Enum
Enumerations

- <= Java 1.4
  
  ```java
  public static final int COLOR_BLUE = 0;
  public static final int COLOR_RED = 1;
  public static final int COLOR_GREEN = 2;
  ```

- **possible problems**
  - type (un)safety
  - no namespace
  - constants hard-compiled in clients
  - only numbers when printed
public enum Color { BLUE, RED, GREEN }
...
public Color clr = Color.BLUE;

• “normal” class
  - can have fields, methods, even the main method
  - subclass of java.lang.Enum
  - for each value – single instance
    • public static final field
    • protected constructor
“Enum without enum”

- how to implement enum in Java 1.4
  - (and how enums are implemented)

```java
class Color {
    private int value;

    public static final Color RED = new Color(0);
    public static final Color GREEN = new Color(1);
    public static final Color BLUE = new Color(2);

    protected Color(int v) {
        value = v;
    }
    ...
}
```
public abstract class Enum <E extends Enum<E>> { ... }

- methods
  - String name()
  - int ordinal()

- each enum has the method values()
  - returns an array with all enum's values

public Colors clr = Colors.BLUE;
System.out.println(clr);  →  BLUE
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    ...

    private final double mass;
    private final double radius;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
Fields and methods

- example

```java
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    double eval(double x, double y) {
        switch (this) {
            case PLUS:   return x + y;
            case MINUS:  return x - y;
            case TIMES:  return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```
Fields and methods

- abstract methods
- particular implementations with each of the values

```java
public enum Operation {
    PLUS { double eval(double x, double y) { return x+y; }},
    MINUS { double eval(double x, double y) { return x-y; }},
    TIMES { double eval(double x, double y) { return x*y; }},
    DIVIDE { double eval(double x, double y) { return x/y; }};

    abstract double eval(double x, double y);
}
```
Java

Variable number of arguments
● „three dots“
● only as the last argument
● either an array or list of arguments can be passed
● in the method, available as an array

```java
void argtest(Object... args) {
    for (int i=0; i < args.length; i++) {
        System.out.println(args[i]);
    }
}
argtest("Hello", "how", "are", "you");
argtest(new Object[] {"Hello", "how", "are", "you"});
```

● methods printf
  - `System.out.printf("%s %d\n", user, total);`
• Are the calls equivalent?

argtest("Ahoj", "jak", "se", "vede");
argtest(new Object[] {"Ahoj", "jak", "se", "vede"});
argtest((Object) new Object[] {"Ahoj", "jak", "se", "vede"});

a) Yes, all of them
b) Only 1. and 2.
c) Only 2. and 3.
d) Each of them will print something different
Annotations
Annotations

- (metadata)
- since Java 5
- allow attaching information to elements of code (to classes, methods, fields,...)
  - in general, can be used in the same places as visibility modifiers
    - but also elsewhere
- written as `@JmenoAnotace`
- own annotations can be created
  - can be specified, where can be used, how can be used,....
- predefined annotations in the package java.lang
  - `@Deprecated`
  - `@Override`
  - `@SuppressWarnings`
Annotations

- can have arguments

```java
@Deprecated(since="1.2", forRemoval=true)
```

- arguments can have default values
  - i.e., can be used without argument value
  ```java
  @Deprecated
  ```

- where can be used
  - classes, fields, methods …
  - method arguments, packages
  - type usage

- can restricted in the annotation definition
Predefined annotations

- @Override
  - marks a method that overrides the method from a parent
  - in a case that nothing is overridden => the compiler will not compile the class
  - usage is optional (but strongly recommended)

```java
class A {
    public void foo() {}
}
class B extends A {
    @Override
    public void foo() {}
}
class C implements Ice {
    @Override
    public void foo() {}
}
class D {
    public void foo() {}
}
class E extends D {
    @Override
    public void bar() {}
}
```
Predefined annotations

- @Deprecated
  - marks API that programmers are discouraged from using
    - replacement of the javadoc tag @deprecated
    - if used => warning when compiled
  - arguments
    - String since
    - default ""
    - boolean forRemoval
    - default false
Predefined annotations

• @SupressWarnings
  – suppress warnings during compilation
  – argument – kinds of suppressed warnings
    • String[] value

• supported kinds depend on a compiler
• always available kinds
  – unchecked – warning for “improper” usage of generics
  – deprecation – warning when deprecated elements are used

  – e.g. @SuppressWarnings(“unchecked“)
    @SuppressWarnings({"unchecked“, “deprecation“})
Lambda expressions
Motivation

- event handling in GUI
- a comparator implementation
- a thread implementation
- ...
  - commonly using an anonymous inner class

```java
interface Comparator<T> {
    int compare(T o1, T o2);
}

class Arrays {
    ...
    void sort(T[] a, Comparator<T> c);
}

Arrays.sort(array, new Comparator<AClass>() {
    public int compare(AClass o1, AClass o2) {
        return o1.x - o2.x;
    }
});
```
Motivation

- the previous example using a lambda expression

```java
Arrays.sort(array, (o1, o2) -> o1.x - o2.x);
```

- informally:
  an lambda expression ~ a block of code with parameters

- since Java 8
Functional interface

- where can be the lambda expressions use?

where an object of an interface with a single abstract method is expected

= functional interface

- a lambda expression = an instance of a functional interface

- but
  a lambda expression does not contain information about which functional interface it is implementing
Functional interface

```java
interface Predicate<T> {
    default Predicate<T> and(Predicate<? super T> other);
    static <T> Predicate<T> isEqual(Object targetRef);
    default Predicate<T> negate();
    default Predicate<T> or(Predicate<? super T> other);
    boolean test(T t);
}
```

- is it functional interface?

  yes
  only a single **abstract** method
Type of a lambda expression

• the same lambda expression can assigned to different interfaces
  Runnable r = () -> {};  
  AutoCloseable r = () -> {};  

public interface Runnable {
  void run();
}

public interface AutoCloseable {
  void close();
}
Type of a lambda expression

- lambda expressions are objects
  
  ```java
  Runnable r = () -> {};  
  Object o = r;
  ```

- but
  
  lambda expressions cannot be (directly) assigned to the Object type
  
  ```java
  Object r = () -> {};  
  ```

  - as Object is not a functional interface
Lambda expression syntax

- a comma-separated list of parameters in parentheses
  - types can be omitted
  - parentheses can be omitted if there is only one parameter
- “arrow” ->
- body
  - single expression
    - return can be omitted
    - no braces
      - cannot be omitted if return is used
  - block
    - in curly braces
Examples of lambda expressions

- \((\text{int } \, x, \, \text{int } \, y) \rightarrow x + y\)
- \((x, \, y) \rightarrow x - y\)
- \((\_\,) \rightarrow 42\)
- \((\text{String } \, s) \rightarrow \text{System.out.println}(s)\)
- \(x \rightarrow 2 \times x\)
- \(c \rightarrow \{ \text{int } s = c.\text{size}(); c.\text{clear}(); \text{return } s; \} \)
Functional interface

- [@FunctionalInterface](#)
  - annotation
  - to mark a functional interface
    - usage is not mandatory
      - similarly to [@Override](#)
References to methods

- **String::valueOf**
  - a reference to a static method
  - equivalent to: `x -> String.valueOf(x)`

- **Object::toString**
  - a reference to a non-static method
  - equivalent to: `x -> x.toString()`

- **x::toString**
  - a reference a method of a particular object
  - equivalent to: `() -> x.toString()`

- **ArrayList::new**
  - a reference to a constructor
  - equivalent to: `() -> new ArrayList<>()`
**Lambda expressions**

- Lambda expressions do not add a new scope of variable visibility

```java
Path first = Paths.get("/usr/bin");
Comparator<String> comp = (first, second) ->
    Integer.compare(first.length(), second.length());
```

- *this* in a lambda expression refers to *this* of a method, in which the lambda expression is created

```java
class Application {
    public void doWork() {
        Runnable runner = () ->
            {System.out.println(this.toString());};
    }
}
```
Lambda expr. compilation

public class AClass {
    ...
    public void foo(AClass[] array) {
        Arrays.sort(array, new Comparator<AClass>() {
            public int compare(AClass o1, AClass o2) {
                return o1.x - o2.x;
            }
        });
    }
}

• but

public class AClass {
    ...
    public void foo(AClass[] array) {
        Arrays.sort(array, (o1, o2) -> o1.x - o2.x);
    }
}
java.lang.Object
Methods

- clone
- equals
- finalize
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait
equals

- boolean equals(Object obj)
  - be aware about the signature
  - defined with the parameter type Object
  - if overridden the parameter Object must be kept
  - example

```java
class Complex {
    long x, y;
    public boolean equals(Object obj) {
        if (obj instanceof Complex) {
            Complex c = (Complex) obj;
            if (c.x == x && c.y == y) {
                return true;
            }
        }
        return false;
    }
}
```
equals

• ideal to declare the method with @Override
  – @Override public boolean equals(Object obj)
• if defined with another type, the method is overloaded but not overridden
  class Complex {
    long x,y;
    public boolean equals(Complex obj) {
      ...
    }
  }  
  – the class contains two method equals
hashCode

- `int hashCode()`  
- hash code of the object  
- used e.g. in the `java.util.Hashtable` and others  
- for the same object must always return the same value  
  - the value need not to be the same in different runs of a program  
- if two objects are equals (by the `equals` method), then the `hashCode` must be the same value  
- two different objects need not to have a different `hashCode`  
  - but it is desirable
clone

- Object clone() throws CloneNotSupportedException
- creates a copy of the object
- must hold
  \[x.clone() \neq x\]
- should hold
  \[x.clone().equals(x)\]
- the class must implement the interface Cloneable
  - otherwise the method throws CloneNotSupportedException
- arrays “implement” the Cloneable
- shallow copy of objects
  - i.e. fields are not cloned
  - for different behavior, the method should be overridden
clone

- overriding clone
  - typical implementation
    * but not mandatory
      ```java
      protected Object clone() {
          Object clonedObj = super.clone();
          ....
          return clonedObj;
      }
      ```
  - after cloning it holds:
    ```java
    a.clone() != a
    a.clone().equals(a)
    ```
**toString**

- returns textual representation of an object
- default
  - `getClass().getName() + '@' + Integer.toHexString(hashCode())`
- should be overridden

```java
class MyClass {
    ....
}
...
MyClass o = new MyClass();
System.out.println(o);  // toString() is called
```