JAVA

Classes

Class definition

- complete definition

```
[public] [abstract] [final] class Name
  [extends Parent]
  [impelements ListOfInterfaces] {
    ... // class body
}
```

- **`public`** – public class
- **`abstract`** – no instance can be created
- **`final`** – class cannot be extended

Constructor

- constructor
  - object initialization
- declaration
  - the same name as the class
  - no return type
  - modifier – only visibility
  - several constructors
    - with different arguments
    - selected by arguments of `new`

```
class MyClass {
  int value;
  public MyClass() { value = 10; }
  public MyClass(int v) { value = v; }
}
```

Object removal

- garbage collector

- `finalize()` method
  - present in every class
  - called before object's removal
  - **it is not a destructor like in other languages**
  - not known when it is called
  - calling is not guaranteed
    - object need not be removed by garbage collecting
      - e.g. at the end of the program
  - calls of `finalize()` are not chained

Deprecated since Java 9

Initialization of fields

- in constructor
  or
- direct

```
class MyClass {
  int a = 5;
  float b = 1.2;
  MyClass2 c = new MyClass2();
  int d = fn();
  int e = g(f);  // error!
  int f = 4;
  ...
}
```

Initialization: `static`

- just once
- before first access or before first instance of a class is created
- direct

```
static int a = 1;
```

- static initializer

```
class MyClass {
  static int a;
  static {
    a = 10;
  }
  ...
}
```

Initialization: "non-static"

- similar to `static` initializer
- necessary for initialization of *anonymous inner classes*

```
class MyClass {
   int a;
   int b;
   {
      a = 5;
      b = 10;
   }
   ...
}
```

Classes: inheritance

- parent specification – **`extends`** *`ParentName`*
- single inheritance
  - single parent only
- class **`java.lang.Object`**
  - each class inherits from this class
    - directly or indirectly
  - the only class without parent
- multiple inheritance only via **Interface**s

Polymorphism

- polymorphism ~ inheritance
- cast
  - automated – child to parent

```
class A { /*...*/ }
class B extends A { /*...*/ }

A a = new B();
Object o = a;

B b = (B) o;
```

Polymorphism – constructor

- constructor of the parent
  - `super()`
- other constructor of the same object
  - `this()`
- calling other constructors
  - only as the first statement and just once
- parent's constructor is called always
  - even if not explicitly called
  - exception – `this()`
- class without constructor declared
  - has default constructor
    - calls `super()` only

java.lang.Object

```
Object clone()
boolean equals(Object obj)
void finalize()
Class<?> getClass()
int hashCode()
void notify()
void notifyAll()
String toString()
void wait()
void wait(long timeout)
void wait(long timeout, int nanos)
```

Classes: visibility of members

- must be specified for each member
- fields and methods
  - **`public`**
    - from everywhere (if the class is also visible)
  - **`protected`**
    - from the same package and children
  - **`private`**
    - just from the same class
  - without a visibility modifier
    - from the same package
- holds within a single module

Classes: other modifiers

- **`final`**
  - field
    - constant
    - must have initializer
    - after initialization cannot be changed
  - method
    - cannot be overridden in children
- **`transient`**
  - field
  - does not belong to a persistent state of the object
- **`volatile`**
  - field
  - non-synchronized access of multiple threads
  - no optimization can be perfomed

Classes: modifiers of methods

- **`abstract`**
  - no method body
  - the class must be also **`abstract`**
    - no instance can be created
  - method body – semicolon
- **`synchronized`**
  - calling thread must obtain a lock on the called object (or the class in the case the method is **`static`**)
- **`native`**
  - native method
  - implementation directly in native code for a particular platform (as an external library)
  - method body – semicolon
- **`static`**
  - see the previous lecture

Classes: method modifiers

- no modifier **`virtual`**
- all methods are virtual
  - static methods ***are not*** virtual

```
public class A {
  public void foo() {
    System.out.print("A");
  }
}

public class B extends A {
  public void foo() {
    System.out.print("B");
  }
}
....
A a = new B();
a.foo();     // prints out B
```

```
public class As {
  public static void foo() {
    System.out.print("A");
  }
}

public class Bs extends As {
  public static void foo() {
    System.out.print("B");
  }
}
....
A a = new B();
a.foo();     // prints out A
```

Static methods

- static methods are called on a class
  - do not belong to any object

```
class As {
  public static void foo() { ........ }
}

As.foo();
```

- they can be “called” on an object (a class instance); but in reality only a type of the reference is taken
  - value of the object is ignored
  - type (and thus a method to be called) is determined at compile time
    - see the previous slide

this

- reference to the object of the executed method
- can be used in methods and initializers only

```
public class MyClass {
   private int a;
   public MyClass(int a) {
      this.a = a;
   }
}
```

super

- access to members of the direct parent
- in the case S is direct parent of C

  `((S) this).name` ~ `super.name`
- `super.super` cannot be used

```
class T1 { int x = 1; }
class T2 extends T1 { int x = 2; }
class T3 extends T2 {
   int x = 3;
   void test() {
     System.out.println(x);                 // 3
     System.out.println(super.x);           // 2
     System.out.println(((T2)this).x);      // 2
     System.out.println(((T1)this).x);      // 1
   }
}
```

super

- `super` can be used with methods too
- WARNING – casting `this` does not work
  - a code can be compiled but the same method will be called recursively

```
class TX1 {
  public void foo() { /*…*/ }
}
class TX2 extends TX1 {
  public void foo() { /*…*/ }
}
public class TX3 extends TX2 {
  public void foo() {
    ((TX1) this).foo();
    System.out.println("TX3.foo()");
  }
}
```

Interfaces

Interface

- only interface
- no implementation
  - since Java 8, there can be an implementation
- can contain
  - method headers
  - fields
  - inner interfaces

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove();
}
```

Interface: fields

- implicitly they are `public`, `static` and `final`
- must be initialized
- `super` and `this` cannot be used in initialization

```
public interface Iface {
   int a = 5;
   String s = "hello";
}
```

Interface: methods

- without implementation
  - implicitly **`abstract`** and **`public`**
  - cannot be
    - **`synchronized`**
    - **`native`**
    - **`final`**
- **`default`** methods
  - since Java 8
  - contains implementation
  - intended for extending interfaces
- **`static`** methods
  - since Java 8
  - the same as the static methods in classes

Interface: inheritance

- multiple inheritance

```
interface Iface1 { ... }
interface Iface2 { ... }

interface Iface3 extends Iface1, Iface2
  { ... }
```

Classes and interfaces

- classes implement interfaces

```
public interface Colorable {
  void setColor(int c);
  int getColor();
}
public class Point { int x,y; }
public class ColoredPoint extends Point
                          implements Colorable {
  int color;
  public void setColor(int c) {
    color = c; }
  public int getColor() { return color;}
}

Colorable c = new ColoredPoint();
```

Classes and interfaces

- a class must implement all methods of its interfaces except the `default` methods
  - not true for `abstract` classes
- a single method in a class can implement several interfaces

```
interface A { void log(String msg); }
interface B { void log(String msg); }

public class C implements A, B {
  public void log(String msg) {
    System.out.println(msg);
  }
}
```

Interfaces and default methods

- the implementation in a class has always precedence over the implementation in interfaces

- if implementing two interfaces with the same **`default`** method, then the method has to be implemented in the class
  - otherwise the class cannot be compiled

```
interface If1 {
  default void foo() {…}
}
```

```
interface If2 {
  default void foo() {…}
}
```

```
class Mixed implements
                If1, If2 {
}
```

cannot be compiled

Interfaces and default methods

- it is forbidden to define a default method for a public method from java.lang.Object

```
interface Iface {
  public default boolean equals(Object obj) {
    return false;
  }
}
```

- the implementation in a class has always precedence over the implementation in interfaces
  - even an inherited one

Interfaces and default methods

```
interface If1 {
  default void foo() {
    System.out.println(“interface”);
  }
}
```

```
class A {
  public void foo() {
    System.out.println(“class”);
  }
}
```

```
class B extends A implements If1 {
   public static void main(String[] args)
     B b = new B();
     b.foo();   // -> “class”
   }
}
```

Java

Arrays

Array definition

- array ~ object
- variable ~ reference

```
int[]      a;     // array
short[][] b;     // 2-dimensional array
Object[]  c,     // array
          d;     // array
long      e,     // non-array
          f[];   // array
```

Array initialization

- "static"

```
int[] a = { 1, 2, 3, 4, 5 };
char[] c = { 'h', 'e', 'l', 'l', 'o' };
String[] s = { "hello", "bye" };

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

Array initialization

- dynamic

```
int[] array = new int [10];
float[][] matrix = new float[3][3];
```

- just several dimensions can be specified
  - but first ones
  - empty brackets for the rest

```
float[][] matrix = new float[3][];
for (int i=0;i<3;i++)
  matrix[i] = new float [3];

// wrong
int[][][][] a = new int[3][][3][];
```

Array initialization

- "non-rectangular" array

```
int a[][] = { {1, 2}, {1, 2, 3}, {1, 2, 3,
  4, 5} };


int b[][] = new int [3][];
for (int i=0; i<3; i++)
  b[i] = new int [i+1];
```

Array initialization

- no constructor is called
- elements in the created array (using new) – default values
  - references – null
  - int – 0
  - ...
- expressions in array creation (**`new`**) – fully evaluated from left

```
int i = 4;
int ia[][] = new int[i][i=3];
// array 4x3
```

Access to array

- `array[index]`
- indexes – always `0..length-1`
- bounds always checked
  - cannot be switched off
  - exception thrown for out of bounds access
    `ArrayIndexOutOfBoundsException`
- array length – field **`length`**

```
int[] a = { 1, 2, 3 };
for (int i=0; i < a.length; i++) {
   ....
}
```

Array ~ object

- `int[] intArray = new int [100];`
- `String[] strArray = new String [100];`

- array is object

```
Object o1 = strArray;    // OK
Object o2 = intArray;    // OK
```

- but

```
Object[] oa1 = strArray; // OK
Object[] oa2 = intArray; // error
```

Array ~ object

```
Object[] oa = new Object [2];
oa[0] = new String("hello");
oa[1] = new String("world");

String[] sa1 = oa;  // error

String[] sa2 = (String[]) oa;
   // error too
   // can be compiled but run-time error
```