Java Programming 2
Lecture #16

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
13 November 2019

"Contemplating orange juice" by Hannah Webb
https://flic.kr/p/bkqXSy

Outline

Immutable objects

Higher-level concurrency with **java.util.concurrent**

Annotations

Immutable objects

“Immutable”?

immutable

[ih-**myoo**-*tuh*-*buh* l]

Spell Syllables

Examples Word Origin

adjective

1. not mutable; unchangeable; changeless.

Immutability in Java

Immutable: internal state cannot change after it is constructed

Examples:

- `String`
- Wrapper classes: `Integer`, `Long`, `Character`, etc.

Advantages of immutability

Immutable objects can be safely shared between data structures or threads

Can save memory:

- Two Strings with the same value are effectively identical …
- … so they can be mapped onto the same object at runtime

Ideal for lookup keys in “dictionary” structures

- Value will never change, so lookup is reliable

What does that mean in practice?

You always need to create new objects for new contents

- No possibility to change state, e.g., `setColour (RED)`

But isn't it more expensive to create new objects all the time instead of reusing them?

- Yes (very, very slightly) …
- … but there are also efficiencies:
  - *Decreased garbage collection overhead*
  - *No need for code to protect objects from corruption*

String operations

Lots of constructors and static initialisers …

Lots of getters …

`charAt`, `indexOf`, `length`

Lots of methods to check the state

`contains`, `compareTo`, `equalsIgnoreCase`, `startsWith`

Other methods all **return a new string** – do not modify current string

`concat`, `toLowerCase`, `replace`, `trim`

What does this mean?

```
public void doStuff() {
    String s = “Hello world”;
    // Doesn’t actually change s at all
    s.toUpperCase();
    // s2 now contains “HELLO WORLD”
    String s2 = s.toUpperCase();
}
```

Creating an immutable class

Instance fields:

- Must be `private` and `final`
- Must have getters but no setters

Constructor:

- Must set complete internal state of object

Methods:

- Don't allow overriding
  - *Easy: declare class `final`*
  - *Fancy: make constructor `private` and use static factory methods to create instances*

Creating an immutable class (2)

If instance fields can be mutable objects, don’t let them be changed

- Don’t provide methods to modify them
- Don’t return the mutable objects directly from getters; return copies instead

Immutable class example

BEFORE

```
public class Person {

    private List<String> names;

    public Person(String[] names) {
        this.names =
                Arrays.asList(names);
    }

    public List<String> getNames() {
        return names;
    }

}
```

AFTER

```
public final class Person {

    private final List<String> names;


    public Person(String[] names) {

        this.names = new
ArrayList<>(Arrays.asList(names));

    }


    public List<String> getNames() {

        return new ArrayList<>(names);

    }

}
```

Higher-level concurrency

Lock objects

Atomic variables

Concurrent collections

Synchronized methods: reminder

Additional keyword: **`synchronized`**

Add to method header

Ensures that:

- Two calls to **synchronized** methods **on the same object** cannot interleave
- When a synchronized method exits, it **happens-before** any other **synchronized** method calls **on the same object**

Constructors cannot be synchronized

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Lock objects

Generalised version of **synchronized** code (simple intrinsic lock)

Basic interface: **java.util.concurrent.locks.Lock**

- Work like intrinsic locks
- Only one thread can own a Lock object at a time
- *Also support wait()/notify()*

Big advantage: allow code to back out of an attempt to acquire a lock

- tryLock() – backs out if lock is not available or if timeout expires
- lockInterruptibly() – backs out if another thread sends interrupt before lock is acquired

Sample Lock code (Java tutorial)

https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html

Atomic variables

Package **java.util.concurrent.atomic**

Defines classes that support **atomic operations** on single variables

- All classes have get() / set() methods that impose **happens-before** – set happens before get
- Atomic compareAndSet() method
- Simple arithmetic methods that apply to integer atomic variables
  - *decrementAndGet(), addAndGet(), getAndAdd() …*

Counters revisited

```
class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Other useful Java libraries related to concurrent programming

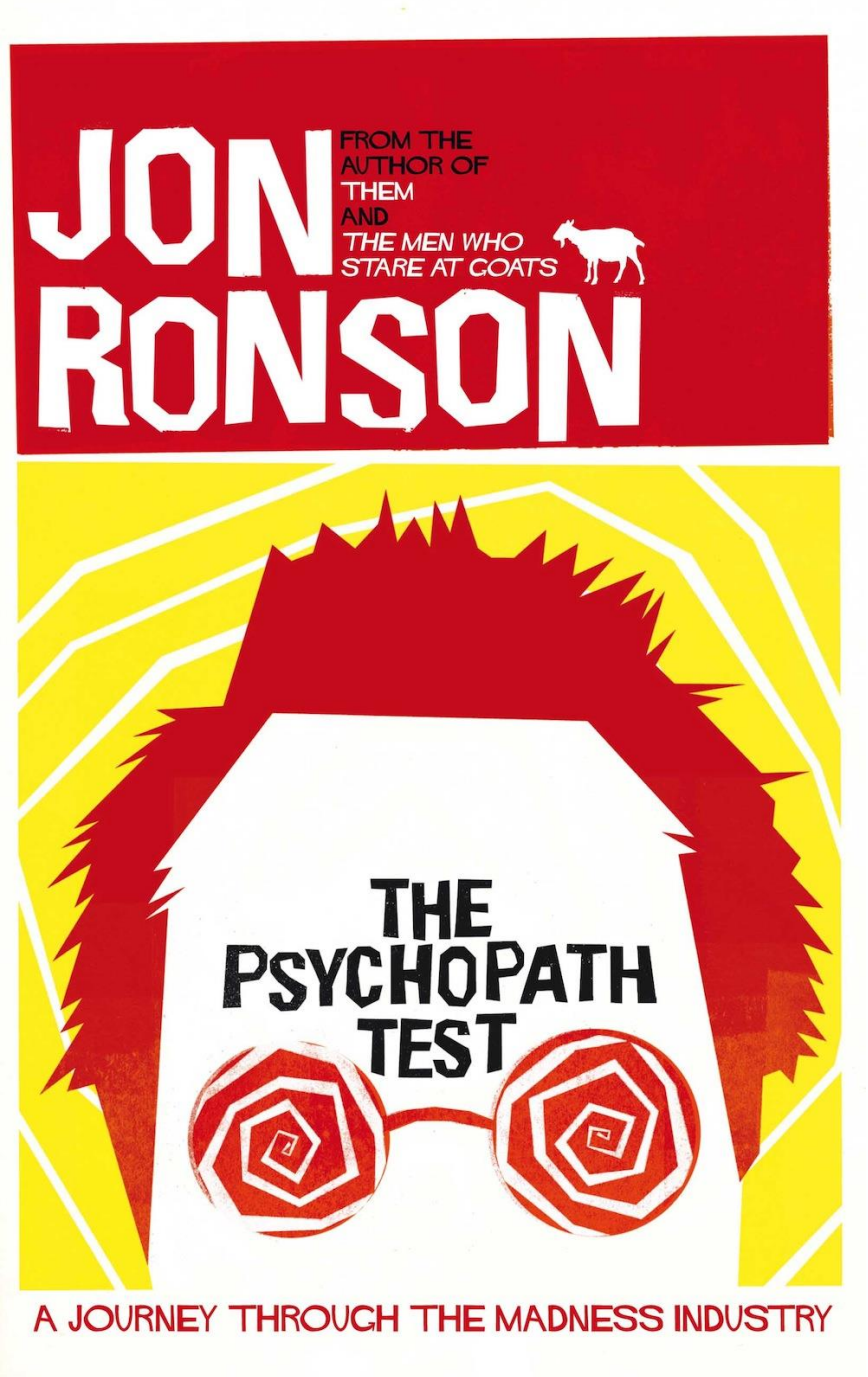
java.util.concurrent: Concurrent collections

- **BlockingQueue**: a first-in/first-out structure that blocks when you attempt to add to a full queue of remove from an empty queue
- **ConcurrentMap:** defines atomic operations on maps (e.g., **putIfAbsent**)
- **ConcurrentNavigableMap:** supports approximate matches

Streams support **parallelStream()** operator – processes Stream objects in parallel (Java runtime decides how to divide things up)

- Note that any methods called in the context of a parallel stream must be thread-safe (locks, atomic, etc)

Annotations



"Always code as if the [person] who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability."

John F. Woods

Annotations

A form of **metadata** – provide data about a program that is not part of the program itself

Have no direct effect on the operation of the code they annotate

Uses

- **Information for the compiler:** detect errors, suppress warnings
- **Compile-time processing:** use annotations to generate code/XML/etc
- **Runtime processing:** some annotations are available

Format of an annotation

Start with an @ sign

```
@Override
public String toString() { … }
```

Refer to the element following them

Must appear **outside** comments

May have arguments inside parens – if no arguments, parens can be omitted

```
@SuppressWarnings("unchecked")
void myMethod() { … }
```

Annotation locations

Generally, applied to declarations (classes, fields, methods, etc.)

Conventionally, each annotation appears on its own line

As of Java 8, annotations can also be applied to the use of types

- Ensures stronger type checking
- Not built into Java itself, but downloadable packages exist
  - *E.g.,* http://types.cs.washington.edu/checker-framework/

```
@NonNull String str; // Won't work without external package
```

Useful predefined annotations

`@Deprecated`

Marks code as “deprecated” – i.e., still included but use is discouraged

`@Override`

Indicates that the labelled method must override a superclass method

`@SuppressWarnings`

Disables particular compiler warnings

All are defined in java.lang (e.g., `java.lang.Override`)

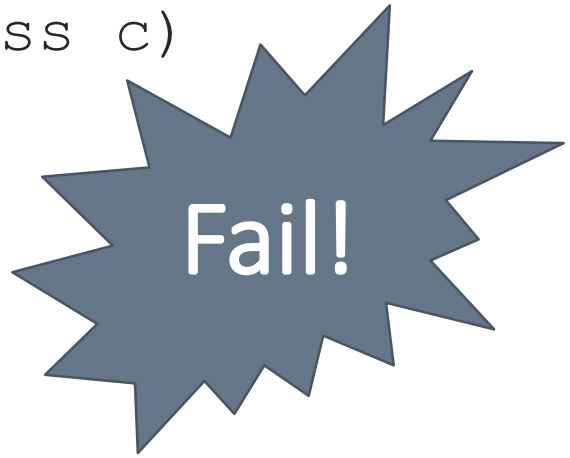
Use of @Override

"Indicates that a method declaration is intended to override a method declaration in a supertype" (Javadoc)

Compiler produces an error message unless this is true

Automatically added by Eclipse whenever you override/implement methods

```
public class MyClass {
   @Override
   public boolean equals
      (MyClass c)
   {
      …
   }
}
```



Use of @SuppressWarnings

Tells compiler to suppress warnings that it would otherwise generate

Argument indicates category:

- **deprecation**: disable warning on use of deprecated method
- **unchecked**: disable warning on use of non-generic code

Full set of Eclipse warnings:

http://help.eclipse.org/mars/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-suppress_warnings.htm

Should be attached to innermost element where they apply

- Do not disable warnings on a whole class if they are needed on one method!

@SuppressWarnings example

```
@SuppressWarnings("unchecked")
public void doSomethingOldFashioned() {
    ArrayList list = new ArrayList();
    list.add ("One");
    list.add (2);
    list.add (3.0);
}
```

Adding annotations in Eclipse

Eclipse automatically adds @Override annotations to any auto-generated methods where it is relevant

- Implementing an interface
- Subclassing an abstract class
- Explicitly choosing “override/implement methods”

It often proposes a “quick fix” to suppress warnings when they occur

- Only do this if you are **REALLY REALLY SURE** the warning is not relevant!