COMP6700/2140 Methods

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

9 March 2017

Methods and `return` Keyword

A method is a separate piece of code which can be called from another piece of code to carry out some operation. In Java, each method must be defined within the body of a class and each class can have several methods. A method accepts zero or more *parameters* (also known as arguments) of certain types, which are passed *by value* from the caller, and returns a *single value* (the result). The return value can be a value of a primitive type, or an object, including an array or a container.

The syntax of method definition:

```
[@annotation-type] [modifier]* return-type method-name(argument list) {
   ... method-body (block statement) ....
   [return expr;]* // where expr instanceof return-type
}
```

For a method which returns an object, the type of the object must be already known (either as imported from an API, or defined somewhere in the program. If a method is declared with the keyword `void` in place of return type, it does not return any value — its purpose is to create *side effects*. If a method *does* return a value, its body must contain one or more *return statements*: `return some_expression`.

Unlike in some languages, in Java **one cannot define a method inside a body of another method** (often the effects of such "internal" functions can be realised by including local inner classes or λ-expressions).

Method Modifiers

Modifiers (just like modifiers of variables) specify the relationship of a method with other parts of the program.

- visibility
  - `private`: visible only within this class
  - no modifier: visible to all classes in the current package
  - `protected`: visible to classes in the current package and subclasses of this class
  - `public`: visible to all classes everywhere
- `final`: this method cannot be overridden by subclasses
- `static`: associated with the class itself rather than objects of the class

Method Parameters

The caller passes *parameters* for use inside the method body.

The argument list is a comma-separated list of parameter declarations surrounded by a pair of matching parentheses:

```
method-name(type1 name1, type2 name2, type3 name3, ...)
```

```
/** Calculate the squares of the numbers from 1 to 10 */
public class SquareInt {
   public static void main(String[] args) {
      for (int i = 1; i <= 10; i++) {
         System.out.printf("number = %d, square = %d%n", i, square(i));
      }
   }

   static int square(int x) {
      return x*x;
   }
}
```

Pass by Value

In Java, all parameters are passed by value.

This means that any changes a method makes to its parameters are not visible to the caller.

```
void badSquare(int x) {
   x = x*x;
}

int b = 2;
square(b);
System.out.println(b); // prints 2, *not* 4
```

Pass by Value (2)

However, for parameters of reference type, it is only the *reference* that is passed by value.

Changes to the reference are not visible to the caller:

```
void badRewrite(String s) {
    s = "good";
}

String b = "bad";
badRewrite(b);
System.out.println(b); // prints "bad", *not* "good"
```

Pass by Value (3)

… but changes to fields of the object that is referred to *are* visible to the caller.

```
class MyType {
    String value;
    public MyType(String v) {
        this.value = v;
    }
}

void goodRewrite(MyType m) {
    m.value = "good";
}

MyType mine = new MyType("bad");
goodRewrite(mine);
System.out.println(mine.value); // prints "good"
```

The Stack and the Heap

The JVM has a *stack architecture*: whenever a method is invoked, it creates a *stack frame* in memory which has its own namespace to hold all the local variables.

This frame is placed on top of the *call stack* — a dynamic data structure in the JVM's memory which follows the *LIFO* (last-in, first-out) rule.

A frame's local variables consist of:

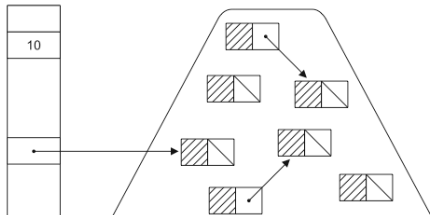
- a reference to `this` (the current object)
- method parameters
- all locally defined variables
- method return value (if any)

When a method terminates and control is returned to the caller, the frame is removed from the stack and all local variables disappear. Do not look for them outside the method — they've already gone, they only existed during the method execution.

If a local variable is a reference, its object is stored on the heap — another (larger) section in memory. The heap contains all objects created with `new` operator, literal strings and primitive fields of objects. Heap memory is managed by a *Garbage Collector*.

Stack and Heap

An array of local variables in a stack frame (left), and the heap of objects waiting for the GC.



(Courtesy of B. J. Evans and M. Verburg, *The Well-Grounded Java Developer*)

Improving the performance of the GC (make it faster, less disruptive to program execution) is an active area of research.

A Program with Nested Method Calls

```
class StackGames  {
   public static void main(String[] args) {
       System.out.println("Starting main, calling m1...");
       m1();
       System.out.println("Exiting m1, finishing main and the program");
   }
   private static void m1() {
       System.out.println("Starting m1, declaring & initializing x1");
       int x1 = 10;
       System.out.printf("local x1 = %d inside m1%n", x1);
       System.out.println("calling m2 and passing it x1...");
       m2(x1);
       System.out.println("Exited m2, finishing m1");
   }
   private static void m2(int i) {
       System.out.println("Starting m2, declaring & initializing x2");
       int x2 = 100;
       System.out.printf("local x2 = %d inside m2%n", x2);
       System.out.printf("x1 * x2 = %d%n", i * x2);
       System.out.println("Exiting m2, returning control to m1");
   }
}
```

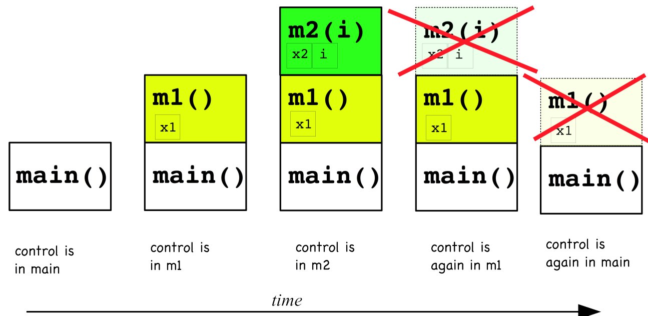
Nested Method Call Execution

Compile and run

```
abx% javac StackGames.java
abx% java StackGames
Starting main, calling m1...                // frame 1 is added to stack
Starting m1, declaring & initializing x1 // frame 2 is added to stack
local x1 = 10 inside m1
calling m2 and passing it x1...          // frame 3 is added to stack
Starting m2, declaring & initializing x2
local x2 = 100 inside m2
x1 * x2 = 1000
Exiting m2, returning control to m1
Exited m2, finishing m1                  // frame 3 is removed from stack
Exiting m1, finishing main               // frame 2 is removed from stack
```

When the program `StackGames` executes, the call stack first grows, then shrinks. The first frame `main` is removed last by the JVM stack manager.

Stack Dynamics



In an application which runs several threads, each one operates its own method call stack. A stack maximal size is usually fixed (by the maximal memory not by the depth), but can be dynamic. When a thread tries to add another frame and exceeds the limit, *StackOverflowError* is thrown. If a dynamic stack tries to allocate a new frame and the JVM has no more memory for it, *OutOfMemoryError* is thrown. Both types of error result in the program termination (cannot be *caught*). The fixed stack size can be controlled at the application launch by the `-Xss` option: `java -Xss2048k`.

Benefits of Methods

Well designed methods (and classes) reduce the effort required on a large programming project. Their benefits include:

- **Reusable code.**

Code encapsulated in methods can be called many times from different locations in a program, or even from different programs.

- **Isolation from unintended side-effects.**

If a calling program communicates with a method only through its parameters (*and the method does not modify fields of the objects passed to it*), then the method cannot corrupt data in the main program. This makes the program easier to debug and maintain.

- **Independent testing of sub-tasks.**

Each method can be tested using an appropriate "harness" before combining it with the rest of the program.

Method Invocation

If the object `x` is an instance of a class X, which contains methods `doThis()`, `doThat(typeY param)` and `returnSmth()`, where the last method is declared to return a value of the type *typeZ*, then other parts of the program which are aware of the object `x`, can call these methods:

```
X x = new X(…);
typeY y = expr; // expr's type must be compatible with typeY
typeZ z;
x.doThis();
x.doThat(y);
z = x.returnSmth();
```

Depending on how the class X is declared, and how the client's (the object from within which the method call is made) class Y is declared, the calling is possible if:

- the method in X is declared `public`, then X and Y can be unrelated
- the method is declared `protected`, then Y must be a *subclass* of X
- the method is declared `private`, then the call can only be made from within the class (X and Y are the same)

Pure Functions and Referential Transparency

In older imperative languages (Fortran, C), functions with side-effects were common (due to unbridled use of global variables), creating difficult debugging problems. Ideally, methods should not change their surrounding at all — *side-effects free*, or *pure*. The ultimate desired property is *referential transparency* (a function call is equivalent to the value it returns). In functional programming all methods are pure.

In view of the *call-by-value* effects when a method is passed a reference parameter, care is needed to prevent unintended change in the object which is referenced by the actual parameter, otherwise state of other objects and the whole program can be adversely affected, causing errors or crashes.

Engineering a Side Effect

An exercise

1. Define a (two-dimensional) *Point* class.
2. Define two classes, *Circle* and *Rectangle* which have necessary fields to represent their known geometric properties (radius), some of which (centre, vertices) are represented as instances of *Point*.
3. Define necessary constructors to control the above geometric characteristics. What is the best way of using those constructor parameters which are *points*? Try the naive way.
4. Add methods `shiftX(double x)` and `shift(double y)` to perform the obvious point objects manipulations. Perhaps, shifting circle and rectangle can be defined using the already defined shifts of *Point*?
5. If you defined your constructors naively, and created objects as follows:

```
Point p1 = new Point(0,0), p2 = Point(10, 10);
Circle c = new Circle(p1, 5);
Rectangle r = new Rectangle(p1,p2); // p1, p2 define bounding box for r
c.shift(-10); // shifting circle c left by 10 units
```

   Nothing is done explicitly to the rectangle `r`, but has its state not changed? How to define constructors to prevent unintended change in `r` from ever happening? Do some search about *copy constructors* to help yourself if necessary.

Operations as Data: Methods?

- Having defined a set of instructions as a method, we can execute them by *calling* it (somewhere in our program):

```
public void sayBlah(String utterance, int n) {
     while (n > 0) {
         System.out.println(utterance);
         n--;
    }
}
...
sayBlah(10);
```

- But can be define another operation in which other, already defined operations can be used as parameters? Can we treat our code (in methods) as data? We cannot pass a method name and make the new method to invoke it, like this:

```
public void socialising(sayBlah sb, int iq) { // cannot do!
    if (iq < 100)
        sb("blah", 100 - iq);
    else
        discussFilm("Melancholia");
}
```

Methods do not have type! (Only objects do.)

Operations as Data: Methods? No, λ-expressions!

- The need for programs to do this kind of trick is real — it may simplify programs substantially by rising the level of abstraction; for example, we would like to maintain a conversation, and depending on the IQ of our interlocutor, to either engage in a small talk, or discuss a complex topic. **Or** — using the same unit of code — "play" tennis, or perform a piano piece for four hands? In other words: Can we abstractly represent all possible actions by a symbol?
- Before Java 8, the only way of achieving this was to use (anonymous) inner classes. What?! (No worries — we'll learn this later.) But it was (still is) awkward.
- Now there is a much better way — λ-expression:

```
BiConsumer<Integer,String> smallTalk = (i,s) -> sayBlah(s, i);
Consumer<String> discussFilm = title -> readReviewFromIMDB(title);
...
public void socialising(BiConsumer lowIQ, Consumer highIQ,
                        String title, int iq) {
    if (iq < 100) lowIQ.accept(100 - iq, "blah");
    else highIQ.accept(topic);
}
...
socialising(smallTalk, discussFilm, "blah", 75); // method call
```

Imaging the possibilities... More (much more) on this later — **F1**, **F3**.

Further Reading

- Horstmann *Core Java for the Impatient*, Ch. 1.9, 2.2
- Oracle's Java Tutorial Defining Methods