THE UNIVERSITY OF MELBOURNE

Enhancing career prospects for women in STEM

Internship Information Session:
Monday 20 August, 3.15pm
Greenwood Theatre

Increase your graduate employability with an internship during your degree!

If you are a female student majoring in Computer Science/IT, come along to this session to find out how to approach your internship search and what additional support is available to you.

—

Register at: **go.unimelb.edu.au/mc46**

Programming and Software Development
COMP90041

Lecture 4

Methods

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Methods

- A method is an operation defined by a class
- *I.e.*, it defines how to do something
- The Java library defines many methods
- And you can define your own
- Similar to functions, subroutines, procedures in other languages
- Java supports two kinds of methods:
  - Class or `static` methods, and
  - Instance or non-`static` methods
- Instance methods are more common, but Class methods are simpler, so we start there

Using Class Methods

- Calling a class method runs the code in the body of the method before returning to execute the code following the method call
- Form: `Class.method(expr1, expr2, ...)`
- Can omit `Class.` if caller defined in same class
- The `expr`s, called arguments, provide data for the method to use
- Arguments are evaluated before executing method
- On completion, the called method can return a value to the caller
- The caller can then use the returned value in further computations

The Math Class

The `Math` class is a library of class methods and constants, including (among many more):

Method	Type	Description
`abs(int)`	`int`	absolute value
`ceil(double)`	`double`	ceiling
`E`	`double`	$e = 2.71828\ldots$
`max(int, int)`	`int`	larger of two ints
`min(int, int)`	`int`	smaller of two ints
`floor(double)`	`double`	floor
`PI`	`double`	$\pi = 3.14159\ldots$
`pow(double,double)`	`double`	a^b
`sqrt(double)`	`double`	$\sqrt{a}$

Example Method Use

`sqrt` is a class method to compute square root

```
import java.util.Scanner;
public class Hypot {
    public static void main(String[] args) {
        System.out.print("Enter triangle sides: ");
        Scanner kbd = new Scanner(System.in);
        double side1 = kbd.nextDouble();
        double side2 = kbd.nextDouble();
        double hypot = Math.sqrt(side1*side1 +
                                 side2*side2);
        System.out.println("Hypot = " + hypot);
    }
}
```

Example Method Use

Program Use

```
nomad% java Hypot
Enter triangle sides: 3 4
Hypot = 5.0
```

Wrapper Classes

- Many other classes provide class methods
- For each primitive type, there is a wrapper class that defines some useful class methods:

Primitive	Wrapper
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Convert from Primitive to String

Wrapper classes provide methods to convert primitive types to strings

Type	**Convert to** `String`
`byte x`	`Byte.toString(x)`
`short x`	`Short.toString(x)`
`int x`	`Integer.toString(x)`
`long x`	`Long.toString(x)`
`float x`	`Float.toString(x)`
`double x`	`Double.toString(x)`
`boolean x`	`Boolean.toString(x)`
`char x`	`Character.toString(x)`

Parsing Numeric Strings

Command line arguments are always strings.
To convert to number types:

Type	Convert from `String s`
`byte`	`Byte.parseByte(s)`
`short`	`Short.parseShort(s)`
`int`	`Integer.parseInt(s)`
`long`	`Long.parseLong(s)`
`float`	`Float.parseFloat(s)`
`double`	`Double.parseDouble(s)`
`boolean`	`Boolean.parseBoolean(s)`
`char`	`s.charAt(0)`

The `main` Method

- `main` is a class method we've been defining
- Java executes the `main` method when running an application
- Begins with:

```
public static void main(String[] args) {
```

- Ends with:

```
}
```

Defining Class Methods

- General form (for now):

```
public static type name(type1 var1,type2 var2,...) {
    :
}
```

- Each *var* is called a parameter
- Parameter is a variable initialised to value of corresponding expression in method call
- Then body (: part) of method is executed
- Types of corresponding arguments and parameters must match
- *type* is type of result returned

Returning Results

- Return result of method with `return` statement
- Form: `return` *`expr`*
- Value of expression *`expr`* is result of method
- Method completes as soon as `return` statement is reached (even if there is code after)
- Type of *`expr`* must match *`type`* of method
- Can have multiple `return` statements, but every execution of method must reach a `return` (but see exception below)
- Compiler error if either is violated
- Can use `return` to terminate loop and return from method

Example Method Definition

```
public class Hypot2 {
    public static void main(String[] args) {
        double side1 = Double.parseDouble(args[0]);
        double side2 = Double.parseDouble(args[1]);
        double hypot = hypot(side1, side2);
        System.out.println(hypot);
    }

    public static double hypot(double side1,
                               double side2) {
        return Math.sqrt(side1*side1 + side2*side2);
    }
}
```

Example Method Use

Program Use

```
nomad% java Hypot 3 4.0
5.0
```

- Original Hypot program was interactive: it asked for input it needed and explained its output
- This version takes all input on the command line
- Interactive version is more user-friendly
- This version is more machine-friendly: easier for another program to control
- You will be asked to write many machine-friendly programs for this reason

When to Define Methods

- In this example, we turned a chunk of code (hypotenuse computation) into a new method
- This is called refactoring
- When to define a new method:
  - When a method gets too big (more than can be easily viewed at once, more than $\approx$ 60 lines)
  - When you repeat similar code multiple times
  - When you can give a good name to a chunk of code (*e.g.*, `hypot`)
- How to break up the work of a program into methods is an important and complex issue
- We will revisit later

Not Returning Results

- Use method call as an expression
- Value of expression is value returned by method
- But can also use method call as statement
- Ignore returned value, just execute for effect (*e.g.*, printing)
- If always want to ignore, use return type `void`
- Means don't return anything
- `main` method has return type `void`
- Then don't need `return` statement
- Can use `return` with no expression to immediately return nothing

Headers and Signatures

- First part of method definition (up to {) is called the method header
- Header defines return type, method name, number and types of parameters, and parameter names
- Method name plus number and types of arguments together are called the method signature
- Signature is used to decide which method to call

Overloading

- `abs`, `min`, and `max` all work on all of `double`, `float`, `int`, and `long` types
- They return the same types
- Overloading: when a method name has multiple definitions, each with different signature
- Java automatically selects the method whose signature matches the call
- You can define your own overloaded methods, too
  - Just define multiple methods with same name but different signatures
- Uses: support multiple types, simulate default arguments

Limitations of Overloading

- You cannot define two methods with the same name and all the same argument types
- You cannot overload based on return type, only parameter types
- You cannot overload operators (*e.g.*, +, *, *etc.*)
- Beware of combining overloading with automatic type conversion!

```
int    bad(int x, double y) {...}
double bad(double x, int y) {...}
```

What if you call `bad(6, 7)`?

`public` and `private`

- The keyword `public` in method header means the method may be used by any method in any class
- The keyword `private` in header means the method may be used from any method only within that class
- Visibility: from where method can be seen
- `public` implicitly promises to (try hard to) maintain that method without changing its signature
- Best practice: make methods private unless they need to be public
- Best practice: make signature of public methods as simple as possible

Local Variables

- Scope: where variable can be referenced
- Variables declared inside methods are local to the method (cannot be used outside)
- Cannot be referenced before declaration is executed
- Variable's value is forgotten when it goes out of scope; gets a fresh value next time in scope
- Variable declared inside a block is scoped to that block
- Parameters are also local to the method
- Local variables cannot be declared `public` or `private`

Defining Constants

- `Math` class defines constants `PI` and `E`
- You can define your own constants for your code
- Form:
  `public static final` *`type name = value`*`;`
- At top level of class declaration
- Can be `public` or `private`
- Java naming convention: all uppercase, words separated with underscores
- *E.g.*:

```
public static final int DAYS_PER_WEEK = 7;
public static final int CARDS_PER_SUIT = 13;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code much easier to understand
- What does this do?

```
x += 168;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code much easier to understand
- What does this do?

```
x += 168;
```

Compare that with:

```
x += DAYS_PER_WEEK * HOURS_PER_DAY;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code much easier to understand
- What does this do?

```
x += 168;
```

Compare that with:

```
x += DAYS_PER_WEEK * HOURS_PER_DAY;
```

- Also symbolic constants defined in one place are much easier to change if necessary, eg:

```
private static final int CHARS_IN_SUBJECT_CODE = 6;
```

When Not to Define Constants

- Don't define a constant for something you can't meaningfully name
- This is useless:

```
public static final int SEVEN = 7;
```

- Don't (usually) define a name for 0 or 1: `n == 0` is just as good as `n == NONE`

Class Variables

- Class variable is a variable that is local to a class
- Lifetime of variable: when it first has a value until it ceases to exist
- Lifetime of class variable is from start of program until exit
- Value survives through message calls and returns
- Value is unchanged until reassigned
- Only one "copy" of each class variable at a time
- Can be declared either `public` or `private`
- It should almost always be `private`
  - Difficult to control if every method can modify it

Class Variables

```
public class ClassVar {
    private static String name = "Someone";
    public static void main(String[] args) {
        greet("Hello");
        setName("Kitty");
        greet("Hello");
        greet("Aloha");
    }
    private static void setName(String name) {
        ClassVar.name = name; // 2 vars called name!
    }
    private static void greet(String greeting) {
        System.out.printf("%s, %s!%n", greeting, name);
    }
}
```

Example Method Use

Program Output

```
Hello, Someone!
Hello, Kitty!
Aloha, Kitty!
```

- This program is stateful: behaviour depends on what has come before
- `greet("Hello")` does two different things!
- Makes it harder to predict program behaviour
- Keep use of class variables to a minimum

Summary

- Executing class (static) methods executes their definition
- Method calls can pass arguments
- Methods can can return a value with a `return` statement
- Methods declared with `private` instead of `public` can only be used inside the same class
- Multiple methods can have same name, if number/types of arguments are different

THE UNIVERSITY OF
MELBOURNE