Architecture of software systems

Lecture 2: Design patterns

David Šišlák

david.sislak@fel.cvut.cz
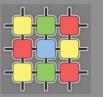
» thread-safe object
» all fields are **final**
(and often **private**)
» no "setters"
» block changes by
finalization of the class
» block override
use private constr. and
static factories
» don't need a copy constr.
» no clone needed
» hashCode can use lazy
initialization and cache
» no defensive copy when
used as a field
» suitable for map and set
elements

24/05/17

```java
final public class ImmutableRGB {

    //Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255
                || green < 0 || green > 255
                || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red, int green, int blue, String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }



    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
                "Inverse of " + name);
    }

}
```
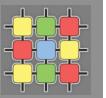
Immutable object - String

» special behavior of concatenation operator (+)

```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```
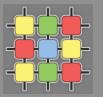
Immutable object - String

» special behavior of concatenation operator (+)

  » implemented through the **StringBuilder** and its *append* method

```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```

```
StringBuilder temp = new StringBuilder(a);
temp.append(b);
temp.append(c);
String q = temp.toString();
```

» but what about?

```
String q = a;
q += b;
q += c;
```

» special behavior of concatenation operator (+)

    » implemented through the **StringBuilder** and its *append* method

```
String a = "a";
String b = "b";
String c = "c";

String q = a + b +c;
```

```
StringBuilder temp = new StringBuilder(a);
temp.append(b);
temp.append(c);
String q = temp.toString();
```
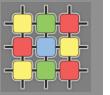
» but what about?

```
String q = a;
q += b;
q += c;
```

    » implies 4 object allocations !!!

» consider manual usage of **StringBuilder** or **StringBuffer** (thread-safe)
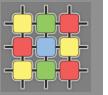
Immutable object

» concatenation of non-String types:

   » String.valueOf({primitives})

   » or

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

» make collections/list/map/set immutable (UnsupportedOperationException):

   » **Collections.unmodifiableCollection(…)**

   » **Collections.unmodifiableList(…)**

   » **Collections.unmodifiableMap(…)**

   » **Collections.unmodifiableSet(…)**

   » **Collections.unmodifiableSortedMap(…)**

   » **Collections.unmodifiableSortedSet(…)**

   » elements are not protected !!! use immutable elements too
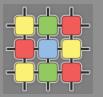
Factory method

» *creational* design pattern

» based on the concept of factories

» define a method for **creating objects in the interface which are subclasses of specific product**

» implementers can decide which class is instantiated (e.g. based on params)

» common usage

  » **toolkits and frameworks**

    » library code needs to create objects of types which are sub-classed by application using the framework

  » **test-driven development**

    » unit tests can use mock objects to simulate operations

» limitations

  » if use private constructors -> class cannot be extended

  » if use protected constructors -> subclass must re-implement all factory methods with exactly the same signatures (static cannot be overriden)
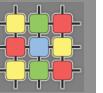
» Creator can define default implementation returning default factory product

» Note:

 » static factory methods – cannot be overridden !

 » Factory object can be instance

» product interface

```java
public interface Trace {
        // turn on and off debugging
        public void setDebug( boolean debug );
        // write out a debug message
        public void debug( String message );
        // write out an error message
        public void error( String message );
}
```

» product A

```java
public class SystemTrace implements Trace {
    private boolean debug;
    public void setDebug( boolean debug ) {
            this.debug = debug;
    }
    public void debug( String message ) {
        if( debug ) {  // only print if debug is true
                System.out.println( "DEBUG: " + message );
        }
    }
    public void error( String message ) {
        // always print out errors
        System.out.println( "ERROR: " + message );
    }
}
```
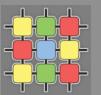
Factory method – example

» product B

```java
public class FileTrace implements Trace {

    private java.io.PrintWriter pw;
    private boolean debug;
    public FileTrace() throws java.io.IOException {
        // a real FileTrace would need to obtain the filename somewhere
        // for the example I'll hardcode it
        pw = new java.io.PrintWriter( new java.io.FileWriter( "c:\trace.log" ) );
    }
    public void setDebug( boolean debug ) {
        this.debug = debug;
    }
    public void debug( String message ) {
        if( debug ) {  // only print if debug is true
            pw.println( "DEBUG: " + message );
            pw.flush();
        }
    }
    public void error( String message ) {
        // always print out errors
        pw.println( "ERROR: " + message );
        pw.flush();
    }
}
```
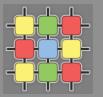
Factory method – example

» direct usage

  » need change
all instantiations
to modify behavior

```
//... some code ...
SystemTrace log = new SystemTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

» direct usage
  » need change
  all instantiations
  to modify behavior

```
//... some code ...
SystemTrace log = new SystemTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

» factory method

```
public class TraceFactory {
        public static Trace getTrace() {
                return new SystemTrace();
        }
}
//... some code ...
Trace log =      TraceFactory.getTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

  » just one place
  to modify behavior

```
public class TraceFactory {
        public static Trace getTrace() {
                try {
                        return new FileTrace();
                } catch ( java.io.IOException ex ) {
                        Trace t = new SystemTrace();
                        t.error( "could not instantiate FileTrace: " + ex.getMessage() );
                        return t;
                }
        }
}
```

Lazy initialization
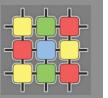
» *delay operation until the first time it is needed*

  » **lazy object creation**

  » **lazy calculation of a value**

  » **lazy class loading**

  » **lazy other expensive process**

» use a flag indicating that the process has taken place

» if not used -> save memory usage and/or processing time

» **lazy class loading**

  » classes are loaded only when they are first referenced

  » *use interfaces or parent classes for field types*
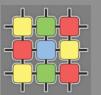
Lazy initialization – example

```
public class MyFrame extends Frame
{
  private MessageBox mb_ = new MessageBox();
  //private helper used by this class
  private void showMessage(String message)
  {
    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

VS

```
public final class MyFrame extends Frame
{
  private MessageBox mb_ ; //null, implicit
  //private helper used by this class
  private void showMessage(String message)
  {
    if(mb_==null)//first call to this method
      mb_=new MessageBox();
    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

» higher importance for complex objects (e.g. Image, DB connection)

» used often for lazy hash code computation in immutable objects
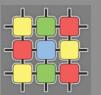
Singleton

» *class has only one instance with a global point of access to it*

» often used to control access to native resources like database connections or sockets

» unique repository of state, alternatively can be implemented as static

» lazy instantiation:

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            _instance = new MySingleton();
        }
        return _instance;
    }
    // ...
}
```

Singleton

» *class has only one instance with a global point of access to it*

» often used to control access to native resources like database connections or sockets

» unique repository of state, alternatively can be implemented as static

» lazy instantiation:

```java
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static synchronized MySingleton getInstance() {
        if (_instance==null) {
            _instance = new MySingleton();
        }
        return _instance;
    }
    // ...
}
```

» how to avoid locking?

Double-checked locking
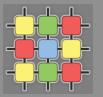
» **reduce the overhead** of acquiring a lock by use of **locking criterion**

» common usage

    » multi-threaded environment

    » combination with lazy initialization

» typical use pattern

    » check the locking criterion without obtaining the lock

    » obtain the lock

    » double-check whether the variable has been already initialized

    » otherwise, initialize
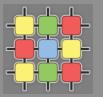
» avoid expense of locking

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                _instance = new MySingleton();
            }
        }
        return _instance;
    }
    // ...
}
```

» avoid expense of locking

```
public static class MySingleton {
    private static MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                if (_instance==null) {
                    _instance = new MySingleton();
                }
            }
        }
        return _instance;
    }
    // ...
}
```

» avoid expense of locking

```java
public static class MySingleton {
    private static volatile MySingleton _instance;

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
        if (_instance==null) {
            synchronized (MySingleton.class) {
                if (_instance==null) {
                    _instance = new MySingleton();
                }
            }
        }
        return _instance;
    }
    // ...
}
```

```java
public static class MySingleton {
    private static MySingleton _instance =
     new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance() {
     return _instance;
    }

    // ...
}
```
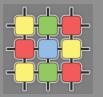
» initialization on demand holder idiom

```
class Foo {
    private static class HelperHolder {
        public static Helper helper = new Helper();
    }
    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}
```

» inner class are not loaded until they are referenced

» BUT

    » if construction fail, it throws **NoClassDefFoundError** during run-time !
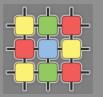
Serializable Singleton - example

```
public static class MySingleton implements Serializable {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```

» is this correct singleton?

```java
public static class MySingleton implements Serializable {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```

» is this correct singleton?

  » **NO** – due to serialization which will create new instance

» Fix – using read resolve

```java
private Object readResolve() {
    return _instance;
}
```

```
public enum MySingleton {
    INSTANCE;

    private boolean test;

    MySingleton() {
        test = true;
    }

    public void myMethod() {
        // ...
    }
}
```

- » serialization issue is fixed by default
- » creation is thread-safe
- » cannot inherit other enums or classes (only can implement interfaces)
- » no lazy intialization
- » different classloaders is still an issue
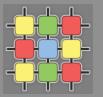
» don't allow subclassing due to static getInstance():

- factory class with method returning singleton instance (requires non-private constructor)

» issue with more VMs in distributed system (e.g. RMI) -> singleton should not be used to store state !!!

» classloaders -> one singleton per each classloader

» *in older JVMs private static references for non-reachable objects was not enough to keep that instance*
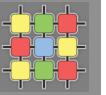
» *parametric singleton*, single instance with given parameter

» often in combination with immutable object

```java
public static class MyMultiton {
    private static final Map<Object, MyMultiton> instances = new HashMap<Object, MyMultiton>();

    private final Object attribute;

    private MyMultiton(final Object attribute) {
        this.attribute = attribute;
    }

    public static MyMultiton getInstance(Object attribute) {
        synchronized (instances) {
            MyMultiton instance = instances.get(attribute);
            if (instance == null) {
                instance = new MyMultiton(attribute);
                instances.put(attribute, instance);
            }
            return instance;
        }
    }
}
```

» beware memory consumption !

» use WeakReference or SoftReference
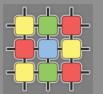
» usage of reflection to create instance

```java
public static void main(String[] args) throws Exception {
    MySingleton s = MySingleton.getInstance();

    Class<?> cl = MySingleton.class;

    Constructor<?> cons = cl.getDeclaredConstructor();
    cons.setAccessible(true);

    MySingleton s2 = (MySingleton) cons.newInstance();
}
```

» Fix 1

```java
SecurityManager mgr = new SecurityManager();
System.setSecurityManager(mgr);
```

```
Exception in thread "main" java.security.AccessControlException: access denied (java.lang.r
eflect.ReflectPermission suppressAccessChecks)
 at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
 at java.security.AccessController.checkPermission(AccessController.java:546)
 at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
 at java.lang.reflect.AccessibleObject.setAccessible(AccessibleObject.java:107)
```
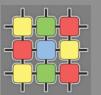
» usage of reflection to create instance

```java
public static void main(String[] args) throws Exception {
    MySingleton s = MySingleton.getInstance();

    Class<?> cl = MySingleton.class;

    Constructor<?> cons = cl.getDeclaredConstructor();
    cons.setAccessible(true);

    MySingleton s2 = (MySingleton) cons.newInstance();
}
```

» Fix 2

```java
public static class MySingleton {
    private static MySingleton _instance = new MySingleton();

    private MySingleton() {
        if (_instance != null) {
            throw new IllegalStateException("Singleton instance already created")
        }
        // ...
    }

    public static MySingleton getInstance()
    {
      return _instance;
    }
    // ...
}
```
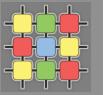
» *behavioral* design pattern

» **selection of algorithm in runtime**

» define common interface for family of algorithms

» make different implementations interchangeable

» usually strategy is selected independently from clients that use it
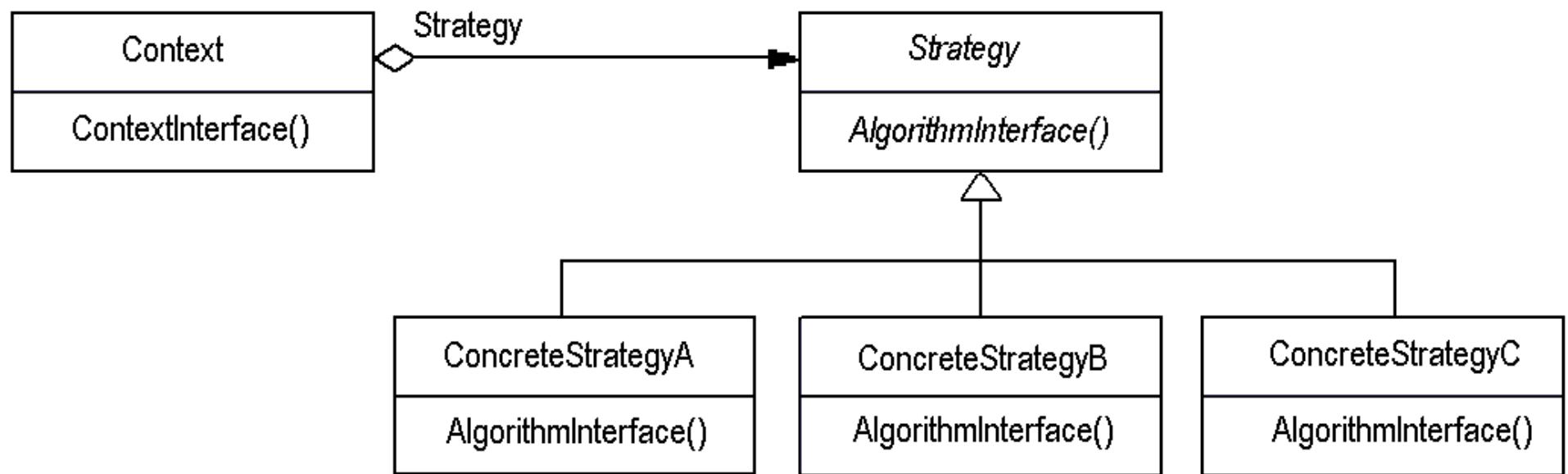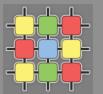
» Strategy

```
public interface SortInterface {
public void sort(double[] list);
}
```

» Context

```
public class SortingContext {
private SortInterface sorter = null;

public void sortDouble(double[] list) {
 sorter.sort(list);
}

public SortInterface getSorter() {
 return sorter;
}

public void setSorter(SortInterface sorter) {
 this.sorter = sorter;
}
}
```

» Initialization

```
public class SortingClient {
public class SortingClient {
 public static void main(String[] args) {
  double[] list = {1,2.4,7.9,3.2,1.2,0.2,10.2,22.5,19.6,14,12,16,17};
  SortingContext context = new SortingContext();
  context.setSorter(new BubbleSort());
  context.sortDouble(list);
  for(int i =0; i< list.length; i++) {
   System.out.println(list[i]);
  }
 }
}
```
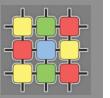
» *structural* design pattern

» **compose objects into tree structures**

» group of objects are treated as a single instance of an object

» clients do not need to use difference between compositions and individuals

» component – interface

» leaf – behavior for primitive objects, no children

» composite – stores child
   components

» client – manipulates
   objects using component

» Example – Graphics, arithmetic express.


```
Client                    Component

                          +Operation()

                    Leaf              Composite          -children

                    +Operation()      +Operation()
                                      +Add(Component)()        1
                                      +Remove(Component)()
                                      +GetChild(int)()
```

for all c in children c->Operation()

24/05/17
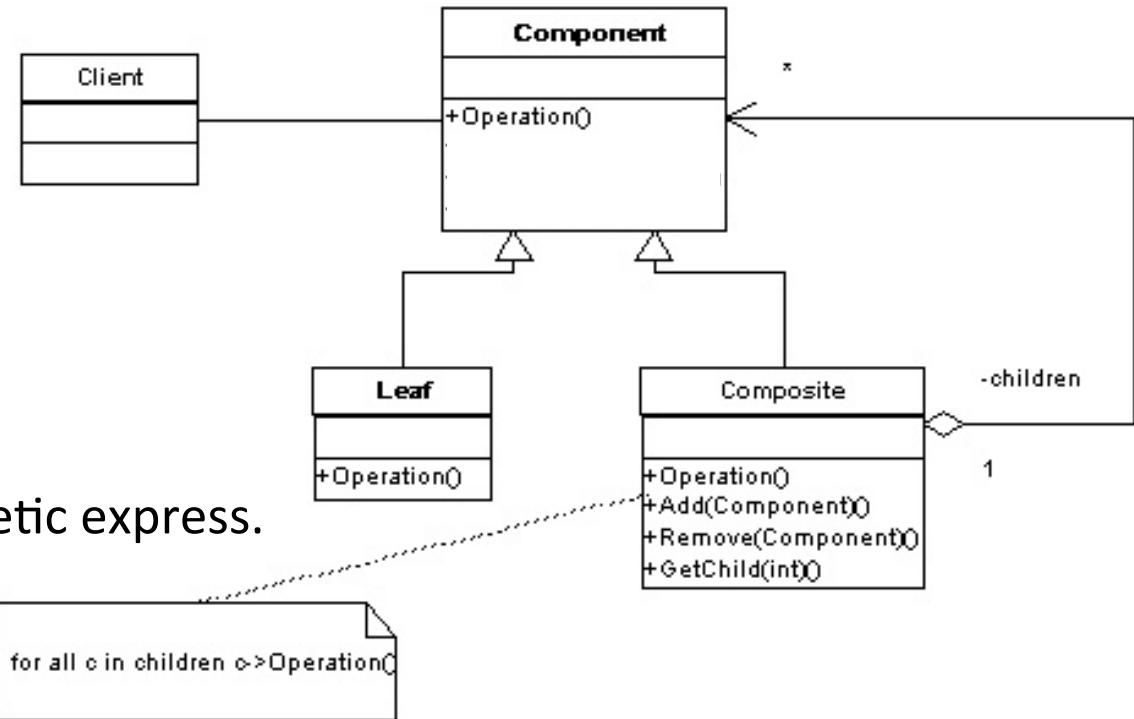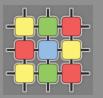
- » *behavioral* design pattern
- » iterator is used to **access elements of an aggregate object** (Collection)
- » sequential without exposing underlying implementation
- » very common in Java libraries (Aggregate – Iterable)
  - » List, HashSet, HashMap, Tree, …

```
Set items = new TreeSet();
...
Iterator seq = items.iterator();
while (seq.hasNext())
{
    Item item = (Item) seq.next();
    ...
    if (...) seq.remove();
}
```
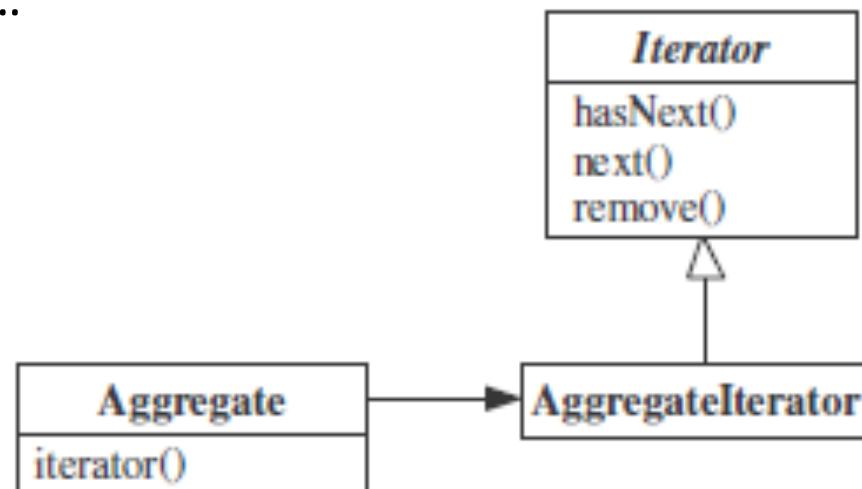

- » Collection can be changed only through iterator -> if changed throws ConcurrentModificationException
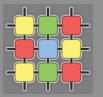
Iterator

» ListIterator - List.listIterator()

  » hasPrevious(), previous()

  » nextIndex(), previousIndex()

  » add(...), set(...)

» since Java 1.5

```
Set<Item> items = new TreeSet<Item>();
...
for (Item item : items)
{   ...
}
```

  » works on arrays, anything implements Iterable
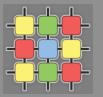
Abstract Factory

- » *creational* design pattern

- » extension of Factory pattern

- » encapsulate a **group of individual factories having common theme**

  - » multiple **factory methods**

- » user creates concrete implementation of abstract factory and then uses generic interface to create concrete objects

Abstract Factory
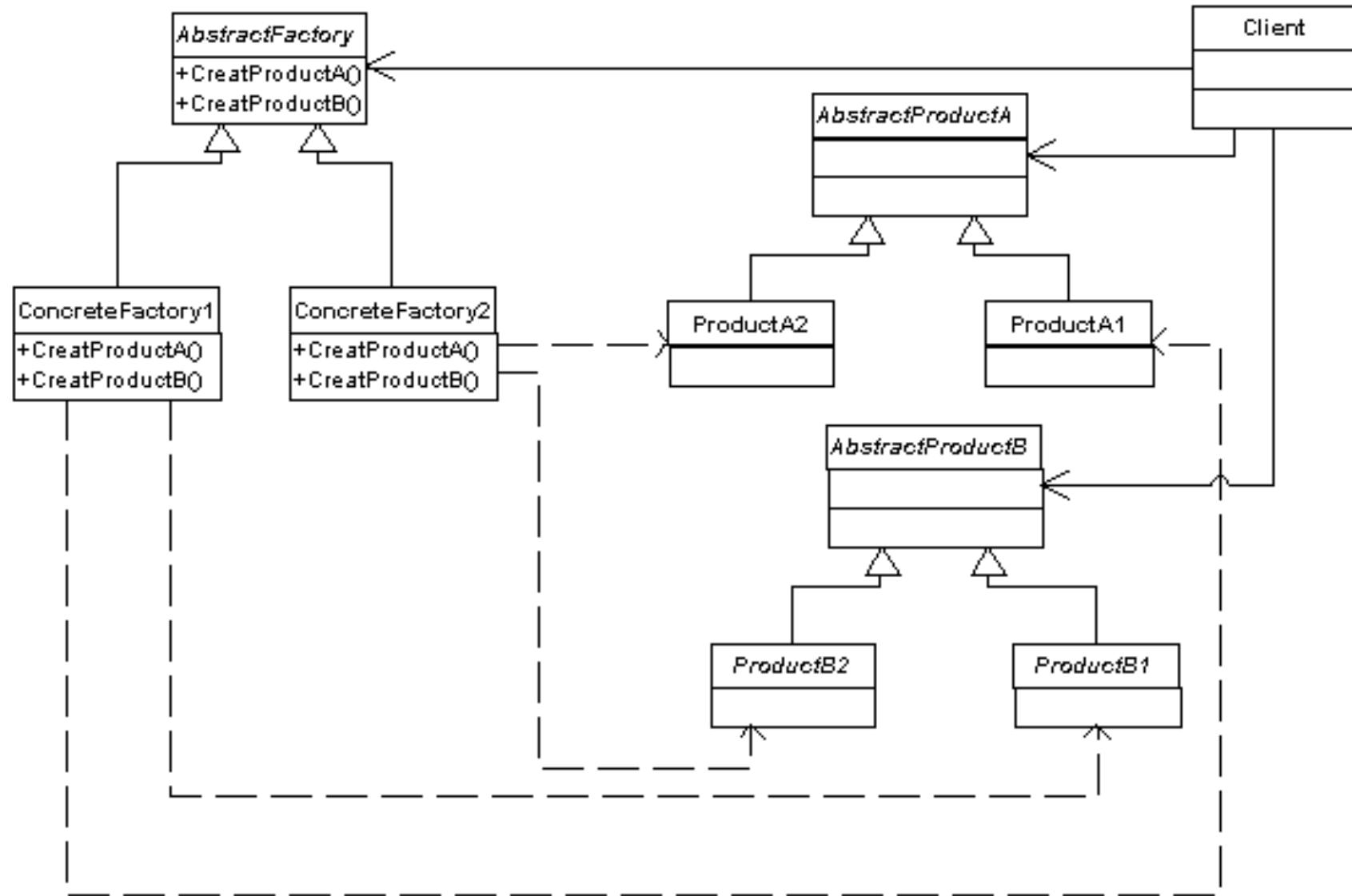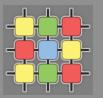
» Example: multiple look-and-feels in GUIs (e.g. Linux vs. Windows)
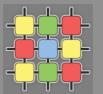
Command

» *behavioral* design pattern

» **encapsulate all information needed to call a method at a later time**

» client
  » instantiates the command
  » provides information required
» invoker
  » decides when to call the method
» receiver
  » contains method's code

» Example
  » Thread pools (java.lang.Runnable)
  » GUI Action
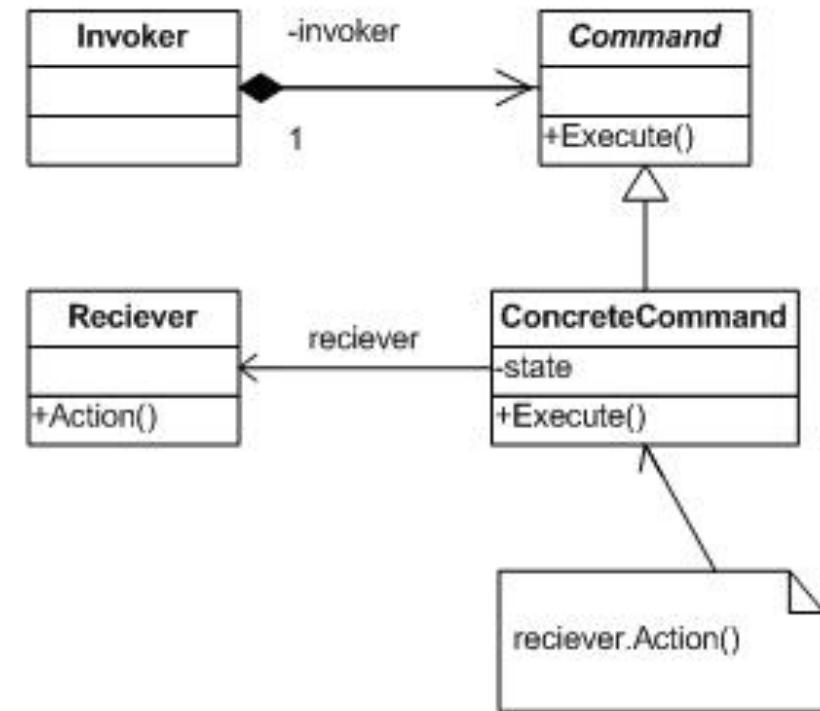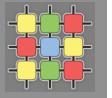
» create initial class

 » must be present in bootstrap class loader

» links the initial class

 » cause loading, linking and invocation of other classes

» initialize class (class vs. instance initialization !)

» start public void main (String[])
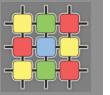
» **Loading**

  » finding binary form of class or interface (e.g. computing on the fly)

  » form **Class**

  » implemented by **ClassLoader**

    » can cache binary representations (can decrypt, verify dig. signature)

    » prefetch them based on expected usage

    » load group of related classes together

» **Linking**

  » binary form into run-time state in JVM

  » verification – structural check (correct opcodes, branches, …)

  » preparation

    » create static fields, fill default values (no initializers !)

    » precompute additional data structures (e.g. method table)

  » resolution of symbolic references – validation, direct reference
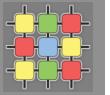
» class is being initialized in the following cases:

  » instance of class has to be created

  » static method of class is invoked

  » non-constant static field of class is used

  » sub-class is initialized

  » invocation of reflective methods over class

  » it is initial class for start-up

» class is not initialized

  » when static final field is initialized with compile-time constant

» **class initialization sequence**

  » super class initialization

  » initialization in declaration order (you cannot use values after, compiled into "<clinit>:()V") :

    » user class static initializers

    » initializers static fields (class + interfaces – by default public static final)
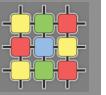
» requires careful synchronization (synchronized on class object)

» **Class object state**

    » verified and prepared

    » being initialized by some thread

    » fully initialized and ready for use

    » in error state – verification failed, initialization failed

           (throws NoClassDefFoundError)

» **Class instance** initialization

    » memory allocation (fields in class + superclasses) -> OutOfMemoryError

    » all variables are set to default values (0, false, null)

    » prepare args for other/super constructor invocation (follow this(…) and follow super(…))

    » execute instance initializers + field initializers in declaration order (compiled into "<init>:()V")

    » execute the rest of the body of constructor
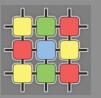
Class instance finalization

» protected void finalize() throw Throwable

  » called before object space is reclaimed by GC

  » you shalll call super.finalize() !!!

  » do not call method explicitly !!!

» class/interface unloading

  » if its class loader is unreachable

  » bootstrap class loader is always reachable -> system classes are never unloaded !

```java
public static class InitializerBlocks {

    public InitializerBlocks(){
        System.out.println("1");
    }


    {
        System.out.println("2");
    }


    {
        System.out.println("3");
    }


    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

```java
public static class InitializerBlocks {

    public InitializerBlocks(){
        System.out.println("1");
    }

    {
        System.out.println("2");
    }

    {
        System.out.println("3");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

2

3

1

```java
public static class InitializerBlocks {
    private boolean boolField = privInstance();

    private final boolean privInstance() {
        System.out.println("2");
        return true;
    }

    public InitializerBlocks(){
        System.out.println("1");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

```java
public static class InitializerBlocks {
    private boolean boolField = privInstance();

    private final boolean privInstance() {
        System.out.println("2");
        return true;
    }

    public InitializerBlocks(){
        System.out.println("1");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

2

1

Initializer block – static variant

```java
public static class InitializerBlocks {
    private static final boolean boolField;

    static {
        // compute value for boolField ...
        boolField = true;
    }

}
```

```java
public static class A {
    public static final A A = new A();
    private A() { }

    private static final Boolean BOOL = true;

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```

» what is the output?

```java
public static class A {
    public static final A A = new A();
    private A() { }

    private static final Boolean BOOL = true;

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```
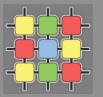
» what is the output?

  » throws NullPointerException – due to recursive class initialization, and auto-unboxing

» correct:

```java
public static class A {
    private static final Boolean BOOL = true;

    public static final A A = new A();
    private A() { }

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```
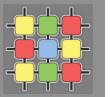
- » classloader types:
  - » **bootstrap class loader**
    - » system class loader – searches runtime, inst. extension, class path
  - » **user-defined class loader**
    - » extraction from encrypted file, verify digital signature
    - » loading from non-standard sources (e.g. network)
    - » generate on the fly
- » each class has
  - » defining class loader – finally define Class
  - » initiating class loader – initiate class loading (e.g. through other CL)
- » class is uniquely identified by pair !
  - » fully qualified name
  - » defining class loader
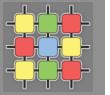
» referenced classes from X are loaded by its defining CL

» each class is loaded only once if it is not previously unloaded

» method **Class loadClass(String)** - qualified name
  » a cache implemented by **Class findLoadedClass(String)**
  » get raw bytes from class from somewhere
  » if ok, define class from array of bytes using **Class defineClass(…)**
  » if failed, delegate loading to other class loader
    » e.g. **Class findSystemClass(String)**
    » e.g. **getParent().loadClass(String)** – CL which creates the current one
  » if still no class, throw **ClassNotFoundException**
  » if resolve is required call **void resolveClass(Class)** to link class
» **Class Class.forName(String)**, **Class.getSystemClassLoader().loadClass(String)**
» **cl.loadClass(String), c.newInstance(), constructor.newInstance(…)**
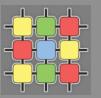
» can examine or modify the run-time behavior

  » create external classes by qualified name (through Class loaders) – do not need to have class during compilation

  » class browser – enumerate members

  » debugger – examine private members

» BUT

  » performance overhead – dynamic resolution, slower

  » security restrictions – security context, e.g. Applet

  » unexpected side-effects – access private fields and methods

Reflection
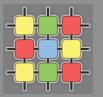
» retrieve Class object

  » **Class getClass()** – returns instance Class representation

  » **XXX.class** – from type, no instance

    » e.g. "aa".class

  » **Class Class.forName(String), CL.loadClass(String)**

  » **Class.getSuperClass(), Class.getClasses(), Class.getDeclaredClasses(), Class.getEnclosingClass(), {Field|Method| Constructor}.getDeclaringClass()**

» examine class modifiers and types

  » **Class.getModifiers()**

  » **Class.getTypeParameters()** – get Generic types

  » **Class.getGenericInterfaces()**

  » **Class.getSuperclass()**

  » **Class.getAnnotations()**

Reflection
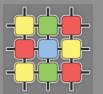
» discovering class members

Member	Class API	List of members?	Inherited members?	Private members?
Field	getDeclaredField()	no	no	yes
	getField()	no	yes	no
	getDeclaredFields()	yes	no	yes
	getFields()	yes	yes	no
Method	getDeclaredMethod()	no	no	yes
	getMethod()	no	yes	no
	getDeclaredMethods()	yes	no	yes
	getMethods()	yes	yes	no
Constructor	getDeclaredConstructor()	no	N/A[1]	yes
	getConstructor()	no	N/A[1]	no
	getDeclaredConstructors()	yes	N/A[1]	yes
	getConstructors()	yes	N/A[1]	no

Reflection
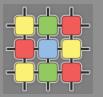
» Fields

  » get field types, generic types

  » get field modifiers

  » get and set field value (private if no security manager)

» Methods

  » get method types including attributes

  » get method modifiers

  » invoke method

» Constructors

  » find constructor with specific parameters

  » get constructor modifiers

  » create new class instance

» Arrays (through java.lang.reflect.Array)

  » get array types

  » create new array

  » get/set array components
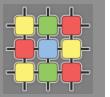
» Method call example:

Class<T> c = Class.forName("MyClass"); // Class<T> c = MyClass.class;

Method m = c.getMethod("myMethod");

Object retVal = m.invoke(object, …);

» Field usage example:

Class<T> c = MyClass.class;

Field f = c.getField("myField");

Object value = f.get(object);