CSE 1710

Lecture 8

Anatomy of an API

Recap and Review of Core Concepts

3.1.1 Overall Layout

Packages	Details
	The Class section
Classes	The Field section
	The Constructor section
	The Method section

3.1.2 Fields

From class java.lang.Math

Field Summary

`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Field Detail

`PI`

`public static final double PI`

The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

See Also: Constant Field Values

3.1.3 Methods

The Math class of java.lang

Method Summary

`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.
`static int`	**`abs`**`(int a)` Returns the absolute value of an `int` value.
`static double`	**`pow`**`(double a, double b)` Returns the value of the first argument raised to the power of the second argument.

The Scanner class of java.util

Method Summary

`double`	**`nextDouble`**`()` Scans the next token of the input as an `double`.
`int`	**`nextInt`**`()` Scans the next token of the input as an `int`.
`String`	**`nextLine`**`()` Advances this scanner past the current line and returns the input that was skipped.
`long`	**`nextLong`**`()` Scans the next token of the input as an `long`.

Method Detail

abs

```
public static double abs(double a)
```

Returns the absolute value of a `double` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Special cases:

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is `NaN`, the result is `NaN`.

Parameters:
`a` – the argument whose absolute value is to be determined
Returns:
the absolute value of the argument.

Key points to remember

Parameters are Passed by Value
what does this mean?

```
double inputtedValue;
```

… here would be some statements which prompt the user for input and read in the inputted value..

```
double myVal = Math.abs(inputtedValue);
```

… so can the method `abs` *possibly change the value of* `inputtedValue`*?*

NO!

Primitive values stored in your variables cannot be inadvertently changed by passing the variables to a method

Key points to remember

Consider this…

```
InputStream inputStream;
inputStream = System.in;
Scanner input = new Scanner(inputStream);
```

… so can the constructor `Scanner` *possibly change the* ***state*** *of* `inputStream`*?*

YES!

… so can the constructor `Scanner` *possibly change the* ***value*** *of* `inputStream`*?*

NO!

Key points to remember about methods

- **Methods can be Overloaded**

- **A class can have methods with the same name, provided the signatures are different.**
- **A class cannot have two methods with the same signature**
- **Recall, signature is the name and parameters (does not include the return).**

Key points to remember about methods

Binding with Most Specific

- The compiler wants **to bind**
  - this means to find the target **to which an identifier resolves**
  - all identifiers must resolve (must correspond to a definition, somewhere)
- To bind `C.m(…)` here's what the compiler does:
  - locates `C` (or else issues No Class Definition Found)
  - locates `m(…)` in `C` (or else issues Cannot Resolve Symbol).
    - If more than one such `m` is found, it binds with the "most specific" one.
    - look at abs as an example

What is a method?

- performs some action
- has a signature and return

range of possibilities?

0 or more parameters, type compatibility must be assured

```
var.methodName( )
Classname.methodName( )
```

What is an attribute?

- holds data
- has a name and a type
- declared and initialized in the class defn

NO parameters

```
var.attributeName
```

In general…

- both are *members* of a class, (also called features)
  - method signatures must be unique, attribute names must be unique
- compiler checks *invocations*:
  - does the signature (or the attribute name) **match** what is in the class definition?
  - the attributes that clients can access are called fields

What is scope? What defines scope?

- the term ***scope*** is used to refer to the block within which a variable has been declared
- does the term ***scope*** apply to ***fields (e.g., class attributes that are public)?***
  - yes, but in a trivial sort of way
  - Their scope is anywhere. The class defines the field. Once a class is imported, the field can be used anywhere.

What is a class?

- a definition
- e.g., a description of a car
- gets created in advance
  - it is compiled, is bytecode
- it (the bytecode) gets loaded into runtime memory by the VM upon invocation of an app

What is an object?

- an actual instance of the thing that was defined
- e.g., an actual car
- gets created at runtime
- it gets "born" during runtime, it "dies" during runtime
- has a *state* during runtime
  - specific values for all attributes

What do classes have?

- they have *definitions* of features:
  - static attributes
  - non-static attributes
  - static methods
  - non-static methods

What do objects have?

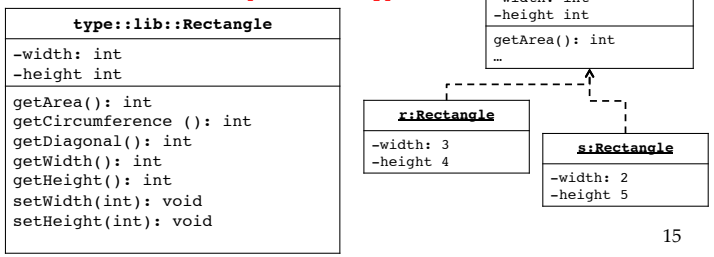
- they have a set of attributes, each of which has a value:
  - the static attributes, if any, are common to all objects of a given type; the value must be the same
  - the non-static attributes are specific to each object; the values may differ
- they have methods defined upon them

UML Class and Object Diagrams

- full class names separated by colons
  - In java code, full class names separated by dots
- attributes in class diagrams:

  `attributeName : type`
  - a + or – symbol in front means private or public, resp'y
- methods in class diagrams:

  `methodName (param) : type`



What is abstraction?

- a process whereby details are replaced with something simpler
  - nature of these details?
    - object properties?
      - » abstraction by parameterization
    - details about how a task is performed?
      - » abstraction by delegation

Why do we use abstraction?

- To reduce complexity

RQ2.19

What is an app vs an application?

- app: a class with a main method
- application: an app plus several components

RQ2.20,2.21

What does it mean to be a client?

- to know how & where to look for components
  - understand package structure
  - understand class names may not be unique
  - understand how to read an API, UML diagrams
    - API: a document that specifies what a component does
- to know what you want your app to do
  - not necessarily *how* to implement each and every sub-component
  - you can delegate this to other components
- to know how to use components
  - how to construct objects or otherwise get references to them
    - for delegation of representation, delegation of tasks
  - how to invoke methods, make use of fields
    - static and non-static variants

RQ2.22-2.23

Illustrations of encapsulation

- knowing how to signal a left turn while driving a car does not break encapsulation
  - knowing how to activate the signal does mean knowing how the signal actually operates
    - e.g., how is signal wired? where is the fuse? what is the wattage of the bulb?.
- encapsulation makes the lives of the client and the implementer easier
  - the client needs not know how the component works
  - the implementer needs not know what is the component used for.

RQ2.24

Can the client and implementer roles be occupied simultaneously?

- Depends on who is looking at the situation
  - with respect to end users
    - the end user is the client
    - the application is an implementer
  - with respect to a particular component (no main method)
    - an app that uses the component is the client
    - the component is the implementer

RQ2.25

What does the VM do when a program crashes or has a bug?

- for crashes
  - VM identifies where the problem occurs in the stack trace
- for bugs
  - VM will not realize that there is a bug, so it cannot possibly flag them
- debugging
  - you (not the VM) need to determine why the program produced an incorrect result
  - may need to trace the entire program

RQ2.26

So what is the difference between a bug and other types of errors?

- a bug
  - depends on some notion of what correct output looks like
- compile-time error
  - compiler has a problem with the syntax
  - need to understand compiler's error message
- run-time error
  - VM had a problem running the byte code
  - need to understand stack trace

RQ2.27-2.29

Who's to blame when run-time errors occur?

- run-time error in the main class
  - could be the user
    - provided invalid input?
  - could be the main class
    - has faulty implementation?
- run-time error in a component
  - could be the main class
    - passed invalid parameters?
  - could be the component
    - has faulty implementation?

RQ2.30-2.31

What are the key concepts about Software Engineering?

- it is study of software projects and their progress
- "Risk Mitigation by Early Exposure" is a key principle it is not about program
  - for instance
    - converting the type of a value at runtime is risky
      - e.g., converting a double to an int will result in data loss)
    - the compiler mitigates this risk by checking type compatibility and refuse to compile if there is a violation

What do I need to know about constants?

- literals embedded in expressions or as parameters are magic numbers
  - you used a literal because:
    - some particular value is needed
    - that particular value is pre-defined and unchanging
- magic numbers should be avoided
- use variables instead of magic numbers
- how do you enforce that the value is predefined and not able to change?
  - use the keyword **`final`** before the declaration.

What do I need to know about contracts?

- useful during development and testing
  - stipulates the division of responsibilities:
    - the client
      - needs to ensure the precondition is met
    - the implementer
      - needs to ensure the postcondition is met
  - if precondition is not met, then it is client's responsibility for whatever happens
    - this absolves the implementer of any responsibility
      - implementer may (1) cause crash or (2) return something which may or may not be as specified under the post
    - a dangerous condition can arise if the false precondition does not cause the program to crash