven-based company that monitors code quality

Professional Use of Java

- Language variants, advanced features, and add-ons
  - Versions: SE, EE, ME, Java Card, Android
  - Features: `docs.oracle.com/javase/8/docs/` Reflection, . . .
  - 3rd-Party Libraries, e.g. Guava at `github.com/google/guava`
- Tools
  - IDEs, Build Tools, Continuous Integration (CI), Testing
  - Code Analysis (PMD, DependencyFinder, JaCoCo, . . .)
  - Web Frameworks, Data Bases, Artifact Repositories
- Other languages on top of Java Virtual Machine (JVM)
  - Scala, Clojure, . . .

More Design Patterns

- Many more design patterns
- Also see: **`SourceMaking.com`**

Java 8 Enhancements

- Functional interfaces: `@FunctionalInterface`
- Lambda expressions (anonymous ‘pure’ functions)

  `(param_1, param_2, ..., param_N) -> body`
- Method references: `qualifier :: methodName`
- Streams: `java.util.stream`

Functional Interfaces

- Interface that defines *exactly one abstract method*
- Annotation `@FunctionalInterface` makes compiler check this
- Name of abstract method is irrelevant

  Only the parameter types and return type matter
- Example:

```
public interface FunctionalityB {
    void doB(String data);
}
```

Lambda Expressions

```
(param_1, param_2, ..., param_N) -> body
```

is an object with as type every functional interface `F` of the form

```
@FunctionalInterface
public interface F {
    public T m(param_1, param_2, ..., param_N);
}
```

where `T` is the type of `body`, or **void** when `body` is a statement

Lambda expression `(String data) -> System.out.println(data)` matches type `FunctionalityB`

Lambda Expressions

```
(param_1, param_2, ..., param_N) -> body
```

is (almost) equivalent to an anonymous local class instantiation:

```
    new F() {
        @Override
        public void m(param_1, param_2, ..., param_N) {
            body;  // when body is a statement
        }
    }
// or
    new F() {
        @Override
        public T m(param_1, param_2, ..., param_N) {
            return body;  // when body is an expression
        }
    }
```

Lambda Expression Examples

Invoke `FunctionalityA.doA()` with a lambda expression as callback:

```
    doA(n, (String data) -> System.out.println(data));
    doA(n, (data) -> System.out.println(data));
    doA(n, data -> System.out.println(data));
```

N.B. There are some subtle differences between lambda expressions and anonymous local classes

There are many predefined functional interfaces: `java.util.function`

`Consumer<T>`	with method	`**void** accept(T t)`
`Supplier<T>`	with method	`T get()`
`Function<T, R>`	with method	`R apply(T t)`
`Predicate<T>`	with method	`**boolean** test(T t)`

Using Standard Functional Interfaces

Interface `Iterable<T>` now has a default method `forEach(Consumer)`, whose definition is equivalent to:

```
    void forEach(Consumer<? super T> action) {
        for (T t : this) {
            action.accept(t);
        }
    }
```

For example, if we have a variable `Set<String> set`, then

```
    set.forEach( s -> System.out.print(s + " ") );
```

will print all elements in the set.

Method References

Any implemented method or constructor can serve as a ‘pure’ function via a **method reference**.

The general syntax is

```
    qualifier :: methodName
```

where the qualifier is either a class type or an expression evaluating to an object, and where the keyword **new** serves as ‘method name’ for a constructor.

Here is an example:

```
    set.forEach( System.out :: println );
```

Streams

- New package `java.util.stream` provides support for **streams**,
- with functional operators for sequential and parallel aggregation:

```java
int result = set.stream()
    .filter( s -> s.length() > 2 )
    .mapToInt( String :: length )
    .sum();
```

```java
Stream.Builder<String> builder = Stream.builder(); // step 1
FunctionalityA.doA(n, s -> builder.accept(s)); // step 2
// doA(n, builder), if FunctionalityB = Consumer<String>
Stream<String> stream = builder.build(); // step 3
stream.filter( s -> s.contains(pattern) )
        .map( s -> '"' + s + '"' )
        .forEachOrdered( System.out :: println );
```

Java 9

- The Java 9 release date has been pushed back to May 2017
- Java 9 will incorporate the more advanced Java Module System

Performance Profiling

- Main concerns in software development:
  - Functional correctness
  - Performance
  - Maintainability
  - . . . *the rest* . . . (incl. usability)
- Performance concerns (assuming you used a proper algorithm):
  - Execution time: where spent?
  - Memory usage: which objects, where created, for how long?
  - Threads: which threads in what states when?

Performance Profiling in NetBeans

- **Options/Preferences** > **Java** > **Profiler** > **General**

  Manage calibration data

- **Profile** > **Profile Main Project...**
  - Configure Session: Methods
  - **Settings**, Project classes
  - Run
  - Hot spots

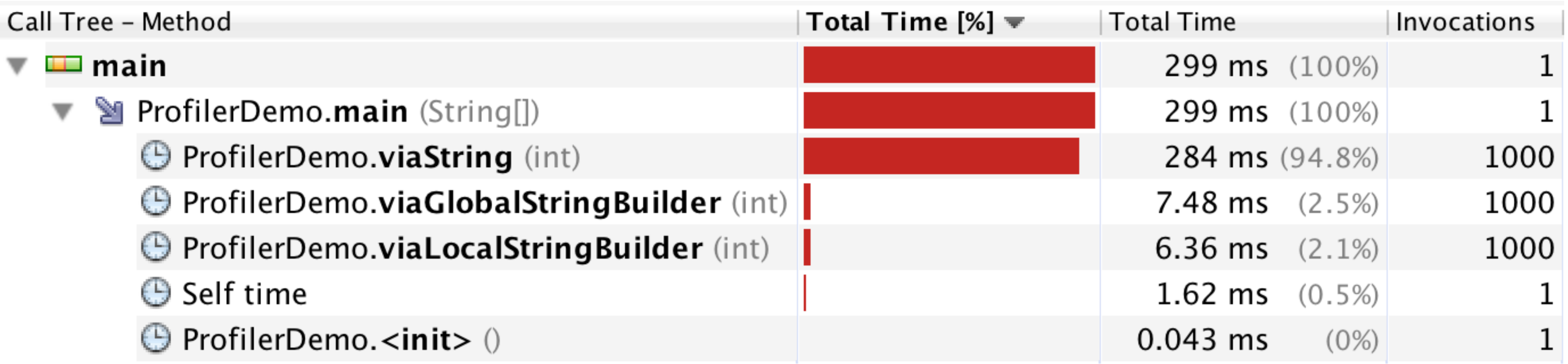
- Shows how often methods are called, and how long they run

- Provides insight in location of performance bottlenecks

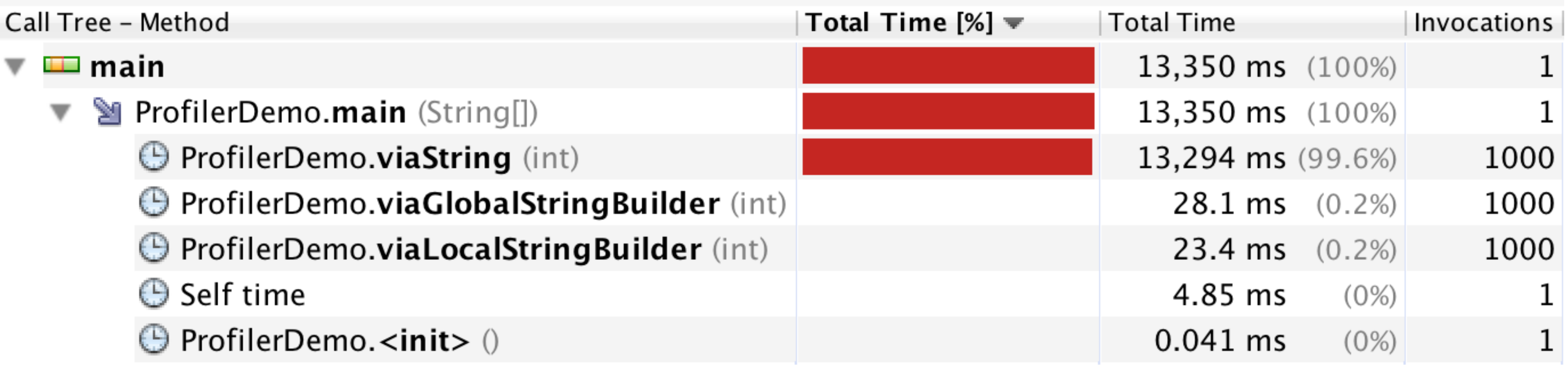
CPU Performance Profiling in NetBeans: Example

Compare `String` to `StringBuilder`: 10^3 vs 10^4 appends (1000 times)

Growth rate: `String`: quadratic; `StringBulder`: linear



Call Tree – Method	Total Time [%] ▼	Total Time	Invocations
▼ main		299 ms (100%)	1
▼ ProfilerDemo.**main** (String[])		299 ms (100%)	1
ProfilerDemo.**viaString** (int)		284 ms (94.8%)	1000
ProfilerDemo.**viaGlobalStringBuilder** (int)		7.48 ms (2.5%)	1000
ProfilerDemo.**viaLocalStringBuilder** (int)		6.36 ms (2.1%)	1000
Self time		1.62 ms (0.5%)	1
ProfilerDemo.**<init>** ()		0.043 ms (0%)	1



Call Tree – Method	Total Time [%] ▼	Total Time	Invocations
▼ main		13,350 ms (100%)	1
▼ ProfilerDemo.**main** (String[])		13,350 ms (100%)	1
ProfilerDemo.**viaString** (int)		13,294 ms (99.6%)	1000
ProfilerDemo.**viaGlobalStringBuilder** (int)		28.1 ms (0.2%)	1000
ProfilerDemo.**viaLocalStringBuilder** (int)		23.4 ms (0.2%)	1000
Self time		4.85 ms (0%)	1
ProfilerDemo.**<init>** ()		0.041 ms (0%)	1

Looking Back

– ○ – ○ – ○ – ○ – ○ –

Source Code Should Be Written with Utmost Care

- Source code is not only intended for the compiler.

  Source code should be readable and verifiable by other engineers.

- Adhere to a coding standard:

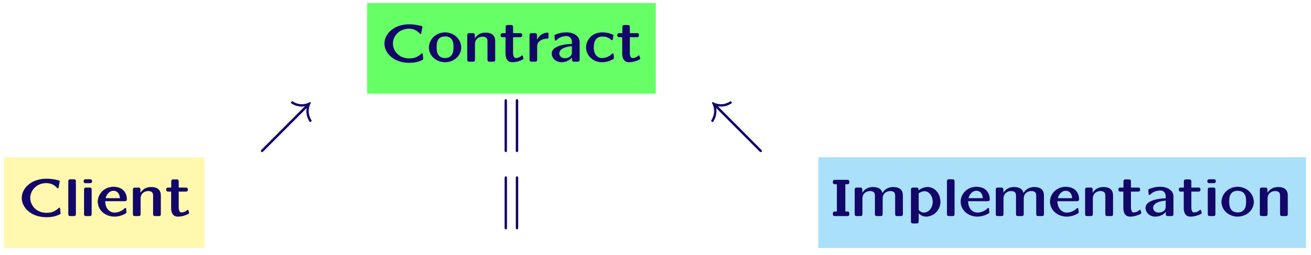
  Pay attention to layout, (javadoc) comments, naming, structure.

- This prevents mistakes, and eases finding and repairing of defects.

- View the extra time that this costs as an insurance fee.

Manage Complexity: Modularization

- Separation of Concerns: *Divide & Conquer* (and hence *Rule*)
- For each facility (function, type, iterator, package, . . .), separate



Contract

Client

Implementation

Client and implementation are based on the (same) contract.

Client and implementation are not directly ‘coupled’.

Dependency Inversion Principle (DIP): depend on abstractions

- Divide & Conquer serves many purposes, including maintainability.

Procedural Abstraction

- Abstract from data operated on (through parameters)
- Abstract from realization of operation (when viewed from usage)
- Abstract from context of use (when viewed from implementation)
- Guidelines for functional decomposition

Errors and Exceptions

IEEE terminology: failure, defect (fault, bug), mistake, error

For non-private methods:

- the precondition should be as weak as possible: unless unacceptable for performance reasons.
- the contract specifies relevant exceptions and their conditions: `@pre P` and `@throw E if ! P`

You are encouraged to check preconditions, e.g. via `assert`, also in **`private`** methods.

In Java, runtime assertion checking is disabled by default! Use `-ea`

Data Abstraction

Abstract Data Type = set of (abstract) values and corresponding operations: construct, destroy, query, modify

In Java

Specification: **`class`** name, **`public`** method headers and contracts,
public invariant
Optionally: public constant names

Implementation: **`private`** instance variables, private (rep) invariant,
abstraction function, public method bodies
Optionally: public constant values, private methods

N.B. Variable of **`class`** type is a reference: *aliasing*!

Java Built-in Protections for Modularization

- Functions (class methods): local variables
- Data types (classes): instance variables, methods

	Access Level			
Modifier	**Class**	**Package**	**Subclass***	**World**
`private`	*Yes*	No	No	No
no modifier	*Yes*	*Yes*	No	No
`protected`	*Yes*	*Yes*	*Yes*	No
`public`	*Yes*	*Yes*	*Yes*	*Yes*

*Outside package

Step-by-step (Test-driven) ADT Development Plan

1. Gather and analyze requirements.
2. Choose requirement to develop next.
3. Specify class & methods informally: javadoc summary sentences.
4. Specify formally: model with invariants, signatures, and contracts.
   Class w/o implementation: no data rep, empty method bodies.
5. Create a corresponding unit test class.
6. Implement rigorous tests.
7. Choose data representation and implement class methods.
8. Test the implementation.
9. Refactor and retest.

Iteration Abstraction

- Problem: Visit each item of a collection exactly once
- Abstract from type of collection, type of items, how to iterate
- An *iterator object* maintains the state of the iteration.
- In general, an iterator object implements `Iterator<T>` providing methods **`boolean`** `hasNext()`, `T next()`, and optionally `remove()`.
- To use enhanced **`for`** statement, collection implements `Iterable<T>`, i.e., provides a method `iterator()` returning an `Iterator<T>`.
- **`for`** `(` *Type* *Identifier* `:` *Expression* `)` *Statement*

SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, see Lecture 10)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, see Lecture 13)
- **D**ependency Inversion Principle (DIP, see Lecture 8)

Goal: to manage change

Also: Don't Repeat Yourself (DRY)

Design Patterns

A Design Pattern is an *outline* of a *general* solution to a design problem, *resuable* in *multiple, diverse, specific* contexts.

It is *not* a solution in the form of source code or a *library* that can be reused "as is".

Taxonomy of Design Patterns

- Creational patterns

  Factory Method, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

  “`SwingWorker`”

Reflection on Method

- Adhering to a method (systematic way of working) has a price.
- It may cost extra development time, code, and execution time.
- Without method, complexity quickly becomes unmanageable, which costs even more.
- Method makes project and product quality better controllable.
- Method reduces time for getting it to work: defect localization and correction, change and reuse.

Looking Ahead

– ○ – ○ – ○ – ○ – ○ –

Programming in the Future

- Techniques to develop algorithms: stepwise refinement

  Correctness by construction

- JML: Java Modeling Language

  Tools to support formally verified software development

- Domain-Specific Languages (DSLs), language technology

  Meta-models, models

  - Text-to-Model: lexical analysis, parsing
  - Model-to-Model: transformation
  - Model-to-Text: formatting, source code generation

Brilliance

Michael Jackson

Software Requirements & Specifications:
a lexicon of practice, principles and prejudices

Addison-Wesley, 1995

ISBN 0–201–87712–0

www.win.tue.nl/˜wstomv/quotes/software-requirements-specifications.html

Brilliance

Some years ago I spent a week giving an in-house program design course at a manufacturing company in the mid-west of the United States.

On the Friday afternoon it was all over.

The DP Manager, who had arranged the course and was paying for it out of his budget, asked me into his office.

Brilliance

‘What do you think?’ he asked.

He was asking me to tell him my impressions of his operation and his staff.

‘Pretty good,’ I said. ‘You’ve got some good people there.’

Program design courses are hard work; I was very tired; and staff evaluation consultancy is charged extra.

Anyway, I knew he really wanted to tell me his own thoughts.

Brilliance

‘What did you think of Fred?’ he asked. ‘We all think Fred is brilliant.’

‘He’s very clever,’ I said. ‘He’s not very enthusiastic about methods, but he knows a lot about programming.’

‘Yes,’ said the DP Manager. He swivelled round in his chair to face a huge flowchart stuck to the wall: about five large sheets of line printer paper, maybe two hundred symbols, hundreds of connecting lines.

Brilliance

‘Fred did that. It’s the build-up of gross pay for our weekly payroll. No one else except Fred understands it.’

His voice dropped to a reverent hush.

‘Fred tells me that he’s not sure he understands it himself.’

Brilliance

‘Terrific,’ I mumbled respectfully. I got the picture clearly.

Fred as Frankenstein, Fred the brilliant creator of the uncontrollable monster flowchart.

That matched my own impression of Fred very well.

‘But what about Jane?’ I said. ‘I thought Jane was very good. She picked up the program design ideas very fast.’

Brilliance

‘Yes,’ said the DP Manager. ‘Jane came to us with a great reputation. We thought she was going to be as brilliant as Fred. But she hasn’t really proved herself yet.

We’ve given her a few problems that we thought were going to be really tough, but when she finished it turned out they weren’t really difficult at all. Most of them turned out pretty simple.

She hasn’t really proved herself yet — if you see what I mean?’

I saw what he meant.

(End of quote)