Interfaces

Motivation for interfaces?

Already seen

Compatible use of different implementations of the same type

Just two examples:

- `TimerClass` and `AnotherTimerClass` are two implementations of `Timer`
- `ArrayList` and `LinkedList` are two implementations of `List`

Union types

Compatible use of different forms of values of the same type

Just two examples:

- rectangles, circles, squares are all shapes, but they are different
- an Abstract Syntax Tree is made of different nodes

The functional approach

Compatible use of different forms of shapes

Solution in OCaml with variant types

```
type shape = Square of float | Circle of float
      | Rectangle of float * float;;

let perimeter = function (* perimeter : shape -> float *)
    Square side -> 4.0 *. side
  | Circle ray -> 2.0 *. Float.pi *. ray
  | Rectangle (width,height) -> 2.0 *. (width +. height);;

let area = function (* area : shape -> float *)
    Square side -> side *. side
  | Circle ray -> Float.pi *. ray *. ray
  | Rectangle (width,height) -> width *. height;;
```

The object-oriented approach

Compatible use of different forms of shapes

Solution in Java with interfaces and classes

```
public interface Shape {
    double perimeter();
    double area();
}
public class Square implements Shape { // a square is a shape
   private double side;
   ...
   public double perimeter() { return 4 * this.side; }
   public double area() { return this.side * this.side; }
}
public class Rectangle implements Shape { // a rectangle is a shape
   private double width;
   private double height;
   ...
   public double perimeter() { return 2 * (this.width + this.height); }
   public double area() { return width * height; }
}
public class Circle  implements Shape { // a circle is a shape
   private double radius;
   ...
   public double perimeter() { return 2 * Math.PI * this.radius; }
   public double area() { return Math.PI * this.radius * this.radius; }
}
```

Functional versus object-oriented approach

Functional approach

- Code is structured by operation (perimeter and area in the example)
- Each operation uses pattern matching to deal with all forms of data (squares, rectangles and circles in the example)

Object-oriented approach

- Code is structured by data (`Square`, `Rectangle` and `Shape` in the example)
- Each class implements all operations (perimeter and area in the example)

Interfaces and instance methods

Example

```
public class TimerClass implements Timer {
    private int time = 60;
    ...
    public TimerClass(Timer otherTimer) { // which instance method is called?
        this.time = otherTimer.getTime();
    }
    public int getTime() { return this.time; }
    ...
}
public class AnotherTimerClass implements Timer {
    private int minutes = 1;
    private int seconds;
    ...
    public int getTime() { return this.seconds + 60 * this.minutes; }
    ...
}
TimerClass t1 = new TimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass(t1); // getTime of TimerClass is called
TimerClass t4 = new TimerClass(t2); // getTime of AnotherTimerClass is called
```

Interfaces and instance methods

Example 2

```
public class ShapeComparator {
    public int compare(Shape s1, Shape s2) {
        double area1 = s1.area(); // which instance method is called?
        double area2 = s2.area(); // which instance method is called?
        return area1 > area2 ? 1 : area1 == area2 ? 0 : -1;
    }
}
ShapeComparator c = new ShapeComparator();
assert c.compare(new Square(2), new Rectangle(4, 1)) == 0;
assert c.compare(new Square(2), new Circle(1)) > 0;
assert c.compare(new Circle(1), new Square(2)) < 0;
```

Important remark

The called instance method depends on the dynamic type of the target object

Reminder

Target object = the object on which the method is called = **`this`**

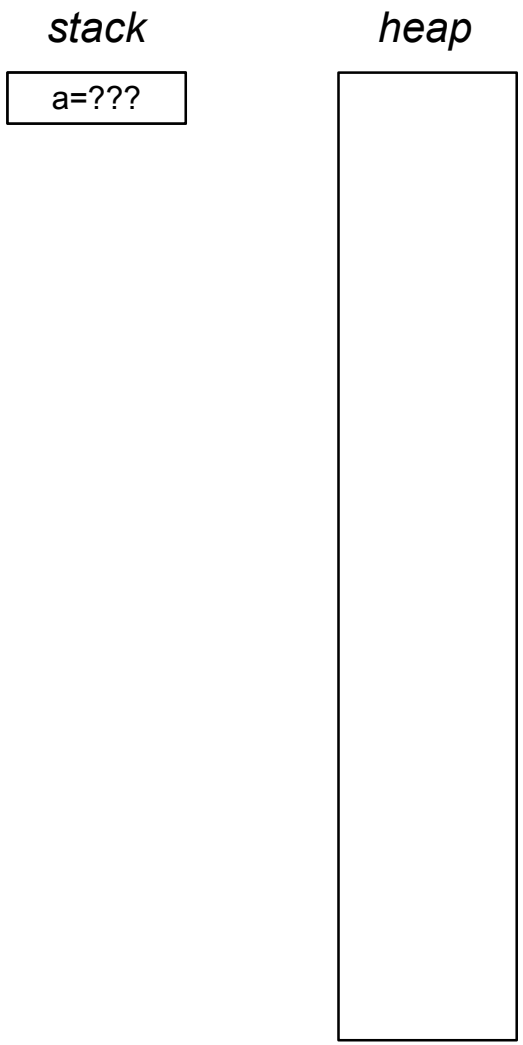
Arrays in Java

A first example

```
public class ArrayUtils {
    public static void init(int[] a) {
        // standard "for" syntax
        for (int i = 0; i < a.length; i++)
            a[i] = i + 1;
    }
    public static int sum(int[] a) {
        int sum = 0;
        // more compact "for-each" syntax
        for (int el : a)
            sum += el;
        return sum;
    }
}
...
int[] a; // a will refer to an array of integers
a = new int[10]; // an array of ten integers is dynamically created
assert a.length == 10;
assert ArrayUtils.sum(a) == 0; // default value for int arrays is 0
ArrayUtils.init(a);
assert ArrayUtils.sum(a) == 55;
```

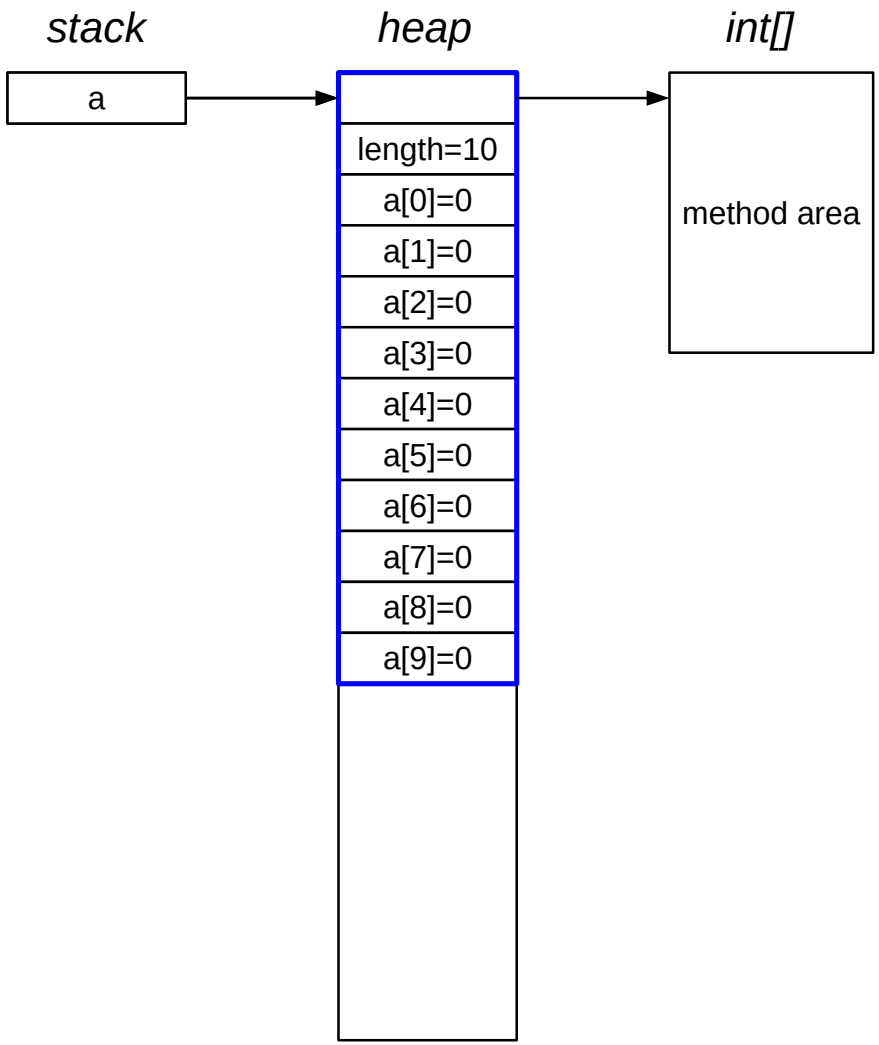
Example with memory model

```
int[] a; ⇐
a = new int[10];
ArrayUtils.init(a);
```



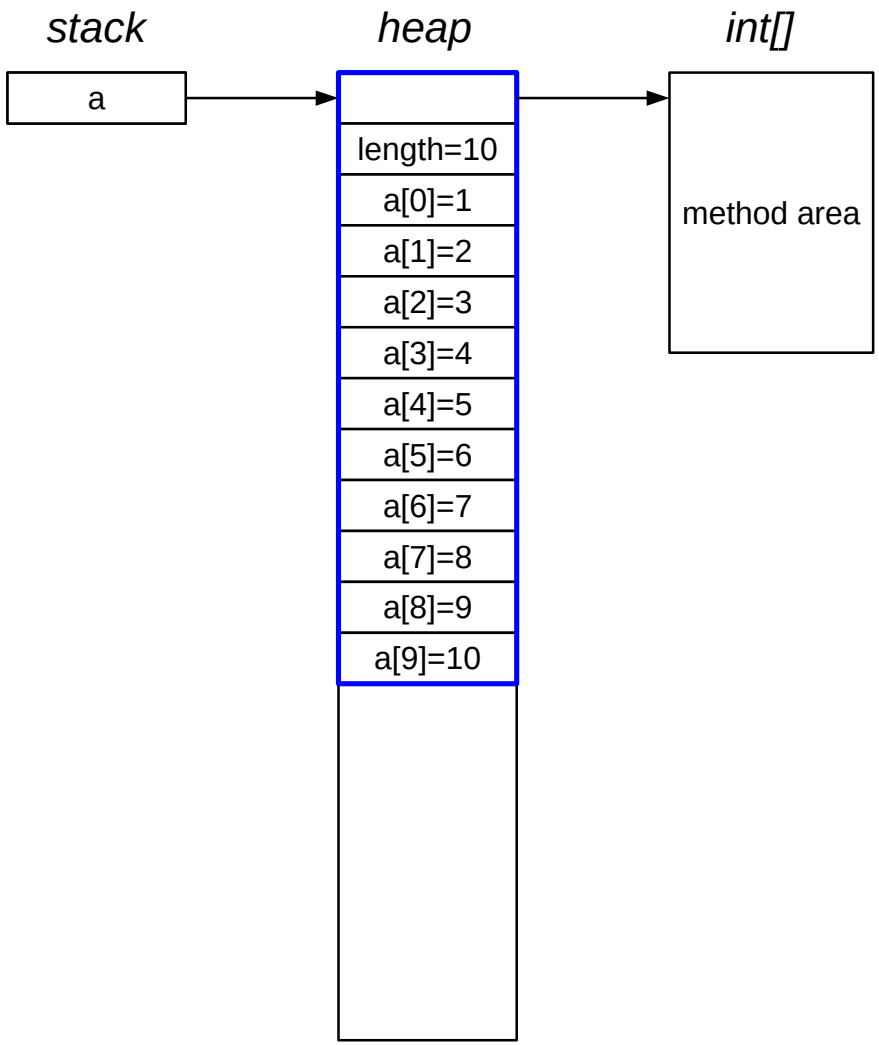
Example with memory model

```
int[] a;
a = new int[10]; ⇐
ArrayUtils.init(a);
```



Example with memory model

```
int[] a;
a = new int[10];
ArrayUtils.init(a); ⇐
```



Arrays in Java

Details

- arrays are special modifiable objects (=reference types)
- array components: indexed instance variables with no name
- arrays can only be created dynamically
- at creation time the length (=number of components) is specified
- the length is a non negative integer and cannot change over time
- `length` is a final instance variable of the array
- components are initialized with their default values
- components are referenced with indices from 0 to `length-1`
- `T[]` is the type of arrays with component type `T`

Arrays with object components

Example

```
String[] a = new String[3];
for(String el : a)
    assert el == null;
a[0] = "zero";
a[1] = "one";
a[2] = "two";
```

Array initialization

Example

```
public class ArrayUtils {
    public static int sum(int[] a) {
        int sum = 0;
        for (int el : a)
            sum += el;
        return sum;
    }
}
int[] a = { 1, 2, 3 }; // initializer usable at declaration time
assert ArrayUtils.sum(a) == 6;
assert ArrayUtils.sum(new int[] { 1, 2, 3, 4 }) == 10; // initializer
    usable at creation time
```

Remark

Initializers do not prevent dynamic array creation

Multi-dimensional arrays

Example

```
int[][] mat1 = new int[3][2]; // a matrix 3x2
assert mat1.length == 3;
for (int[] row : mat1) {
    assert row.length == 2;
    for (int el : row)
        assert el == 0;
}
int[][] mat2 = new int[3][]; // the last size can be omitted, each row is null
assert mat2.length == 3;
for (int[] row : mat2)
    assert row == null;
// an array with variable length rows
int[][] mat3 = { { 1, 1 }, { 1, 2, 1 }, { 1, 3, 3, 1 } };
```

Remarks

- multi-dimensional arrays are just arrays of arrays
- dimension can be larger than 2

Main methods in Java

In a nutshell

- execution of a Java program can only starts from a class with a main method
- a main method must always have this form:

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

- command-line arguments are passed to `arg` as an array of strings

Standard output in Java

In a nutshell

- `System.out` is the standard output
- `System` is a useful predefined class (as `String`)
- `System.out` is a final class variable of `System`
- `System.out` has type `PrintStream`
- `PrintStream` is a class of the predefined I/O Java library

Useful overloaded instance methods of `PrintStream`

```
void println()
void println(boolean b)
void println(char c)
void println(char[] s)
void println(double d)
void println(float f)
void println(int i)
void println(long l)
void println(Object obj)
void println(String s)
```

```
void print(boolean b)
void print(char c)
void print(char[] s)
void print(double d)
void print(float f)
void print(int i)
void print(long l)
void print(Object obj)
void print(String s)
```

Basic I/O in Java via argument passing

Program example

```
public class PrintArguments {
    public static void main(String[] args) {
        for (String s : args)
            System.out.println(s);
    }
}
```

Example of execution

```
$java PrintArguments one two three
one
two
three
```

Large-scale modularization features

Two main features at two different levels

- Modules contain/export logically related packages/sub-packages (since Java 9)
- Packages and sub-packages contain/export logically related classes

Remark

For simplicity Java projects will be only based on the *unnamed module*

Packages

Main package features

- public classes can be accessed from outside, non public classes can only be accessed from within their package
- accessible hierarchical namespaces, they can contain subpackages
- defined by several compilation units (files in the same directory)

Simple and fully qualified type names

Example: package `java.io` contains class `PrintStream`

- `PrintStream` is the simple name, used inside `java.io`
- `java.io.PrintStream` is the fully qualified name, used outside `java.io`

Compilation units

Example

```
/* file ColoredLine.java must be placed in directory shapes */
package shapes;

import java.awt.Color; // Color is an abbreviation for java.awt.Color

class Point { // not visible outside shapes
    ...
}

public class ColoredLine {
    private Point a;
    private Point b;
    private Color color = Color.BLACK;
    ...
    public Color getColor() { return this.color; }
    public void setColor(Color color) { this.color = color; }
}
```

Compilation units

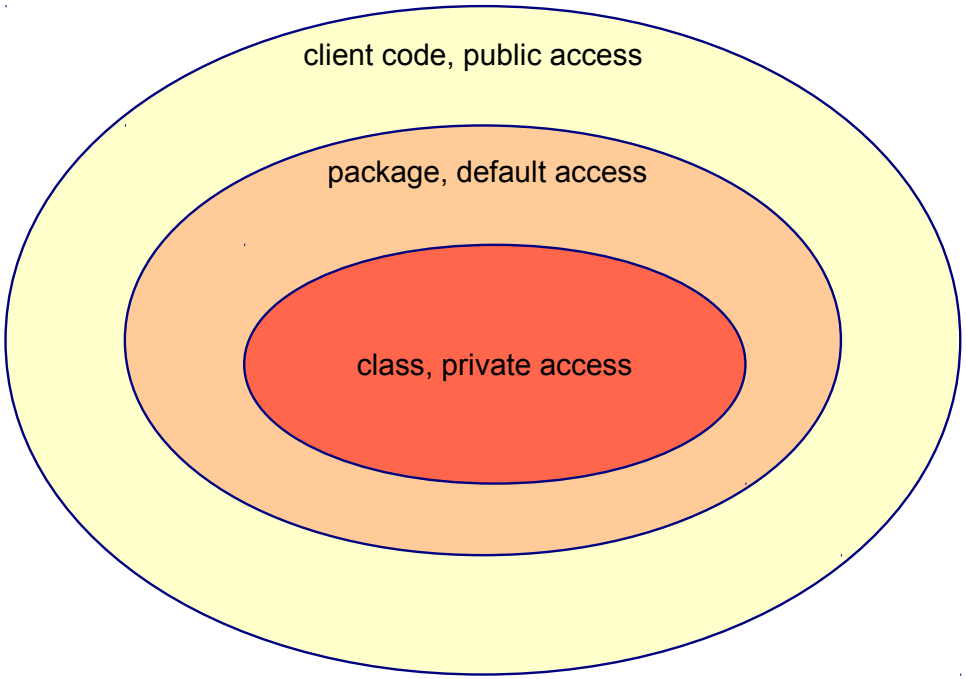
What is a compilation unit?

A compilation unit consists of three parts

- **`package`** declaration.
  If not present, the unit is part of an unnamed package
- **`import`** declarations
- declarations of top level classes

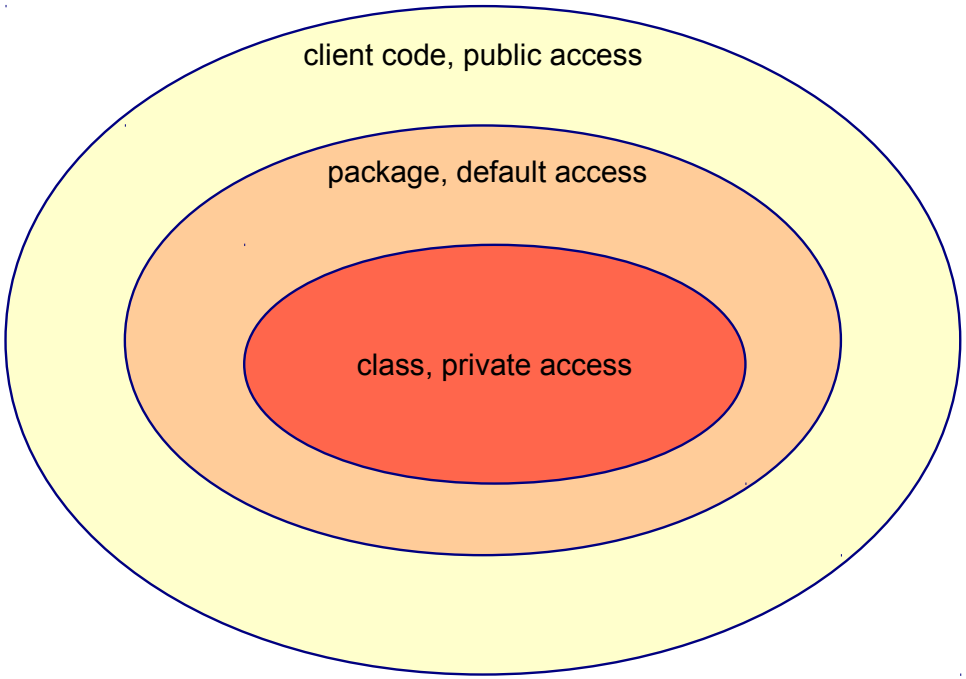
Package access

```
package p; // file C.java, directory p
public class C {
    void m() { // no access modifier implies package access
        /* ... */
    }
}

package p; // file Test.java, directory p
public class Test {
    public static void main(String[] args) {
        C c = new C();
        c.m(); // ok
    }
}
```



Package access

```
package p; // file C.java, directory p
public class C {
    void m() { // no access modifier implies package access
        /* ... */
    }
}

package q; // file Test.java, directory q
public class Test {
    public static void main(String[] args) {
        p.C c = new p.C();
        c.m(); // error
    }
}
```



Package access

Remark

Package is the default access

- `private` modifier: only accessible in the class
- `public` modifier: accessible everywhere the class is accessible
- no modifier: only accessible in the package

Packages and subpackages

Main rules

- packages are hierarchical namespaces: declarations in the same package must have distinct names, declarations in different packages can have equal names
  - Example: `java.io.InputStream` and `org.omg.CORBA.portable.InputStream` are two different classes. The former is declared in package `java.io`, the latter in `org.omg.CORBA.portable`, which is a subpackage of `org.omg.CORBA`
- a package or a subpackage can contain just subpackages
- the name of a (sub)package reflects a file system path

Remark

there is no special access relationship between a package and its subpackages

API Packages and subpackages

Documentation

- documentation on the Java API available on the official web site
  `https://docs.oracle.com/en/java/javase/13/docs/api/index.html`
- documentation also accessible through the IDEs (Eclipse, IDEA)
- we will consider only packages of the `java.base` module

Type imports

Useful feature to refer a class of another package with its simple name

Remarks

1. all public classes in package `java.lang` of module `java.base` are automatically imported
2. useless imports are ignored (example: importing a class of the same package)

Single imports and on demand imports

- **`import`** `java.util.Scanner;`
  the single type `Scanner` is imported
- **`import`** `java.util.*;`
  all public types of `java.util` are imported as needed
- single imports take precedence in case of conflicts
- single imports must avoid name conflicts

Static imports

Static imports are used for abbreviating names of static variables and methods

Single imports and on demand imports

```
import static java.lang.System.out;
```

- the single static variable `out` is imported from `System`
- the abbreviated form `out` can be used instead of `System.out`

```
import static java.lang.System.*;
```

- all accessible static members of `System` are imported as needed