COMP6700/2140 Code as Data

Alexei B Khorev

Research School of Computer Science, ANU

March 2017

Topics

1. What does treating code as data mean?
2. Why is treating code as data useful?
3. How to pass code for deferred execution?
   - Inner Classes
   - λ-Expressions
4. *Account* example
   - Pre-lambda's way (Java 1–7)
   - Lambda's way (Java 8)
5. λ's way 1: When the name does not mater, and everything else is inferred
6. λ-expression examples
   - Callbacks
   - Threads
   - Comparators
7. λ's way 2: `invokedynamic` — no byte code proliferation
8. Reasons for λ-s

Fleeting Nature of Software Requirements

Software is... soft

- Easy to change
- Easy to mandate change (new/modified requirements)
- Cost of supporting changeability is **not so low**
  - need to change/extend code
  - need to test it
- Solution? Use of language/platform with which the cost of change is lower

Bank Account Example

In a traditional banking example of an *Account* class, there are "common-sense" class members:

```
class Account {

   private double balance = 100.5;
   private final double interests;

   Account(double bal, double intr) {
       ... ... ...
   }

   double getBalance() { ... }
   private void setBalance(double bal);
   void deposit(double x) { ... }
   void withdraw(double x) throws InsufficientFundsException { ... }
   void printBalance() { ... }
   void accrueInterests() { ... }
   ...
}
```

Processing accounts

There are hundred of thousands (or millions) of such accounts in the database. They regularly need processing — update their data (balance, payments, interests etc).

```
// somewhere in the client code
List<Account> accounts = /* get them from the database or smth */;
for (Account a: accounts) {
   a.accrueInterests(); // changes the balance value in every account
}
```

Now comes the first change: add *check account aspect* to allow bank checks be issued against the account balance; the checks will be settled at regular intervals as a part of existing account processing practice:

```
// somewhere else in the client code
List<Account> accounts = ...;
for (Account a: accounts) {
   if (a.isCheckAccount())
      a.processChecks(); // changes the balance value in every account
}
```

State (value) parameterisation

Ability to pass value to a method allows to express *state parameterisation* — depending on the value of object (its state) computation will have different effect (different values will be returned):

$$\texttt{state} \Longrightarrow \texttt{value}$$

Good (that's why we have methods with formal parameters!), but not flexible enough if we need to change computation, but the cause of this change is not readily expressed through the change of state. For example, how to express the rule (criterion) which is used to select accounts for the purpose of performing a certain task:

```
accountsOfInterest = getAccounts(accounts, <aspect-specific>);
process(accountsOfInterest, <task>);
```

- How to express <aspect-specific> "parameter"?
- How to express <task> "parameter"?

Aspects and Tasks

Aspect: select check accounts, Task: settle the due checks

```
// body of getAccounts(accounts, <check-accounts>)
List<Account> getCheckAccounts(accounts) {
   List<Account> checkAccounts = new ArrayList<>();
   for (Account a: accounts) {
       if (a.isCheckAccount())
           checkAccounts.add(a);
   }
return checkAccounts;
}
// body of process(checkAccounts, <settle-the-checks>);
for (Account a: checkAccounts)
  a.processChecks();
```

The two methods can be replaced by one, but often is useful to keep them separate as the `checkAccounts` may be used elsewhere.

Aspects and Tasks

Aspect: select term deposits which mature `today`, Task: accrue due interests

```
// body of getTermDeposits(accounts, today, <mature-today-deposits>)
List<Account> getTermDeposits(accounts, today) {
   List<Account> matureDeposits = new ArrayList<>();
   for (Account a: accounts) {
       if (a.isTermDeposit() && a.maturityDate().equals(today))
           matureDeposits.add(a);
   }
   return termDeposits;
}
// body of process(termDeposits, <accrue-interests>);
for (Account a: checkAccounts)
   a.accrueInterests();
```

More similar processing tasks can be easily formulated.

Treating code as data

Can we reduce amount of code repetition? Can we replace multiple methods (for selection and processing in different aspect-tasks) by a fewer methods (one?) in which aspects and tasks are represented by parameters?

The aspects and tasks are expressible as code (boolean expressions `a.isCheckAccount()` and `a.isTermDeposit()`; different methods `accrueInterests()` and `processChecks()` with similar signature `account -> void`). Can we represent these codes by a formal parameter?

We have methods to express code, but can we pass methods to other methods as parameters? And can we define methods which themselves return methods, not values?

- Suppose `aspectMethod` will represent either `getCheckAccounts`, or `getTermDeposits` (and other) "method-values", and the `processMethod` will represent either `processChecks` or `accrueInterests` (and other) "method-values"
- Is it possible to define a "super"-method to perform all the processing tasks at once when it's invoked with corresponding "actual method values"?

```
process(accounts, aspectMethod, processMethod);
```

Yes, it is! But not in such naive form — method name is **not** a legal expression, and the invocation expression, like `getCheckAccounts()`, is **not the code**, it is **the value** which the corresponding *actual* method returns.

Passing code as Data

Everything is an object in Java (except for primitive, but let's forget about them). Objects have code inside them (defined in object's class). If we have an interface, say *Action*, with a *single* method in it, say `Action.doIt()`, then having a reference `action` of the type *Action* we can pass it to a method when the latter is defined (formal parameter `action`), and when invoking the method, we pass a real object `action` to it. The only problem is to implement the *Action* interface and create an actual object parameter for passing, *before or during* the method invocation.

```
interface Action<T> {
   void doIt(T t);
}
void aMethod(T t, Action<T> action) { // can return a type as well
   action.doIt(t); // t can also come for outside (closure)
})
... ...
class ActionOne implements Action<Account> { // Account is a value for T
   void doIt(Account a) { ... };
}
Action<Account> actionOne = new ActionOne();
aMethod(t0, actionOne); // the code value is passed and executed
```

The "code value" is created when the interface *Action* is implemented *before* an object of this type is created and passed to an invoked method. Can it be made more value-like passing, when the actual parameter is passed directly?

Anonymous Inner-classes

Java always (since Java 1.1 of 1997) could represent code by a parameter, and pass this code to a duly defined method. But this was rather awkward. It required inner-classes, and often was done without given them a name, i.e. via *anonymous inner-classes*.

```
V v = ...;
aMethod(new Action<Account> {
   void doIt(Account a) {
     // if v is used, then this is a closure
     ...
   }
});
```

What we do inside the implementing body of `doIt()` is up to us (provided that the type *Account* allows it).

This was a standard way to define

- callbacks in GUI programs
- thread execution tasks
- aspects in implementation of standard algorithms like sorting

Examples of the old way

- *Callbacks* in GUI programs (define the action in response to a signal which a widget registers, *eg*, a button click):

```
scene.onKeyPressedProperty().set(new EventHandler<KeyEvent>() {
    @Override
    public void handle(KeyEvent ke) {
       if (ke.getCode() == KeyCode.R)
          rect.getTransforms().add(new Rotate(22.5, 410, 200));
    }
});
```

- *Thread execution code* (by implementing the *Runnable* interface)

```
new Thread(new Runnable {
   public void run() {
      for (int i = 0; i < 1000; i++)
         doWork();
   }
).start();
```

- Perform *sorting operations* (and others) on collections when the sorting criterion is passed as an object of *Comparator* interface.

```
Collections.sort(books, new Comparator<Book>() {
    public int compare(Book b1, Book b2){
        return b1.toString().length() - b2.toString().length();
    }
});
```

Accounts of the old way

```
interface Operation  {
   void operation(Account7 a);
}

static Operation interests = new Operation() {
   public void operation(Account7 a) {
      a.accrueInterests();
   }
};

static Operation checks = new Operation() {
   public void operation(Account7 a) {
      a.processChecks();
   }
};

Map<String, Operation> bankProcedures = new HashMap<>();
bankProcedures.put("interests", interests);
bankProcedures.put("checks", checks);

// if today is the day to pay interests
process(accounts, bankProcedures.get("interests"));
```

The new way: λ-expressions

Since the interface used in passing code tricks has only one method (to be implemented when a "code-value" is created), its name is not important. The rest of method's signature, parameters, return values and exceptions, can be *inferred* by analysing the expression(s) of the method body (including the `return` statement). Such one-method interfaces (not counting the recently allowed default methods, of course) are now referred to as *functional*.

These ideas allowed to replace an object of anonymous inner class by a new type of *lambda* (λ) *expression* — which is *semantically* (but not as the byte code!) *equivalent* to that object:

```
aMethod(Account a -> doIt(a)); // with explicit type of a, or
aMethod(a -> doIt(a));         // without it (compiler will infer)
```

λ expression syntax

```
(parameters) -> expression // for simple codes

(parameters) -> { statements; } // for multi-statement codes
```

When the compiler sees a lambda, it creates an instance of a functional interface.

Lambda examples

(from M. Naftalin's Lambda FAQ on-line λ-tutorial)

```
1. (int x, int y) -> x + y  // takes two integers, returns their sum
2. (x, y) -> x - y          // takes two numbers, returns their difference
3. () -> 42                 // takes no values, returns 42 (parentheses needed)
// (next) takes a string, prints its value to stdout, returns nothing
4. (String s) -> System.out.println(s)
5. x -> 2 * x               // takes a number, returns its double value
// (next is a block)  takes a collection, clears it, returns its previous size
6. c -> { int s = c.size(); c.clear(); return s; }
```

Lambda Syntax Rules

- Parameter types are declared or omitted (they will be inferred, one cannot mix explicit and implicit use).
- `return` statement (if used) should be present in every branch; for a *block body* — the rules for using the `return` keyword are the same as for an ordinary method body.
- For a single expression the use of `return` is optional (inferred).
- Expression may evaluate to a value (value compatible), or may not (void compatible).
- Despite the appearance, determining the type of objects is **not** postponed till run-time — Java is still statically typed.

Examples of the new way

How much shorter, yet more expressive the old statements look when rewritten with λ-expressions! Only code that matters (the "code") is present, all the ceremony is dropped.

- **Callbacks**:

```
scene.onKeyPressedProperty().set(ke ->
   { if (ke.getCode() == KeyCode.R)
        rect.getTransforms().add(new Rotate(22.5, 410, 200));
   }
);
```

- **Thread code** (by implementing the *Runnable* interface);

```
new Thread(() -> {
   for (int i = 0; i < 1000; i++)
      doWork();
   }
).start();
```

- **Sorting** operations using the *Comparator* object:

```
Collections.sort(books,
   (Book b1, Book b2) -> b1.toString().length() - b2.toString().length());
```

Accounts of the new way

```
interface Operation  {          // no need for Operation
   void accept(Account8 a);     // we can use API's Consumer
}

static Operation interests = a -> a.accrueInterests(); // simpler,
static Operation checks = a -> a.processChecks();      // alright, BUT
// no need for interests, no need for checks
Map<String, Operation> bankProcedures = new HashMap<>();    // only these
bankProcedures.put("interests", a -> a.accrueInterests());  // lines are
bankProcedures.put("checks", a -> a.processChecks());       // retained

// if today is the day to add interests
process(accounts, code.get("interests"));
```

Sweet... but there is MORE:

Indy — `invokedynamic` code in JVM (new, since Java SE 7)

- Compile the code in pre-Java 8, and for every anonymous inner class there will be a bytecode class (demo with Java7/Account7)
- Compile the code in Java 8, and no additional bytecode classes will be produced for any lambda expressions created in the code (demo with Java8/Account8)

Why λ's were introduced?

- To facilitate adoption of *functional programming style* into *Java*. Many concurrent and reactive (event-driven) problems have better solutions when dealt with in functional way;
- λ's allow to defer the execution of the code till later (better solution than using methods and inner-classes);
- allow to capture the context when executed (closures);
- allow lazy programming (conditional execution, more memory efficient when containers are processed with streams);
- allow composition of various operations applied on stream objects;
- allow to split multiple executions (eg, when operating on lists) into subtasks and performs them concurrently;
- last-not-least, use of λ's instead of (anonymous) inner classes (predominantly in call-backs) puts a break on byte-code classes proliferation, example — BuilderAtWork.java. The new JVM's instruction called `invokedynamic` is used to execute the λ-expression code in the bytecode, which does not involve new classes created and loaded, and has performance improvements.

Stylistic advantages

- Anonymous (less info)
- Functions (more flexible)
- Passed around (as argument to a method or stored in a variable)
- Concise

Where to look for this topic in the textbook?

- Hortsmann's *Core Java for the Impatient*, Ch. 3.4, 3.6–3.9
- Oracle's Java Tutorial on Lambda Expressions
- Maurice Naftalin Lambda FAQ (very useful)