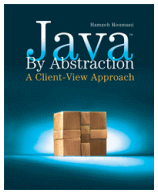


Chapter 3

Using APIs

1

Outline

2

3.1.1 Overall Layout

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

3

3.1.2 Fields

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

4

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.

5

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.

6

Key points to remember about methods

- Methods can be "Overloaded"
- A class cannot have two methods with the same signature (even if the return is different).
- If the signatures are different, methods can have the same name
  - this situation is an example of "overloading"

7

Key points to remember about methods

- "Binding with Most Specific"
- Situation: the invocation `C.m(…);`
  - `C` is a class name, `m` is the name of the method *invoked* on `C`
- During compilation, the compiler (javac) attempts to:
  1. locate the class `C`, and then
  2. locate the method `m` (that is, the method with the signature specified by the "…" in the *invocation*)

This is called binding (as in "to bind `C.m(…)`")

- If #1 fails, No Class Definition Found error is issued
- If #2 fails, "cannot find symbol" error is issued
- For #2, if more than one such m is found, the compiled picks the "most specific" one.

8

3.2 A Development Walkthrough

9

3.2.1 The Development Process

- Analysis — The Requirement: Input & its validation Output & its formatting
- Design
- Implementation
- Testing
- Deployment

10

The Development Process

- Analysis
- Design — An algorithm (function) that determines the output given the input.
- Implementation
- Testing
- Deployment

11

The Development Process

- Analysis
- Design
- Implementation — Turn the algorithm into a program
- Testing
- Deployment

12

The Development Process

- Analysis
- Design
- Implementation
- Testing — Does the program meet the requirement?
- Deployment

13

The Development Process

- Analysis
- Design
- Implementation
- Testing
- Deployment — Installation, porting, training, support…

14

3.2.5 Input Validation

Invalid inputs are the cause of most errors in programs. Therefore, upon encountering one, a program must either:

- **Print a message and end**
- **Print a message then allow the user to retry several times or decide to abort.**
- **Trigger a runtime error; i.e. crash.**

For now, let us use the 3rd via a method in Toolbox:

```
static void crash(boolean, String)
```

15

3.2.6 Assertions

A simple yet powerful tool to guard against errors that arise from misunderstandings.

Whenever you believe that some non-trivial condition is true, assert it, e.g.

```
assert payment >=0;
```

You cannot assert a validation because user input is not under your control. Hence, do not confuse assert (a Java statement) with crash (a method).

16

3.3 General Characteristics of Utility Classes

17

3.3.1 Memory Diagrams

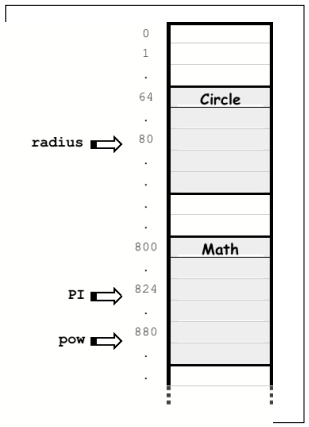
Let us compile and load the program, Circle, which uses a field and a method in the Math utility class.

```
import java.util.Scanner;
import java.io.PrintStream;

public class Circle
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      PrintStream output = System.out;
      output.print("Enter radius: ");
      int radius = input.nextInt();
      output.println(Math.PI * Math.pow(radius, 2));
   }
}
```

18

Memory Diagrams



19

3.3.2 Advantages of Utility Classes

Simplicity

- To access a static field f in a class C, write: C.f
- To invoke a static method m in a class C, write C.m(…)
- There is only one copy of a static class in memory

Suitability

- A utility class is best suited to hold a groups of methods that do not hold state, e.g. java.lang.Math.
- Even in non-utility classes, static is best suited for features that are common to all instances, e.g. the MAX_VALUE field and the parseInt method of the (non-utility) class: Integer.

20

3.3.3 Case Study: Dialog I/O

Two static methods in:

`javax.swing.JOptionPane`

- **To display a message:**

```
void showMessage(null, message)
```

- **To prompt for and read an input:**

```
String showInputDialog(null, prompt)
```

Note that showInputDialog returns a String. Hence, if you use it to read a number, you must invoke one of the parse methods in the corresponding wrapper class.

21