CSE 1020: Week 3

Topic: Using classes and APIs

To do: Textbook Chapter 3; Lab 3

1

Outline

- Modules
- APIs and class use
- Input/output
- Boolean and relational operations
- Software (re)engineering

2

Outline

- **Modules**
- **APIs and class use**
- **Input/output**
- **Boolean and relational operations**
- **Software (re)engineering**

3

Modules

Abstraction

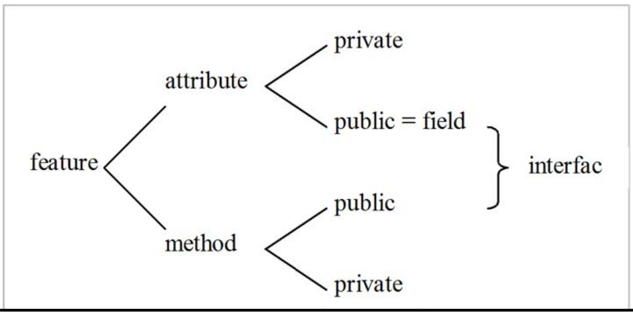
- A typical app makes use of predefined components.
- For convenience, we group related components together in classes.
- For example,
  - in MkChange we used print from the PrintStream class.
- Other components that provide services related to output also reside in the PrintStream class.
- Similarly, many components related to mathematical concepts are grouped together in the Math class.
- Such components are abstractions that can be used without knowing how they are implemented.

4

Modules

Information hiding

- The class hides the details of how the operations of the class are implemented from users of the class.
- Those aspects of the class that are visible to the user are said to be public.
- Other aspects of the class, having to do with how it is implemented, are not available to the user and are said to be private.
- Information on the public aspects of the class are collected together in the class API, which users can consult.



5

Modules

Static classes

- The simplest kind of class is a static class or a module.
- As examples:
  - ToolBox is a static class;
  - Math is a static class.
- Such classes simply work as a grouping mechanism for related data and operations.

6

Modules

A static class provides

- static methods: operations that belong to the class that can be used by users (without knowing implementation)
  - e.g., ToolBox.getBMI;
- static constants: whose values can be retrieved by users
  - e.g., Integer.MAX_INT;
- static variables: whose values can be retrieved and changed by users
  - e.g., IO.fillChar.
- Remark: All features (methods, constants, variables) of a static class are static.

7

Modules

How to refer to static class components

- Methods: ClassName.methodName(parameters)
  - e.g., ToolBox.getBMI(165.0,"6'1");
- Constants: ClassName.constantName
  - e.g., Math.PI;
- Variables: ClassName.variableName
  - e.g., IO.fillChar;

8

Modules (and beyond)

Static classes

- The simplest kind of class is a static class or a module.
- For example, Math is a static class.

Non-static classes

- There also is another kind of class where the user can create customized versions, called instances, according to a predefined template.
- The instances are called objects.
- Such classes have non-static methods and fields.

Terminology

- A class is static if it does not allows us to define our own copies.
- A class is non-static if it does allow us to define our own copies.

9

Outline

- Modules
- **APIs and class use**
- Input/output
- Boolean expressions and relational operators
- Software (re)engineering

10

APIs and class use

What is an API

- The term API stands for Application Programming Interface
- Documents how another program can access a given class.
- Hides implementation detail.

Why we care: Guide to ready made software modules

- As an applications programmer, we use the API of a class for two main reasons
  1. By perusing the API of a class we can determine if it provides useful functionality for the task that we are addressing.
  2. If we discover useful functionality, then the API tells us how to access it.

11

APIs and class use

API anatomy: Overview

- There is variability in the layout of particular APIs.
- We will review a fairly standard layout and the one that we try to adhere to in CSE1020.
- As a particular example, we consider the Math class.

12

APIs and class use

API anatomy: Basic terminology

- The term field will refer to a piece of information that is maintained about the class as a whole.
- Examples:
  - Math.E;
  - Math.PI.

13

APIs and class use

API anatomy: Basic terminology

- The term method will refer an operation that can be performed with respect to the class.
- Examples:
  - Math.abs;
  - Math.min.

14

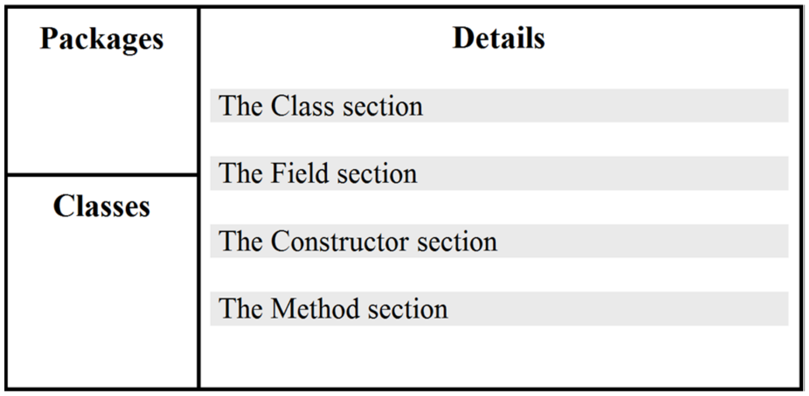
APIs and class use

API anatomy: Basic terminology

- The terms *field* and *method* are not standard.
- Following are terms that can be deemed equivalent.
  - field = attribute = variable = data member = property
  - method = message = member function = sub

APIs and class use

API anatomy: Overall layout



APIs and class use

API anatomy: Structure

- The documentation is divided into 3 main parts

1. **Descriptive paragraph:** Provides a high level overview of the class.
2. **Summary:** A terse specification of the interfaces to the fields and methods that are available.
3. **Detail:** Provides additional information about the fields and methods that are available.

17

APIs and class use

API anatomy: The descriptive paragraph

public final class **Math**
extends Object

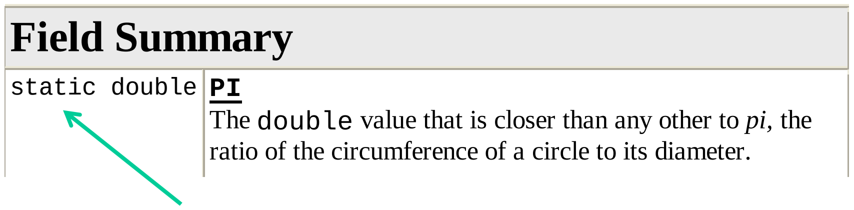
The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
…

Since:
JDK1.0

18

APIs and class use

API anatomy: Fields



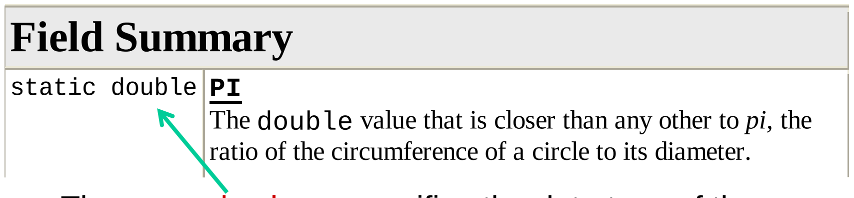
Field Summary

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.

- The first column specifies if the field pertains to the class as whole (or instances of the class).
- The term static denotes that the field pertains to the class as a whole
- (A blank denotes that the field pertains to instances of the class.)

APIs and class use

API anatomy: Fields



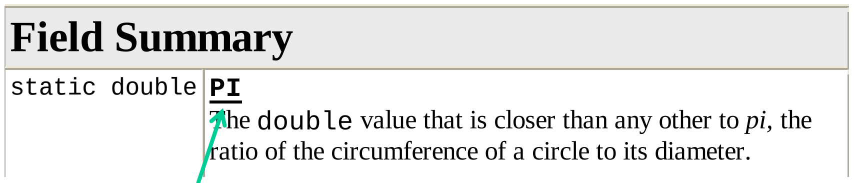
Field Summary

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.

- The second column specifies the data type of the field.
- The data type can be any primitive type (or any object type.)

APIs and class use

API anatomy: Fields



Field Summary

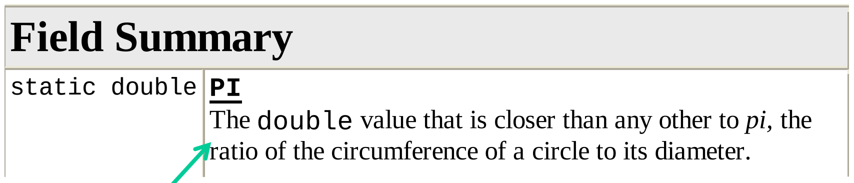
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

- The third column specifies the name of the field.
- Follow Java and CSE1020 style manual naming conventions.

21

APIs and class use

API anatomy: Fields



Field Summary

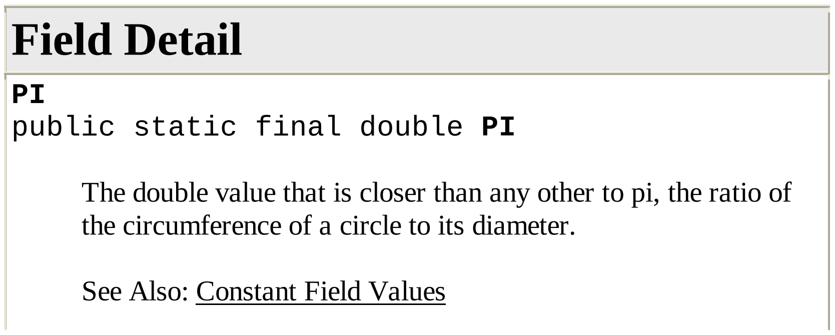
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

- The text provides a terse human language description of the field.

22

APIs and class use

API anatomy: Fields



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.

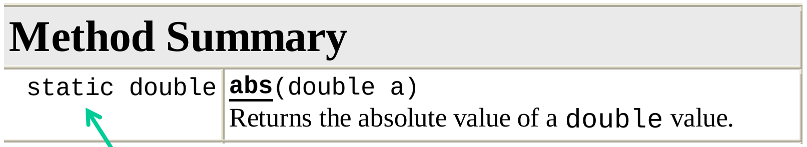
See Also: Constant Field Values

Here we see specification of
- visibility (public)
- if a field is constant (final)
- more descriptive detail

23

APIs and class use

API anatomy: Methods



Method Summary

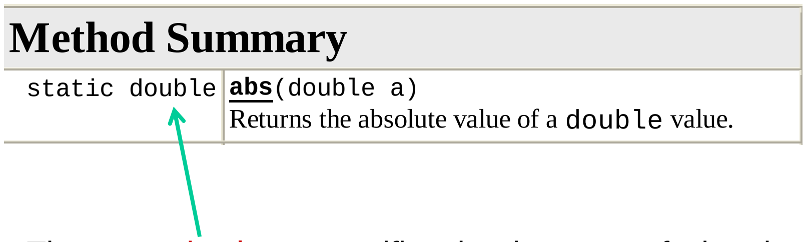
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.

- The first column specifies if the method is for the class as a whole (or instances of the class).
- The term static denotes that the method is for the class as a whole (a blank indicates applicability to instances).

24

APIs and class use

API anatomy: Methods

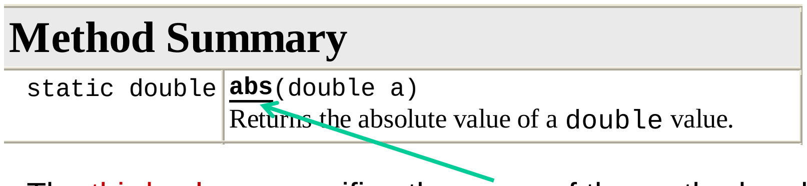


Method Summary	
`static double`	`abs(double a)` Returns the absolute value of a `double` value.

- The second column specifies the data type of what the method returns.
- The keyword void is used if nothing is returned.

APIs and class use

API anatomy: Methods

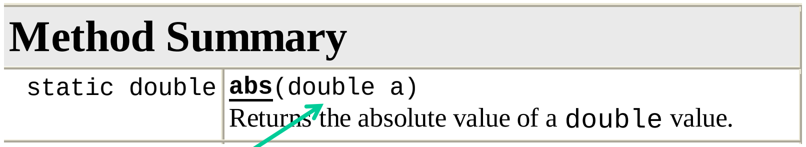


Method Summary	
`static double`	`abs(double a)` Returns the absolute value of a `double` value.

- The third column specifies the name of the method and its parameters (type and parameter name).
  - We call this the method's *signature*.
- The parameters provide the information for the method to do its job.
- Not all methods take parameters.

APIs and class use

API anatomy: Methods



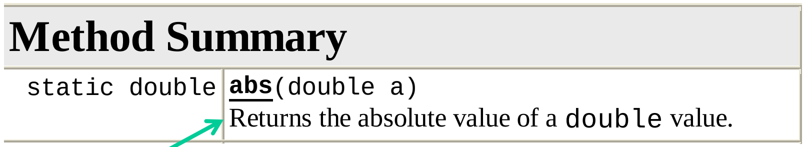
Method Summary	
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.

- The third column specifies the name of the method and its parameters (type and parameter name).
  - We call this the method’s *signature*.
- The parameters provide the information for the method to do its job.
- Not all methods take parameters.

27

APIs and class use

API anatomy: Methods



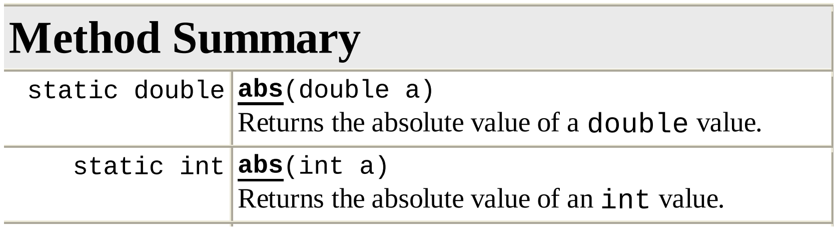
Method Summary	
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.

- The text provides a terse human language description of the method.

28

APIs and class use

API anatomy: Methods



Method Summary

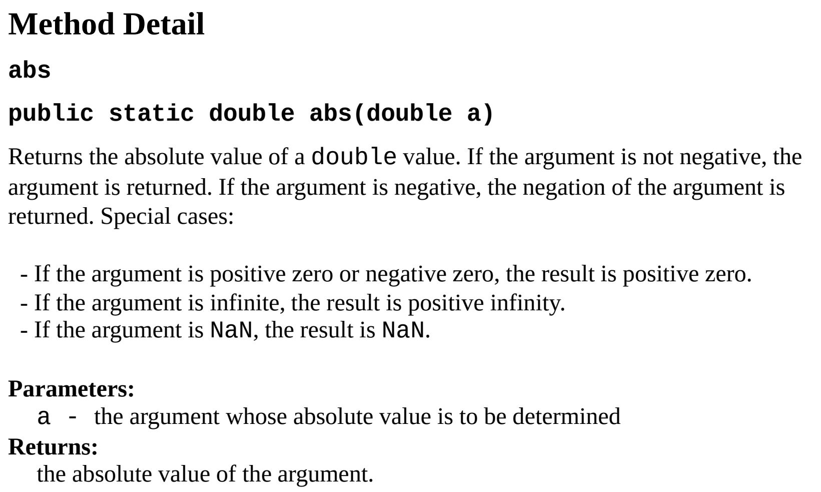
`static double`	`abs(double a)` Returns the absolute value of a `double` value.
`static int`	`abs(int a)` Returns the absolute value of an `int` value.

- Notice that it is possible to define multiple methods with the same name…
- …as long as the complete signatures all differ.
- This process is called overloading.
- Allows the conceptually same operation to be performed on different parameter types.

29

APIs and class use

API anatomy: Methods



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.

Here we see specification of
- visibility (public)
- greater explanatory detail

30

APIs and class use

A few other points of note

- **Parameters are Passed by Value:**
  Values stored in your variables cannot be inadvertently changed by passing the variables to a method
- **Binding with Most Specific:**
  To bind C.m(…) the compiler locates C (or else issues No Class Definition Found) and then 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.

31

APIs and class use

A good place for additional self study

- The ability to read and read quickly an API is an important skill for students 1020.
- Indeed, this is a skill that will help you throughout your programming career.
- Spend a good amount of time perusing the API of, e.g., Math.
- Check to make sure you understand what is presented by writing little apps that make use of what you find.

32

APIs and class use

How we gain access to a class

- To refer to a class in our program, we make use of the import statement

```
import packageName.ClassName;
```

e.g.,

```
import java.io.PrintStream;
```

- To import all classes in a package

```
import packageName.*
```

33

APIs and class use

How we gain access to a class

- To refer to a class in our program, we make use of the import statement

```
import packageName.ClassName;
```

e.g.,

```
import java.io.PrintStream;
```

- To import all classes in a package

```
import packageName.*
```

Remark

- You do not need to explicitly import classes in package java.lang

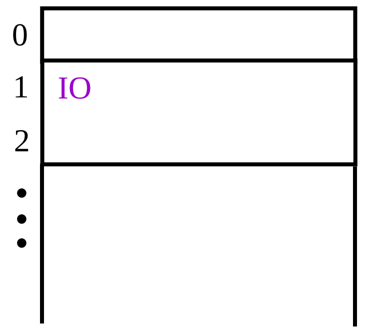
34

APIs and class use

Memory diagram: Importing and using a class

- When an application is launched, the class files are loaded into memory.
- That is, a copy of the class is loaded into memory by the operating system.
- Import statements tell which classes need to be loaded, e.g.,

  import type.lang.IO;
- The same copy is used for the rest of the program run.



35

APIs and class use

Using static methods

- To invoke a static method *m* in a class *C*, the following steps are followed:
  1. Ensure that class *C* is reachable (already available or imported).
  2. Determine the signature of *m*: parameters, their data types and their order.
  3. Invoke using the dot: *C.m(parameters)*
- Example: double x = Math.sin(3.14);

36

APIs and class use

Using static methods

- To invoke a static method *m* in a class *C*, the following steps are followed:
  1. Ensure that *C.class* is reachable (already available or imported).
  2. Determine the signature of *m*: parameters, their data types and their order.
  3. Invoke using the dot: *C.m(parameters)*
- Example: double x = Math.sin(3.14);

Remarks

- To call the methods we supply an argument (or actual parameter) for each formal parameter.
- When the call occurs
  1. The values of the arguments are assigned to the formal parameters.
  2. The method is executed.

APIs and class use

Using static methods

- To invoke a static method *m* in a class *C*, the following steps are followed:
  1. Ensure that *C.class* is reachable (already available or imported).
  2. Determine the signature of *m*: parameters, their data types and their order.
  3. Invoke using the dot: *C.m(parameters)*
- Example: double x = Math.sin(3.14);

Remarks

- A void method, e.g., println() is often in a statement by itself.
- A non-void one like nextInt() is usually part of a statement.

APIs and class use

Using static attributes

- To access a static attribute *a* in a class *C*, the following steps are followed:
  1. Ensure that class *C* is reachable (already available or imported).
  2. Access using the dot: *C.a*
- Example:
  - double x = Math.PI;

39

APIs and class use

Using static attributes

- To access a static attribute *a* in a class *C*, the following steps are followed:
  1. Ensure that *C.class* is reachable (already available or imported).
  2. Access using the dot: *C.a*
- Examples:
  - double x = Math.PI;

Remark

- Since Math.PI is defined as a constant (final), it would be an error to attempt

```
Math.PI = 3.14; // Illegal
```

40

API and class use

Example: Problem definition (Requirements)

- Let’s calculate the circumference of a circle given the radius.

41

API and class use

Example: Analysis (Requirements)

- **Input:** Radius of circle.
- **Output:** Circumference of circle.
- **Format:**

  The circle’s circumference is <answer>.

42

API and class use

Example: Design

- **Algorithm:** We know the standard formula

$$circumference = 2\,\pi\,radius$$

- **Constants:** π
- **Variables:** radius, circumference

43

API and class use

Example: Design

- **Algorithm:** We know the standard formula

$$circumference = 2\,\pi\,radius$$

- **Constants:** π
- **Variables:** radius, circumference

Remarks

- In the Math class we find Math.PI, which is a useful way to represent π.
- The radius and circumference are reals, following 1020 convention we will make them doubles.

44

API and class use

Example: Implementation:

- Declaration
- Input
- Computation
- Output

45

API and class use

Our Template

```
/*
Class to ???.
Author: ???                                         Date: ???
*/
import type.lang.*;
import java.util.Scanner;
import java.io.PrintStream;

// Definition of the ClassName class.
public class ClassName
{   public static void main(String[ ] args)
    {   Scanner input = new Scanner(System.in);
        PrintStream output = System.out;
        // App specific DICO.
    }
}
```

46

API and class use

Our Template

```
/*
Class to calculate circumference of a circle given radius.
Author: R. Wildes                                   Date: 05/02/13
*/
import type.lang.*;
import java.util.Scanner;
import java.io.PrintStream;

// Definition of the Circumference class.
public class Circumference
{    public static void main(String[ ] args)
     {    Scanner input = new Scanner(System.in);
          PrintStream output = System.out;
          // App specific DICO.
     }
}
```

47

API and class use

DICO: Declaration

```
// Declaration.
double radius, circumference;
```

48

API and class use

DICO: Input

```
// Input
output.print("Enter circle radius: ");
radius = input.nextDouble();
```

49

API and class use

DICO: Computation

```
// Computation
```

$$circumference = 2\,\pi\,radius$$

50

API and class use

DICO: Computation

```
// Computation.
circumference = 2 * Math.PI * radius;
```

51

API and class use

DICO: Output

```
// Output.
The circle’s circumference is <answer>.
```

52

API and class use

DICO: Output

```
// Output
output.print(“The circle’s circumference is ”);
output.printf(“%.2f”, circumference);
output.println(“.”);
```

53

API and class use

Test and deploy

- Try this example out for yourself.
- It will be up on our section website (deployed).
- Modify it to perform other simple operations while you explore the use of static classes (e.g., Math, etc.)

54

Outline

- Modules
- APIs and class use
- **Input/output**
- Boolean and relational operations
- Software (re)engineering

55

Input/Output

General

- We will make use of facilities provided by the classes
  java.util.Scanner;
  java.io.PrintStream;
- We concentrate on
  - Reading input from the standard input, keyboard
  - Writing to the standard output, screen
- In the upcoming examples, we assume the declarations

```
Scanner input = new Scanner(System.in);
PrintStream output = System.out;
```

56

Input/Output

Strings

- To read a string (from the keyboard) we can write

  `String name = input.nextLine( );`

  or better

  ```
  output.print("Please enter your name: ");
  String name = input.nextLine( );
  ```
- The method nextLine( )
  - Waits until the user has entered a text line (terminated by hitting the enter key)
  - Returns with the entire line (w/o the end-of-line marker) as a string

57

Input/Output

Numbers

- To read an int (from the keyboard) we can write

  `int age = input.nextInt( );`

  The method nextInt( )
  - Waits until the user has entered an entire line
  - Attempts to interpret it as an integer
  - Returns with an int, if the interpretation succeeds
  - Returns with an error/exception, if failure
- To read a double, you can use nextDouble, which works similarly

  `double amount = input.nextDouble( );`

58

Input/Output

Printing out a value

- We have been using the println() method from the PrintStream class.

  output.println(amount);

  or better

  output.println(“The amount is ” + amount);
- Also available, if we do not want the newline is

  output.print(“The amount is “);

  output.println(amount);

  // amount appears on same line as previous output

59

Input/Output

Formatted output

- The printf() methods allow you to print data in a specified format.
- The desired format is specified via an additional string argument.

  output.printf(%[flag][width][.precision]conversion, x);

  where
  - flag can be 0 or ,
  - width total field width
  - precision gives the number of decimal places
  - conversion is one of d, s, f, or n

60

Input/Output

Formatted output

- The printf() methods allow you to print data in a specified format.
- The desired format is specified via an additional string argument.

  output.printf(%[flag][width][.precision]conversion, x);

  where
  - flag can be
    - 0 to left pad integers

    or
    - , to use a thousands separator for numbers
  - width total field width
  - precision gives the number of decimal places
  - conversion is one of d, s, f, or n

61

Input/Output

Formatted output

- The printf() methods allow you to print data in a specified format.
- The desired format is specified via an additional string argument.

  output.printf(%[flag][width][.precision]conversion, x);

  where
  - flag can be 0 or ,
  - width total field width
  - precision gives the number of decimal places
  - conversion is one of d, s, f, or n

62

Input/Output

Formatted output

- The printf() methods allow you to print data in a specified format.
- The desired format is specified via an additional string argument.

  output.printf(%[flag][width][.precision]conversion, x);

  where
  - flag can be 0 or ,
  - width total field width
  - precision gives the number of decimal places
  - conversion is one of
    - d valid for integer output
    - s valid for string output
    - f valid for real number output
    - n is used to output a new line

63

Input/Output

Formatted output

- The printf() methods allow you to print data in a specified format.
- The desired format is specified via an additional string argument.

  output.printf(%[flag][width][.precision]conversion, x);

  where
  - flag can be 0 (to left pad integers) or , (use a thousands separator for numbers)
  - width total field width
  - precision gives the number of decimal places
  - conversion is one of d, s, f, or n
- Remarks
  - The % and conversion are mandatory.
  - The other components are optional.

64

Input/Output

Formatted output

- Examples

```
double y = 4.3333333;
output.printf(“%.1f”,y); // prints 4.3
```

65

Input/Output

Formatted output

- Examples

```
double y = 4.3333333;
output.printf(“%.1f”,y); // prints 4.3
output.printf(“%,12d”, 1234567); // prints _ _ _ 1,234,567
```

66

Input/Output

Formatted output

- Examples

  double y = 4.3333333;
  output.printf(“%.1f”,y); // prints 4.3
  output.printf(“%,12d”, 1234567); // prints _ _ _ 1,234,567
  output.printf(”PI to 3 decimals is %.3f%n”, Math.PI);
  // prints PI to 3 decimals is 3.142 with a new line at the end.

67

Input/Output

Formatted output

- Examples

  double y = 4.3333333;
  output.printf(“%.1f”,y); // prints 4.3
  output.printf(“%,12d”, 1234567); // prints _ _ _ 1,234,567
  output.printf(”PI to 3 decimals is %.3f%n”, Math.PI);
  // prints PI to 3 decimals is 3.142 with a new line at the end.
  // The previous examples would not add a new line!

68

Input/Output

Remark

- More generally, our I/O methods will input and output from other sources (e.g., files, other programs).
- Here, we have been assuming the default values for
  - Standard in: the keyboard
  - Standard out: the screen

69

Outline

- Modules
- APIs and class use
- Input/output
- **Boolean and relational operations**
- Software (re)engineering

70

Boolean and relational operations

Boolean expressions

- A hallmark of intelligent behavior is the ability to perform different actions based on the evaluation of some condition.
- As a simple example, we may charge different (transit) fares depending on age of the traveler.

71

Boolean and relational operations

Boolean expressions

- If our programs are to perform in a flexible fashion, then they must be able to represent and evaluate such situations.
- For example, we might want to verify that certain conditions holds on our input (input validation) and, say, exit if not.

```
output.print(“Enter your age: “);
int age = input.nextInt();
// somehow send error message to user if age<0
```

- We say that such conditions are represented by boolean expressions.

72

Boolean and relational operations

Relational operators

- Simple boolean expressions can be obtained by comparing two numerical values using a relational operator.
- For example, (assuming x, y and age appropriately declared and initialized)

```
boolean b1 = x < y;
boolean b2 = x >= 0;
boolean b3 = age == 17;
```

73

Boolean and relational operations

Relational operators

- In Java, the relational operators are as follows.

==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

74

Boolean and relational operations

Relational operators

- In Java, the relational operators are as follows.

==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Remark

- For future reference: You cannot compare Strings or other objects using these operators.

75

Boolean and relational operations

Beware: Comparing floating point numbers

- Recall that floating point numbers are represented with limited precision.
- Therefore, a test for strict equality (e.g., via ==) is to be avoided.
- Instead, we check to see if two numbers of interest are *close enough*

```
double final EPSILON = 0.00000000000001;
boolean close = Math.abs(x – y) <= EPSILON;
```

76

Boolean and relational operations

Beware: Comparing floating point numbers

- To get a little more detailed:
- Since precision decreases with magnitude…
- …it can be useful to divide by some measure of the magnitude of the numbers being compared.
- For example, we consider

$$\frac{|x - y|}{\max(|x|, |y|)} \leq \varepsilon$$

- Which is reasonably coded as

```
double magnitude = Math.max(Math.abs(x), Math.abs(y));
boolean close = Math.abs(x–y) <= EPSILON * magnitude;
```

77

Boolean and relational operations

Logical operators

- More complex boolean expressions can be built by using logical operators, e.g., (assuming age an initialized int)

```
boolean test;
test = 13 >= age && age <= 17;
test = (13 >= age && age <= 17) || age >= 65;
test = !(13 >= age && age <= 17);
```

78

Boolean and relational operations

Logical operators

- In Java, the logical operators are as follows.

&&	conjunction (and)
||	disjunction (or)
!	not

79

Boolean and relational operations

Precedence in between logical operators

- Precedence is as with standard logic
  - ! higher precedence than &&
  - && higher precedence than ||
- Example (with p, q booleans)

  p && q || !p && r

  is interpreted as

  (p && q) || ((!p) && r)
- You can alter the interpretation by using parentheses.

80

Boolean and relational operations

Order of evaluation

- The && and || operators are evaluated left to right.
- Evaluation stops as soon as the answer can be determined
  - We call this lazy or short circuit evaluation.
- For example,

```
boolean test = d != 0 && n/d > 1;
```

avoids division by zero.

81

Boolean and relational operations

Flags

- It can be useful to declare (and initialize) boolean variables to represent the truth value of certain conditions.
- As examples (assuming age an int)

```
boolean senior = age >= 65;
boolean child = age < 13;
boolean discount = senior || child;
```

- As boolean variables are used in this fashion, we refer to them as flags.

82

Boolean and relational operations

Truth tables

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

83

Boolean and relational operations

Truth tables

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

84

Boolean and relational operations

Truth tables

p	**q**	**p && q**
true	true	true
true	false	false
false	true	false
false	false	false

p	**q**	**p || q**
true	true	true
true	false	true
false	true	true
false	false	false

p	**!p**
true	false
false	true

85

Outline

- Modules
- APIs and class use
- Input/output
- Boolean and relational operations
- **Software (re)engineering**

86

Software (re)engineering

Deployment: A call from our client

- The company that commissioned the MkChange program is very distressed to find out that it does not check to make sure that a non-negative value is supplied by the user.
- We tell them that they failed to provide us with a validation during the analysis phase of the software development cycle.
- They are not very amused by this comment; however, they give us a chance to rectify the situation.

87

Software (re)engineering

Requirements: Input validation

- Client tells us that if the amount entered is less than zero the program should
  - Print a message, ” Amount must be non-negative.”
  - and then exit.

88

Software (re)engineering

Design reconsidered

- Somehow, we want to verify that amount now has a value that is greater than or equal to zero when entered by the user,
- i.e., want

  amount >= 0

  to be true to continue with the computation
- or else (if the condition is false) send user error message and exit.

89

Software (re)engineering

Design reconsidered

- We find the following entry in the API for the ToolBox class of type.lib

Method Summary

static void crash (boolean condition, java.lang.String msg)
Test the passed condition and throw an exception if it is true.

90

Software (re)engineering

Implementation: DICO

- Because we adhered to good programming style, we know right where to go to correct the problem.

```
// Input
output.print(“Enter the amount in cents: ”);
amount = input.nextInt();
```

91

Software (re)engineering

Implementation: DICO

- Because we adhered to good programming style, we know right where to go to correct the problem.

```
// Input with validation
output.print(“Enter the amount in cents: ”);
amount = input.nextInt();
ToolBox.crash(amount<0, “Amount must be non-negative.”);
```

92

Software (re)engineering

Implementation: DICO

- Because we adhered to good programming style, we know right where to go to correct the problem.

```
// Input with validation
output.print("Enter the amount in cents: ");
amount = input.nextInt();
ToolBox.crash(amount<0, "Amount must be non-negative.");
```

- BTW, we also need to include a statement to gain access to the crash: import type.lib.ToolBox

93

Software (re)engineering

Finishing up

- We (re)test.
- We (re)deploy.
- …And await the next call from our client.

94

Summary

- Modules
- APIs and class use
- Input/output
- Boolean and relational operations
- Software (re)engineering

95