Inf1B

Java API

Perdita Stevens
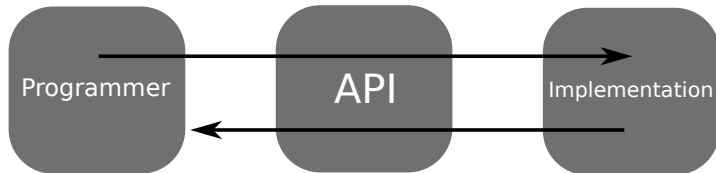adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics

Built-in Classes

The Java API / Class Library

Application Programming Interface

The interface between the user of the code and the implementation itself is called an Application Programming Interface (API).

Major Benefit: Underlying implementation can be changed (improved) without affecting the user of the API.

Java API

Some functionality is used often by most programs, e.g.

- ▶ Printing to the console: System.out. println ("Hi")
- ▶ Handling sequences of multiple characters:
  String msg = "Error: invalid value!"
- ▶ Generating a random number:
  Integer num = Integer. parseInt (args [0])
- ▶ etc.

To avoid the reinvention of the wheel over and over, a library with standard functionality and classes is provided for every programming language

In Java this is called the **Java API** or **Java Documentation**
https://docs.oracle.com/en/java/javase/11/docs/api/
index.html

Packages

Organising Classes

Things that need to be changed together
should live together.

But **Classes** are not enough.

Organising code

Java Version	Number of Classes in Library
11	4410
10	6002
9	6005
8	4240
7	4024
6	3793
5.0	3279
1.4.2	2723
1.3.1	1840

Organising code

Java Version	Number of Classes in Library
11	4410
10	6002
9	6005
8	4240
7	4024
6	3793
5.0	3279
1.4.2	2723
1.3.1	1840

A way of organising code on a higher level is needed, i.e. of organising classes.

Organising classes in packages
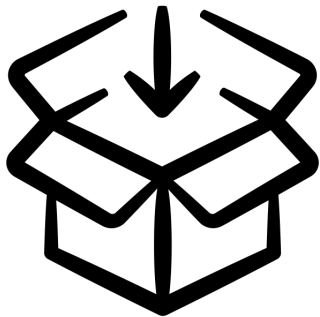
Created by Gregor Cresnar
from Noun Project

In Java, **packages** are used to organise classes.

Think of them as subfolders (which they usually are anyway).

Organising classes in packages

Consider for example `java.lang` which contains fundamental classes for using the language, e.g. `Integer`, `Maths`, `String`

or

`java.util` which contains various utility classes, e.g. `Arrays`, `Date`, `Scanner`

Naming Convention package names start with a lower case symbol and subpackages separated by '.'

Using Packages

Using a class from a package in your code, requires you to specify the entire name including the package prefix:

```java
public class DatePrinter {
  public static void main(String[] args) {
    java.util.Date today = new java.util.Date();
    System.out.println("Today's date is: "
                        + today.toString());
  }
}
```

Output

```
Today's date is:  Mon Nov 02 17:28:20 GMT 2020
```

Using Packages

To save you some writing work, you can import necessary classes. This allows you to skip the package prefix.

```java
import java.util.Date;
public class DatePrinter {
  public static void main(String[] args) {
    Date today = new Date();
    System.out.println("Today's date is: "
                       + today.toString());
  }
}
```

Import statements need to be outside of the class definition. You can also import all classes from a package:

import java.util.*

but this is (often considered) bad practice.

Using Packages

Static imports allow you to skip class identifiers for calling class methods or using static constants.

```java
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.SimpleDateFormat;
public class CalendarPrinter {
 public static void main(String[] args) {
  SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd HH:mm:ss");
  Calendar calendar = new GregorianCalendar(2019,1,15,13,24,56);
  int year       = calendar.get(Calendar.YEAR);
  int month      = calendar.get(Calendar.MONTH);
  int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);

  System.out.println(sdf.format(calendar.getTime()));
  System.out.println("year: " + year +
                     " month: " + month +
                     " dayOfMonth: " + dayOfMonth);
 }
}
```

Without static import.

Using Packages

Static imports allow you to skip class identifiers for calling class methods or using static constants.

```java
import static java.util.Calendar.*;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.SimpleDateFormat;
public class CalendarPrinter {
 public static void main(String[] args) {
  SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMM dd HH:mm:ss");
  Calendar calendar = new GregorianCalendar(2019,1,15,13,24,56);
  int year       = calendar.get(YEAR);
  int month      = calendar.get(MONTH);
  int dayOfMonth = calendar.get(DAY_OF_MONTH);

  System.out.println(sdf.format(calendar.getTime()));
  System.out.println("year: " + year +
                     " month: " + month +
                     " dayOfMonth: " + dayOfMonth);
 }
}
```

With static import.

Using Packages

I am using `Integer`, `String` and `Math` all the time but never need to import anything!

Using Packages

I am using `Integer`, `String` and `Math` all the time but never need to import anything!

All classes from the **java.lang** package are included automatically into every Java program.

Creating your own packages

You can create your own packages by using the package keyword.

```java
package com.dateapp.output;

import java.util.Date;
public class DatePrinter {
  public static void main(String[] args) {
    Date today = new Date();
    System.out.println("Today's date is: "
                       + today.toString());
  }
}
```
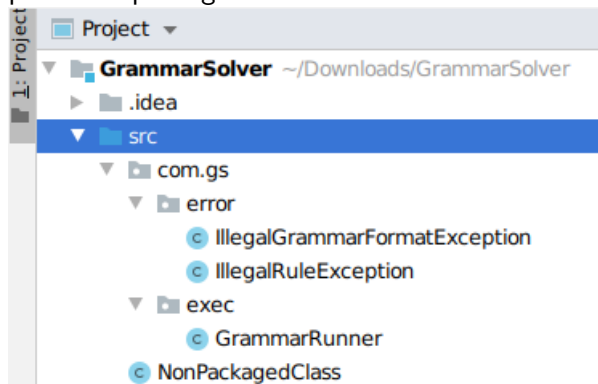
The package definition needs to go into the first line of your class document.

Also, make sure you put the underlying file in the correct subfolder.

Default package

The **default package** indicates that your source files are in no
particular package.

Namespace management

Packages maintain their own isolated namespaces

```
com.myapp.graphics.Utils
com.myapp.io.Utils
```

Classes with the same name can co-exist in the same program if they are in different packages.

Java API

With this knowledge, let's take another quick look at the API.
`https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

Strings

An example from the class library

String: basis for text processing

Underlying set of values: sequences of Unicode characters.
In Java Strings are immutable: none of the operations change the value.

```
public class String
```

	String(String s)	*create a string with same value as s*
char	charAt(int i)	*character at index i*
String	concat(String t)	*this string with t appended*
int	compareTo(String t)	*compare lexicographically with t*
boolean	endsWith(String post)	*does string end with post?*
boolean	equals(Object t)	*is t a String equal to this one?*
int	indexOf(String p)	*index of first occurrence of p*
int	indexOf(String p, int i)	*as indexOf, starting search at index i*
int	length()	*return length of string*
String	replaceAll(String a, String b)	*result of changing all as to bs*
String[]	split(String delim)	*result of splitting string at delim*
boolean	startsWith(String pre)	*does string start with pre?*
String	substring(int i, int j)	*from index i to index j − 1 inclusive*

http://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Typical String Processing Code

is the string a palindrome?

```java
public static boolean isPalindrome(String s) {
    int N = s.length();
    for (int i = 0; i < N / 2; i++) {
        if (s.charAt(i) != s.charAt(N - 1 - i))
            return false;
    }
    return true;
}
```

extract filenames and extensions from a command-line argument

```java
String s = args[0];
int dot = s.indexOf(".");
String base = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

print all lines from standard input containing the string "info"

```java
while (!StdIn.isEmpty()) {
    String s = StdIn.readLine();
    if (s.contains("info"))
        System.out.println(s);
}
```

print all ac.uk URLs in text file on standard input

```java
while (!StdIn.isEmpty()) {
    String s = StdIn.readString();
    if (s.startsWith("http://") && s.endsWith("ac.uk"))
        System.out.println(s);
}
```

Format Strings

How to gain more fine-grained control over print strings.

println can be Clunky

The student named 'Lee' is aged 18.

Using string concatenation

```
System.out.println("The student named '"
+ name
+ "' is aged "
+ age
+ ".");
```

String with Format Specifiers, 1

Target String

```
"The student named 'Lee' is aged 18."
```

String with Format Specifiers, 1

Target String

```
"The student named 'Lee' is aged 18."
```

String with Gaps

```
"The student named '_' is aged _."
```

String with Format Specifiers, 1

Target String

```
"The student named 'Lee' is aged 18."
```

String with Gaps

```
"The student named '_' is aged _."
```

String with Format Specifiers

```
"The student named '%s' is aged %s."
```

String with Format Specifiers, 1

Target String

```
"The student named 'Lee' is aged 18."
```

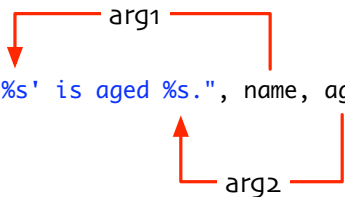## String with Gaps

```
"The student named '_' is aged _."
```

String with Format Specifiers

```
"The student named '%s' is aged %s."
```

- %s is a placeholder for a string.
- Called a format specifier.
- Each format specifier in a string gets replaced by an actual value.

String with Format Specifiers, 2


```
String.format("The student named '%s' is aged %s.", name, age);
```

arg1

arg2

String with Format Specifiers, 3

Define a Format String

```
String str =
String.format("The student named '%s' is aged %s.",
name, age);
System.out.println(str);
```

Output

The student named 'Lee' is aged 18.

printf, 1

Shorter version

```
System.out. printf ("The student named '%s' is aged %s.",
name, age);
```

Output

The student named 'Lee' is aged 18.

printf, 2

Convert char to String

```
System.out.printf("'%s' is for Apple.", 'A');
```

Output

'A' is for Apple.

printf, 2

Round to 2 decimal places

```
System.out.printf("The value of pi is %f", Math.PI);
System.out.printf("The value of pi is %.2f", Math.PI);
```

Output

```
The value of pi is 3.141593
The value of pi is 3.14
```

printf, 2

Round to 2 decimal places

```java
System.out.printf("The value of pi is %f", Math.PI);
System.out.printf("The value of pi is %.2f", Math.PI);
```

Output

```
The value of pi is 3.141593
The value of pi is 3.14
```

Include a newline

```java
System.out.printf("The value of pi is %f\n", Math.PI);
```

Code Documentation

Code Documentation

Providing well documented code is an essential skill of a software developer.

► Tell other developers how to use your code.
► Understand the workings of a complex algorithm more quickly.
► Find your way around your own code when you come back to it after some time.
► Supports the development process by helping you think through a given problem.

Types of Documentation

Comments within the code.

```java
public static int sum(int[] data) {
    int sum = 0;
    /* This loop
        iterates over
        each entry in
        the data array */
    for (int i = 0; i < data.length; i++)
    {
        // accumulate sum of each data entry
        sum += data[i];
    }
    return sum;
}
```

Improve clarity of specific parts of an algorithm or "activate" / "deactivate" specific code sections quickly.

Types of Documentation

Javadoc comments preceding methods and classes.

```
/**
 * First sentence of the comment should be a
 * summary sentence.
 * Documentation comment is written in HTML, so it can
 * contain HTML tags as well.
 * For example, below is a paragraph mark to separate
 * description text from Javadoc tags.
 * <p/>
 * @author Krishan Kumar
 */
public class Calculator {
    public static int sum(int[] data) {
        int sum = 0;
        ...
```

Describe the functionality and intended use of specific software
components.

Types of Documentation

Javadoc comments preceding methods and classes.

```java
/**
 * Calculates the sum of all entries in a given integer array.
 * Empty arrays are considered to have a sum of zero.
 *
 * @param data input array containing the data
 * @return sum of all values in given data
 * @throws NullPointerException if the array is null
 */
public static int sum(int[] data) {
    if (data == null)
        throw new NullPointerException("Data must not be null.");

    int sum = 0;
    for (int i = 0; i < data.length; i++)
        sum += data[i];
    return sum;
}
```

Use a contract-style specification between function author and function user which defines the delivered output for provided input.

Javadoc

@param Used in method comments. It describes a
        method parameter. The name should be the
        formal parameter name. The description
        should be a brief one line description of the
        parameter.

@return Used in method comments. It describe the
        return value from a method with the exception
        of void methods and constructors.

@throws Used in method comments. It indicates any
        exceptions that the method might throw and
        possible reasons for the occurrence of this
        exception.

Javadoc

Java provides a generator for API style documentations using javadoc entries in code.

Demo

How much commenting do I need to do?

How much commenting do I need to do?

javadoc every method, class and field/constant

within code **???**

Good Comments vs. Bad Comments

It is not always easy to decide if comments are useful or if more comments actually make the code less readable.

Let's consider some examples ...

```
// ...
```

Good Comments vs. Bad Comments

Don't write comments that are glaringly obvious from simply looking at the code.

```
return 1;  // returns 1
```

Good Comments vs. Bad Comments

Don't write comments that are glaringly obvious from simply
looking at the code.

```
return 1;   // returns 1
```

```
int[] data = {1, 2, 3, 4};

// print every entry in data
for (int i = 0; i < data.length; i++) {
  System.out.println(data[i]);
}
```

Good Comments vs. Bad Comments

Don't write comments that are glaringly obvious from simply looking at the code.

```
return 1;  // returns 1
```

```
int[] data = {1, 2, 3, 4};

// print every entry in data
for (int i = 0; i < data.length; i++) {
  System.out.println(data[i]);
}
```

Assume that the person reading your code understands Java.

Good Comments vs. Bad Comments

Don't write comments that are simply not true.

```
// always returns true
public static boolean isActive() {
  return false;
}
```

Good Comments vs. Bad Comments

Don't write comments that are simply not true.

```
// always returns true
public static boolean isActive() {
  return false;
}
```

This can actually become difficult and work intensive as soon as
your code starts changing over time.

Good Comments vs. Bad Comments

Avoid comments where you could make the code more clear by restructuring it and using helpful variable and method names.

```java
public static String get() {
  // Load the participants from the database
  Entry[] arr = db.getAll();

  // just get the participant's names
  String[] res = new String[arr.length];
  for(int i = 0; i < res.length; i++) {
    res[i] = arr[i].getName();
  }
  return res;
}
```

Good Comments vs. Bad Comments

Avoid comments where you could make the code more clear by
restructuring it and using helpful variable and method names.

```java
public static String[] getParticipants() {
  Person[] participants = database.getAllParticipants();

  String[] pnames = new String[participants.length];
  for(int i = 0; i < participants.length; i++) {
    pnames[i] = participants[i].getName();
  }
  return pnames;
}
```

Good Comments vs. Bad Comments

Avoid comments where you could make the code more clear by restructuring it and using helpful variable and method names.

```java
public static String[] getParticipants() {
  Person[] participants = database.getAllParticipants();

  String[] pnames = new String[participants.length];
  for(int i = 0; i < participants.length; i++) {
    pnames[i] = participants[i].getName();
  }
  return pnames;
}
```

You would call this self-documenting code

Good Comments vs. Bad Comments

Don't do any of this nonsense ...

```
// This code sucks, you know it and I know it.
// Move on and call me an idiot later
```

Good Comments vs. Bad Comments

Don't do any of this nonsense ...

```
// This code sucks, you know it and I know it.
// Move on and call me an idiot later
```

```
// magic, do not touch!
```

Good Comments vs. Bad Comments

Don't do any of this nonsense ...

```
// This code sucks, you know it and I know it.
// Move on and call me an idiot later
```

```
// magic, do not touch!
```

```
/* Class used to workaround Richard being
a f***ing idiot */
```

How much commenting do I need to do?

javadoc every method, class and field/constant

within code to explain why you are doing things a certain way, if that way is non-obvious

Consistent Coding Style

Not only documentation but also a consistent coding style improve your code quality.

- ▶ class, method and variable naming conventions
- ▶ spacing
- ▶ placement of brackets
- ▶ positioning of class elements
- ▶ ...

Consistent Coding Style

Not only documentation but also a consistent coding style improve your code quality.

- ▶ class, method and variable naming conventions
- ▶ spacing
- ▶ placement of brackets
- ▶ positioning of class elements
- ▶ ...

Consider the **Inf1B Coding Conventions** Document!

Third Party Libraries

A lot of library code is provided by other developers for you to use.

They are usually distributed as **jar** files.

Summary

- ► The Java language comes with a set of predefined classes wrapping up most often used functionality.
- ► Packages are used to organise classes by topic.
- ► Strings and String formatting are useful
- ► For high quality code, you should write documentation and comments (see **Inf1B Coding Conventions**)
- ► Third Party Libraries

Reading

Java Tutorial
Chapter 8 *Packages*
Chapter 9 *Numbers and Strings*

Inf1B Coding Conventions
Based on Objects First, *Appendix J*