Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- **Exception Handling**
- **Parameter Passing Mechanisms**
- **Abstract Classes**
- **Interfaces**
- **Dependency Injection**

Exception Handling

- Exceptions are events that indicate an exceptional circumstance/problem the programmer should/must deal with.
- Exceptions can be handled with the try/catch/finally statement or with the throws-clause attached to method declarations.
- Exceptions neither caught or thrown lead to the termination of the program.

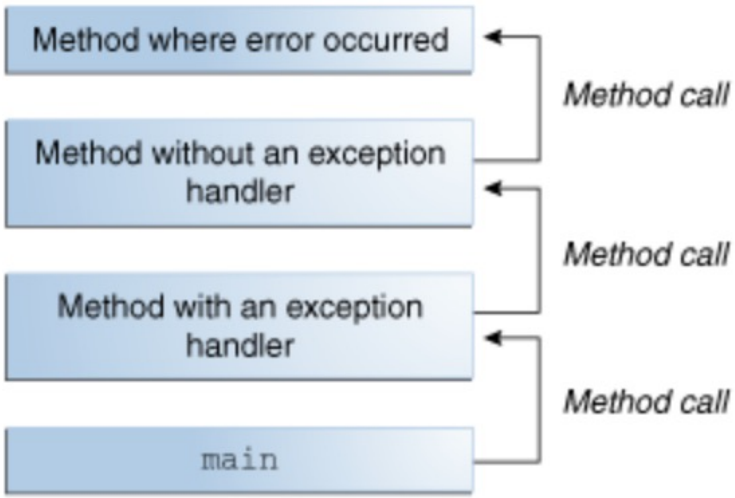
```
public int divide (int x, int y) {
       try                                  { return x/y; }
       catch (ArithmeticException e) { // handle exception }
       finally                              { // clean-up;
                                              //always executed }
}
```

```
public int divide (int x, int y) throws ArithmeticException {
       return x/y;
}
```

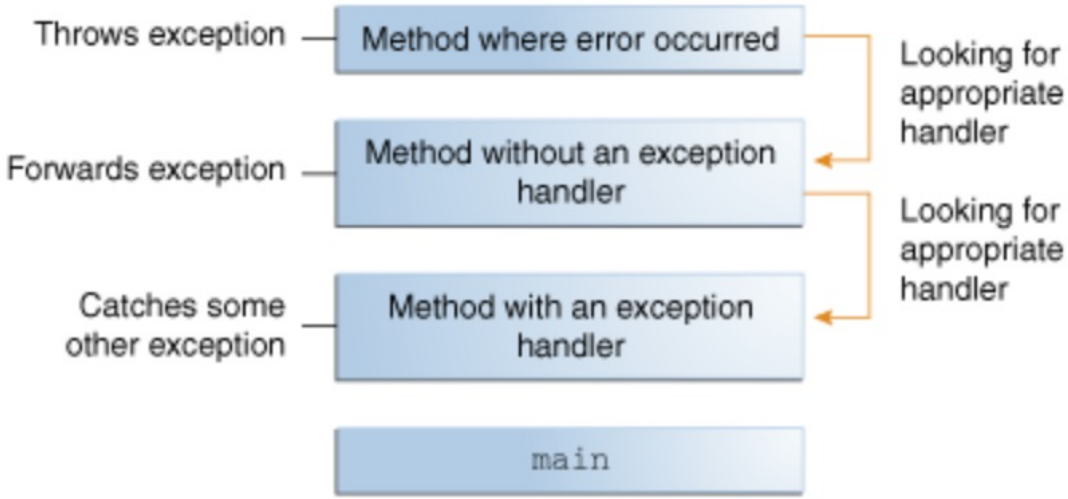
Exception Handling

Hierarchical concept

- The runtime system searches the call stack for a method that contains code that can handle the exception.
- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.



The call stack.



Searching the call stack for the exception handler.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exception Handling

Checked Exceptions

- Are subject to the Catch or Specify Requirement, i.e., they must be caught or specified to be thrown.

Unchecked Exceptions

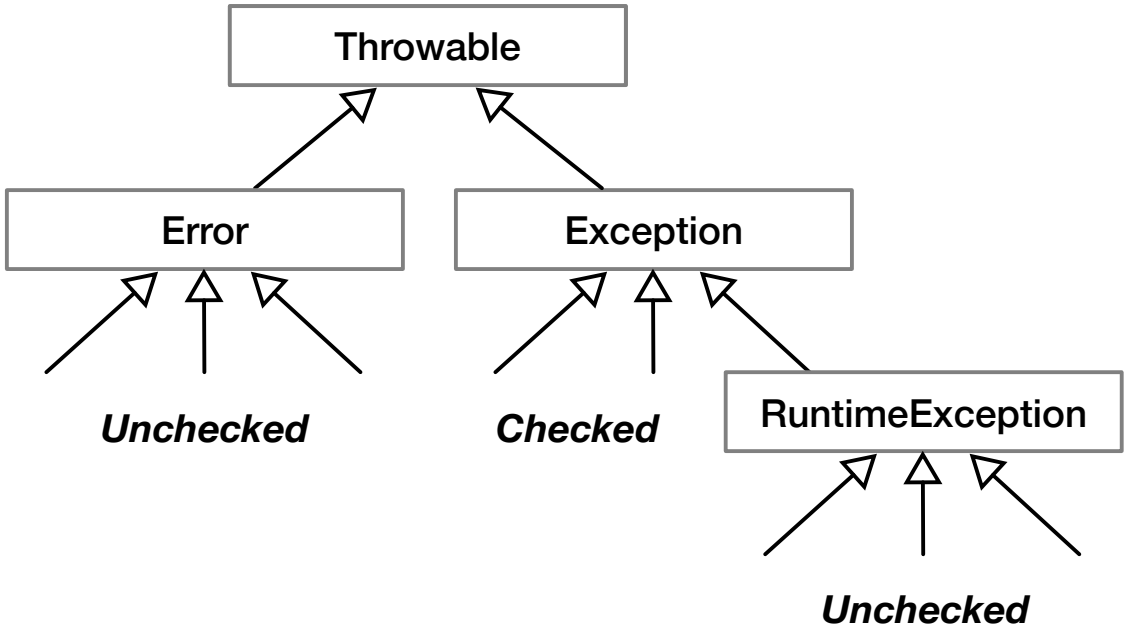
- Are not subject to the Catch or Specify Requirement.
- **`Error`**

  Indicates a serious problem that an application should not try to catch.
- **`RuntimeException`**

  These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.

Exception Handling

- The class **`Exception`** and any subclasses that are not also subclasses of **`RuntimeException`** are checked exceptions and must be caught or declared to be thrown.
- **`RuntimeException`** and its subclasses are unchecked exceptions.
- Errors (i.e., instances of **`Error`**) are also unchecked and should not be caught.



Exception Handling

```
...
URL url = null;
try {
    url = new URL("http://java.sun.com/");
}
catch (MalformedURLException e) {
    e.printStackTrace();                    // for debugging
    ...
}
```

```
String delayString = ...;

try {
   delay = Integer.parseInt(delayString);  // String -> int
}
catch (NumberFormatException e) {
   delay = DEFAULT_DELAY;                  // use default
}
```

Exception Handling

```
public class TestTextFileReader {

   public static void main(String args[]) {
      try {
         BufferedReader br = new BufferedReader(
                             new InputStreamReader(
                             new FileInputStream(args[0])));
         String line = null;
         while ((line = br.readLine()) != null) {
               System.out.println(line);
         }
      }
      catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("usage: java TestTextFileReader <file>");
      }
      catch (FileNotFoundException e) {
        System.out.println("File " + args[0] + " not found.");
      }
      catch (IOException e) {
        System.out.println("Error while reading " + args[0] + ".");
        e.printStackTrace();
      }
   }
}
```

Exception Handling

- **Try without catch**
  - there is no catch block but instead a throws clause
  - finally is always executed – in this case after the return, before control is passed to the caller

```
public Object take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
          notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
      lock.unlock();
    }
}
```

Exception Handling

- The **try-with-resources** statement …

```
String readFirstLineFromFile(String path) throws IOException {

    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }

}
```

… instead of …

```
String readFirstLineFromFileWithFinallyBlock(String path) throws IOException
{
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

See: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Parameter Passing Mechanisms

In a procedure/method/function call the values of the variables

- from a caller (i.e., actual parameters/arguments) are transferred
- to the callee (called procedure, formal parameters/arguments)

usually by one of the following mechanisms:

- Standard Mechanisms
  - Call by value
  - Call by reference
- Other Mechanisms
  - Call by copy
  - Call by value-result (copy-in, copy-out)
  - Call by result
  - Call by name

Parameter Passing Mechanisms

- **Call by value** (C, Java)

  The actual arguments are evaluated in the caller's environment and their values are bound to the formal parameters of the procedure. With call-by-value actual arguments cannot be changed.

- **Call by reference** (Fortran, C++)

  The actual arguments must be variables *(l-values).* Their addresses are bound to the parameters of the procedure. Call-by-reference allows a procedure to change the actual arguments.

Parameter Passing Mechanisms

Java - call by value

```
public class ParameterTransfer {
    static void setRadius(double r, Circle c) {
        c.radius = r; r = 0; c = null;
    }
    public static void main(String[] args) {
        double maxr = 100.0;
        Circle c    = new Circle(1.0,1.0,1.0);
        setRadius(maxr, c);
        System.out.println("maxr = " + maxr);
        System.out.println("c.radius = " + c.radius);
    }
}
```

```
> java ParameterTransfer
maxr = 100.0
c.radius = 100.0
```

Parameter Passing Mechanisms

C++ - call by value

```
#include <iostream>

void myFun(int n) {
    n = n + 1;
}

int main() {
   int n = 1;
   myFun(n);
   std::cout << n << std::endl;                    // Ausgabe: 1
}
```

Note: When an object is passed by value, the object will be copied (C++ copy constructor).

Parameter Passing Mechanisms

C++ - call by reference

```
#include <iostream>

void myFun(int &n) {
    n = n + 1;
}

int main() {
   int n = 1;
   myFun(n);
   std::cout << n << std::endl;                    // Ausgabe: 2
}
```

Parameter Passing Mechanisms

Example: C++ - call by value (using pointers)

```
#include <iostream>

void myFun(int *n) {
    *n = *n + 1;
}


int main() {
   int n = 1;
   myFun(&n);
   std::cout << n << std::endl; // Ausgabe: 2
}
```

Parameter Passing Mechanisms

- **Call by value-result** (Fortran sometimes, CORBA *inout* parameters)

  Actual arguments are evaluated and their values are copied into local variables created for each parameter. Upon return, values from local variables are copied back to actual arguments.

- **Call by result** (CORBA out parameters)

  Upon return, values from formal parameters are copied back to actual arguments.

Parameter Passing Mechanisms

- **Call by name** (Algol 60)

  The arguments are not evaluated in the caller's environment. The text of the arguments is passed to the procedure and replaces the parameters in the procedure body (similar to C/C++ preprocessor). Call-by-name has the advantage of delaying the evaluation of an argument until it is used.

Parameter Passing Mechanisms

Algol 60 - call by name (Jensen's Device)

```
begin
   integer i
   real procedure sum (expr, i, low, high);
   value low, high;
   real expr;
   integer i, low, high;

   begin
      real s;
      s := 0;
      for i := low step 1 until high do
         s := s + expr;
         comment the value of expr depends on the value of i
         sum := s
    end;
    print (sum (1/i, i, 1, 100))
end
```

$$\sum_{i=1}^{100} 1/i$$

Parameter Passing Mechanisms

Side Effect

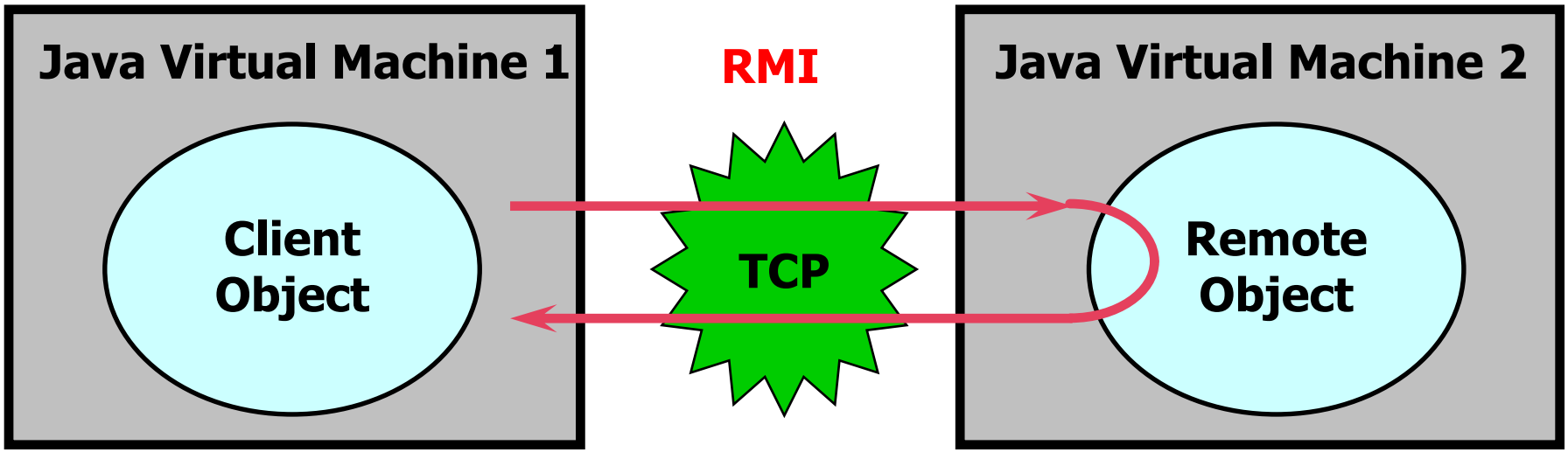
- A method or function is said to have a side effect if it has an observable effect besides returning a value or modifying the value of some of its parameters, e.g., if it modifies global variables or other variables outsides of its local environment.

- Functions without side effects are often called pure functions.

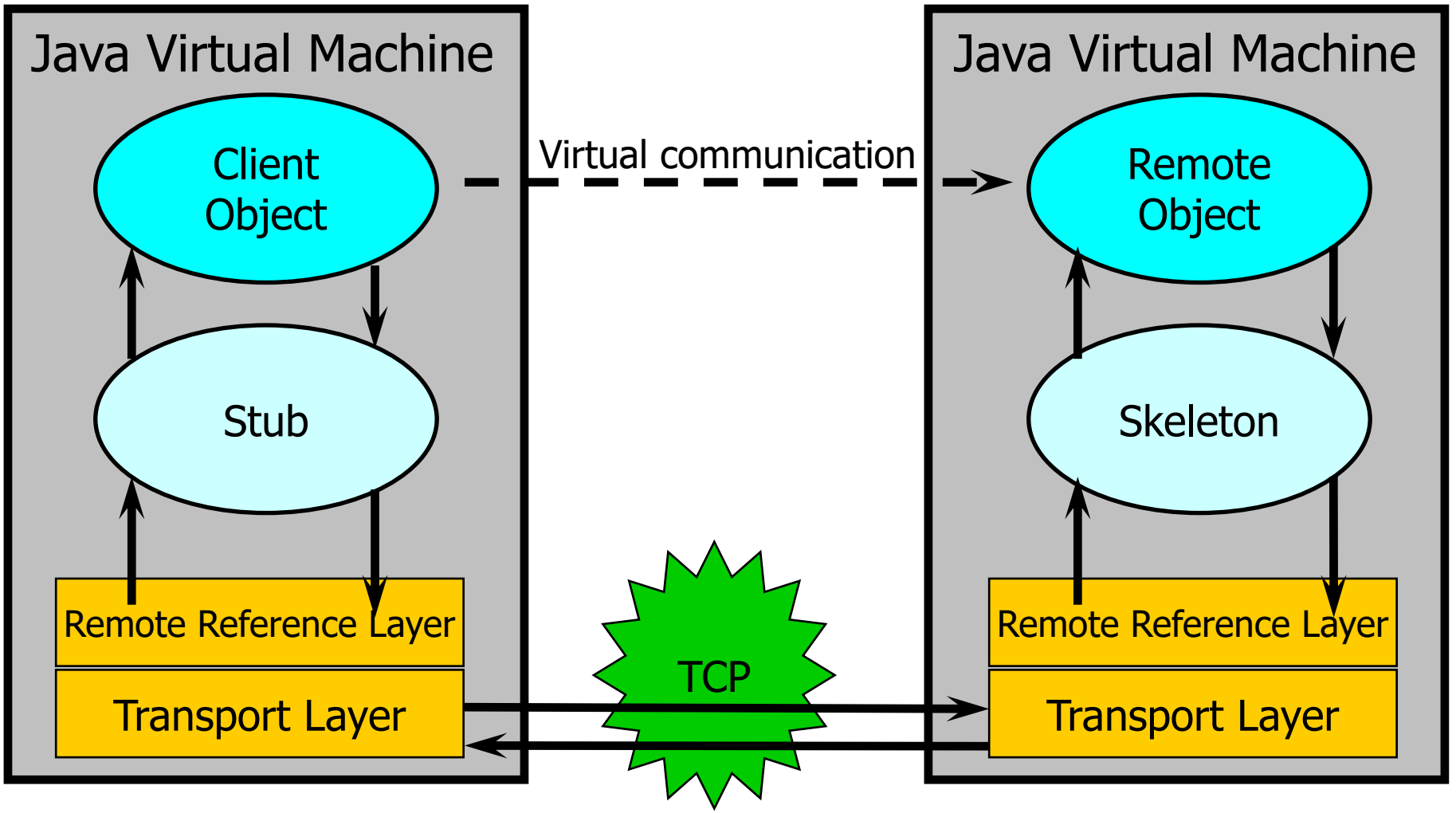
Parameter Passing - Distributed Applications

Remote Method Invocation (RMI)

- Java's high-level communication mechanism for passing objects in distributed applications.
- A remote object (aka. server) is an object whose methods may be invoked from a remote JVM, usually running on a different machine.
- A remote object is described by one or more remote interfaces.



Remote Method Invocation (RMI)



Remote Method Invocation (RMI)

A remote method invocation constitutes invoking a method of a remote interface.

- Same syntax as a normal method invocation.
- Synchronous & blocking.
- Parameter passing mechanism are different for ordinary and remote objects.

```
interface Server extends Remote {
   String sayHello(Person p) throws RemoteException;
}
```

```
class Client {
   Server s;
   Person c;
   ...
   s = s.sayHello(c);
   ...
```

```
class MyServer
        extends UnicastRemoteObject
        implements Hello {
   String sayHello(Person p) {
      return "hello from " + p.name +"!";
   }
   ...
```

Parameter Passing - Java RMI

- Parameters of primitive types are passed by value to a remote method.

- Ordinary (non-remote) objects

  If an ordinary object is passed, the object will be serialized and passed by copy.

- Remote objects

  If a remote object is passed, only the stub (proxy) of the remote object is passed, while the object itself remains in the JVM where it has been instantiated.

Abstract Classes

- Abstract classes may be used to represent abstract concepts.
- Abstract classes are declared with the keyword **`abstract`**.
- Abstract classes cannot be instantiated, but they can be subclassed.

```
public abstract class Shape {
        ...
}
```

Abstract Classes

- An abstract class has usually on ore more **abstract methods**.
- Abstract methods are declared with the keyword **`abstract`**.
- Abstract methods specify only the interfaces (return type and parameter types) but do not have an implementation.

```
abstract class Shape {                          // abstract class
  abstract double area();                       // abstract method

}
```

Abstract Classes

- A class that has at least one abstract method must be declared as abstract.
- Besides abstract methods, abstract classes may also have variables, constructors and (non-abstract) methods.

```
abstract class Shape {                          // abstract class
   static int nrOfShapes;                       // class variable
   Point origin;                                // instance variable
   Shape() { nrOfShapes++; }                    // constructor
   abstract double area();                      // abstract method
   Point getOrigin() {return origin; }          // method
}
```

Abstract Classes

- A subclass of an abstract class A can be instantiated if it implements all abstract methods of A.

```
abstract class Shape {                          // abstract class
   static int nrOfShapes;                       // class variable
   Point origin;                                // instance variable
   Shape() { nrOfShapes++; }                    // constructor
   abstract double area();                      // abstract method
   Point getOrigin() {return origin; }          // method
}

class Circle extends Shape {
      double r;
      double area() { return r*r*3.14; } // implementation
}
...
Shape s = new Circle();
System.out.println(s.area());
```

Abstract Classes

```
...
Shape shapes[] = new Shape[N];              // array of shapes
shapes[j] = new Circle();
...
/* Compute area of all shapes
 * Note: code will also work for future subclasses of Shape
 */
double sum = 0;
for (int i=0; i < shapes.length; i++)
     sum += shapes[i].area();
```

Abstract Classes

```
class Rectangle extends Shape {

        double h,w;
        double area() { return h*w; }
}
```

```
Shape shapes[] = new Shape[N];              // array of shapes

shapes[j] = new Circle();

shapes[k] = new Rectangle();

...

/* Compute area of all shapes
 * Note: code also works for rectangles
 */

double sum = 0;

for (int i=0; i < shapes.length; i++)

     sum += shapes[i].area();
```

Abstract Classes

- If a subclass B of an abstract class A does not implement all abstract methods inherited from A, B must itself be declared as abstract.

```
abstract class Shape {                          // abstract class
   static int nrOfShapes;                       // class variable
   Point origin;                                // instance variable
   Shape() { nrOfShapes++; }                    // constructor
   abstract double area();                      // abstract method
   Point getOrigin() {return origin; }          // method
}

// abstract subclass of Shape
abstract class ColoredShape extends Shape {
   Color c;
   void setColor(Color color) { c = color; }
}
```

Interfaces

- Interfaces are used to decouple the specification of behavior (functionality) from its implementation.

- An interface usually contains abstract methods only (no implementation).

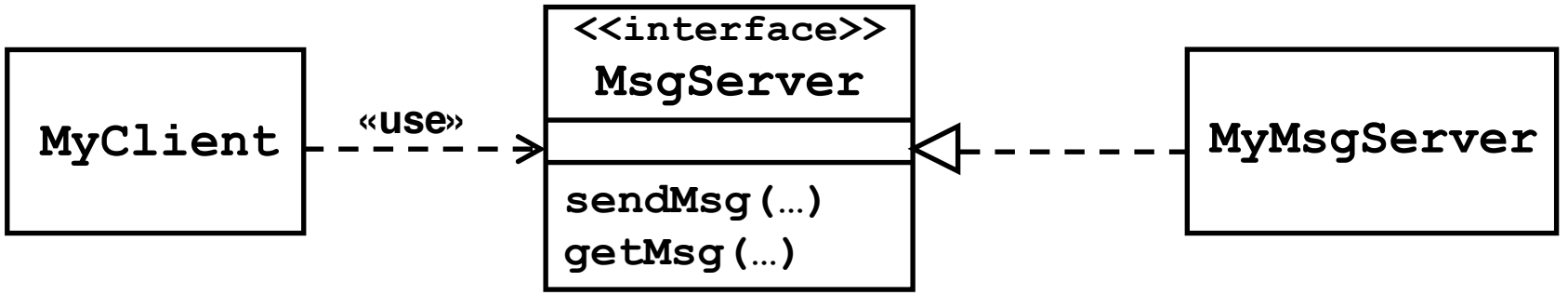
- Interfaces form a contract between a provider (realization, implementation) and users/clients. This contract is enforced at build time by the compiler.

- An interface is used without knowing its implementation.

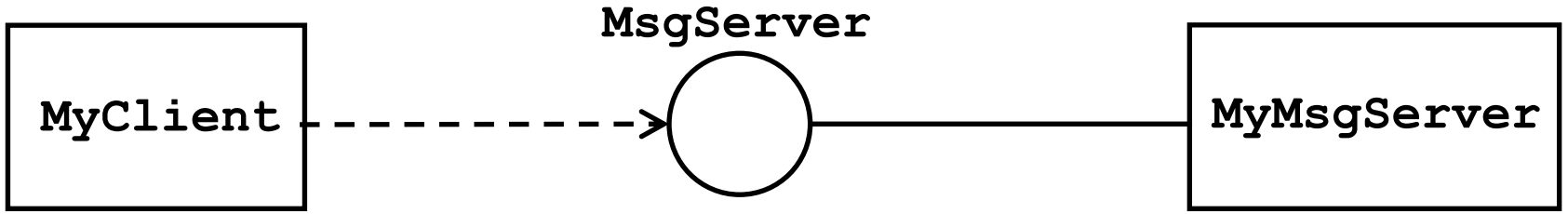
- If a class implements an interface, all methods of the interface must be implemented before the class successfully compiles.

Interfaces - UML

- Two perspectives: use and realization (=implementation) of an interface



- Alternative: Lollipop-Notation



- By agreeing on an interface *(contract)*, software that uses the interface (e.g., client side) and software that implements the interface (e.g., server side) can be developed independently of each other.

Interfaces

In Java, an interface

- defines a reference data type
- usually contains only abstract methods but no fields
- cannot be instantiated
- may contain constants, default methods, static methods and nested types.

```
public interface MsgServer {
        public Status getStatus();
}
```

- If a class implements an interface it must implement all methods of the interface.

```
public class MightyMsgServer implements MsgServer {
        public Status getStatus() { return new Status(); }
}
```

Interfaces

- A class can implement multiple interfaces but extend only one class.

```
public interface Collectible {
        public double value();
}

public class Stamp implements Collectible, Serializable {
        public double value() { ... }          // implementation


public class OldTimer extends Car implements Collectible {
        public double value() { ... }         // implementation
}
```

Interfaces

- Objects of a class **`A`** which implements and interface **`I`** have both the type **`A`** and the type **`I`** (cf., polymorphism).

```
public class OldTimer extends Car
                        implements Collectible {
        public double value() { ... }
}


OldTimer chevy57 = new OldTimer();
Car car = chevy57;
Collectible c = chevy57;
```

Interfaces

- If a class A implements an interface I, this property is inherited by subclasses, i.e., a subclass S of A inherits the implementation of the interface.

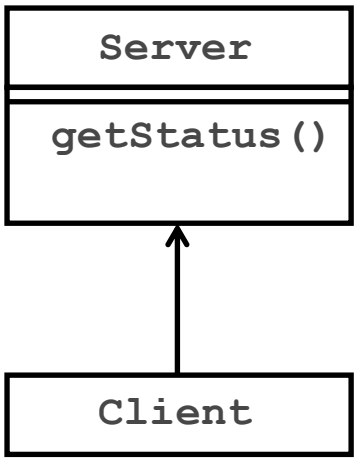
- If class A implements interface I, the clause **`implements I`** is optional for a subclass S of A.

- By just looking at the declaration of a class, without taking into account all its superclasses, it might no be visible that the class implements a certain interfaces.

Interfaces

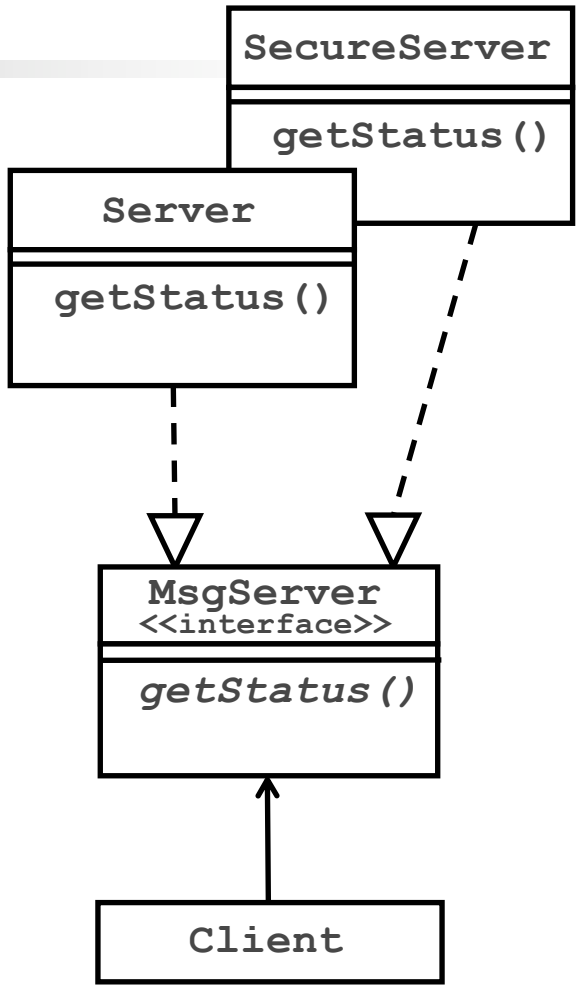
- Interfaces may be used to break tight coupling of classes.
- Interfaces make it possible to interchange implementations (at runtime) without breaking client code.

```
class Server {
   public Status getStatus() { ... }
}

class Client {
   Server s = new Server(...);
   System.out.println("Server:"+s.getStatus());
}
```



Interfaces

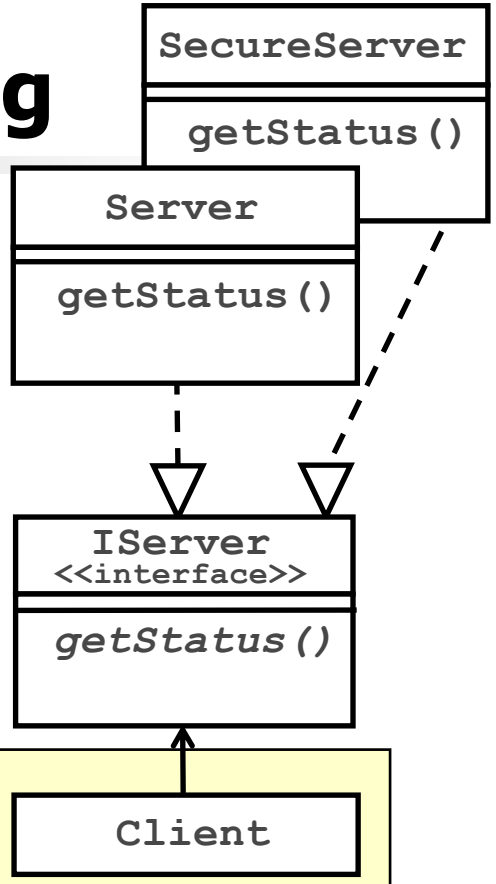
```
class Server implements MsgServer {
   public Status getStatus() { ... }
}

interface MsgServer {
   public Status getStatus();
}

class Client {
   MsgServer is = new Server(...);
   System.out.println("Server:"+is.getStatus());
}
```



"Program to an interface, not to an implementation."

Interfaces & Dynamic Class Loading

```
class Server implements IServer {
   public Status getStatus() { ... }
}
class SecureServer implements IServer {
   public Status getStatus() { ... }
}
interface Iserver {
   public Status getStatus();
}
```




```
class Client {
 public static void main(String[] args){
  try {
   IServer is = (Calendar.getInstance().get(Calendar.HOUR_OF_DAY)> 10) ?
                (IServer) Class.forName("Server").newInstance() :
                (IServer) Class.forName("SecureServer").newInstance();
   System.out.println("Server:"+is.getStatus());
  }
  catch (Exception e) { System.out.println(e.getMessage()); }
 }
}
```

Interfaces

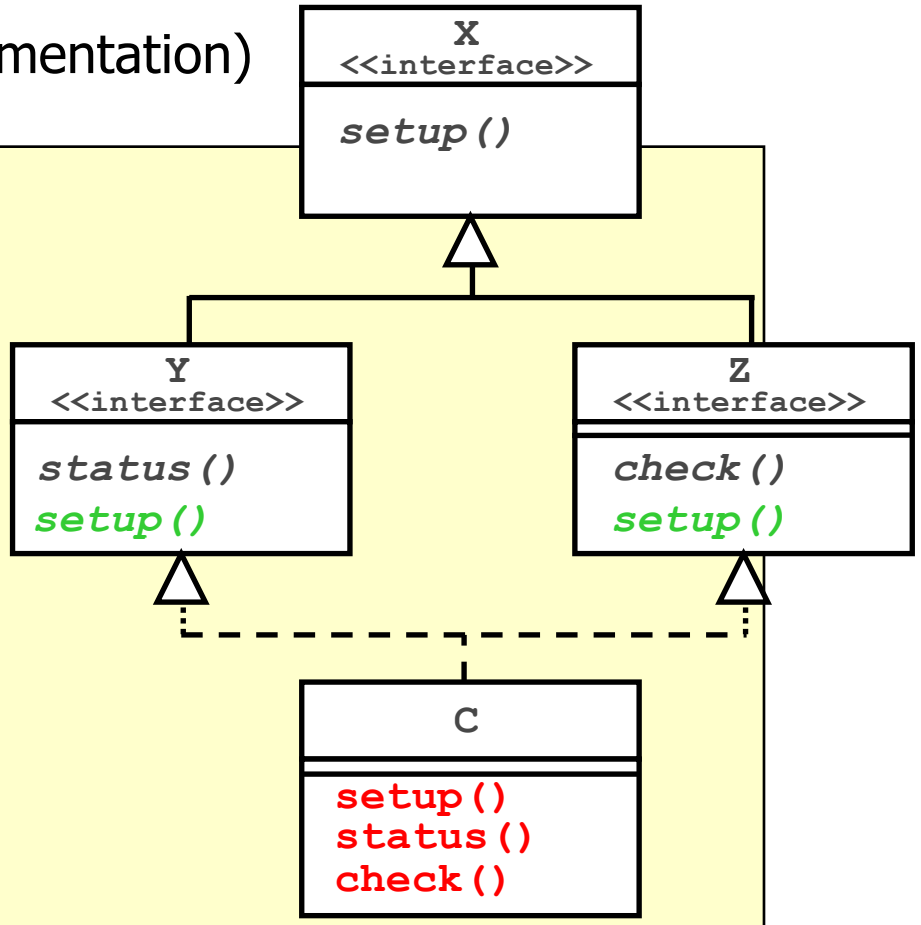
- Interfaces can be extended similarly to classes.
- However, an interface can extend multiple other interfaces.
- Interfaces thus support multiple inheritance of behavior, as opposed to *multiple inheritance of implementation*, which is not possible in Java.

```
interface A {
      void a();
}

interface B {
      void b();
}

interface C extends A, B {          // C inherits a() und b()
       void c();
}
```

Interfaces – Diamond Problem

- Multiple inheritance of behavior (not implementation)

```
interface X {
   void setup();
}
interface Y extends X {
   void status();
}
interface Z extends X {
   void check();
}

class C implements Y,Z {
    public void setup() { ... }    // implementation
    public void status() { ... }   // implementation
    public void check() { ... }    // implementation
}
```



Interfaces

- Default methods are (non-abstract) interface methods that provide a default implementation.
- Default methods have been introduced to support seamless interface evolution, since they allow to add new methods to interfaces without breaking existing client code (binary compatibility).
- Simplify implementation of interfaces offering standardized implementations.

```
interface I {
   void i();
   default void j() { ... // implementation of j() }
}


class C implements I {     // C inherits impl. of j()
   void i() { ...          // implementation of i() }
}
```

Interfaces

Extending an interface that has default methods:

- Do nothing – default methods will be inherited
- Re-declaration of the default methods makes them abstract.
- Re-definition of the default methods overrides them.

```
interface I {
   default void i() { ... }
}

interface J extends I {
   void i();   // i() ist nun wieder abstrakt }
}

interface K extends I {
   default void i() { ... } // i() neu implementiert
}
```

Interfaces: Multiple Inheritance – Default Methods

```
interface I {
   default void m() { System.out.println("I.m()"); }
}
```

```
interface J {
   default void m() { System.out.println("J.m()"); }
}
```

```
public class MyClass implements I, J {   // compiler error

   ...

}
```

- Compiler error since duplicate default methods named `m` with same signature are inherited from the types `J` and `I`.

Interfaces: Multiple Inheritance – Default Methods

```
interface I {
   default void m() { System.out.println("I.m()"); }
}
```

```
interface J {
   default void m() { System.out.println("J.m()"); }
}
```

```
class MyClass implements I, J {
   ...

   @Override
   public void m() {
      I.super.m();          // use default implementation of I
   }

}
```

- Resolve conflict through method overriding.

Multiple Inheritance: State, Type, Implementation

- Multiple inheritance of state
  - not supported in Java
- Multiple inheritance of type (behavior)
  - A class can implement multiple interfaces
- Multiple inheritance of implementation
  - Only possible for interfaces with default/static methods.
  - If a class implements multiple interfaces containing methods the same default/static methods, the compiler or user has to resolve conflicts.

See: https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

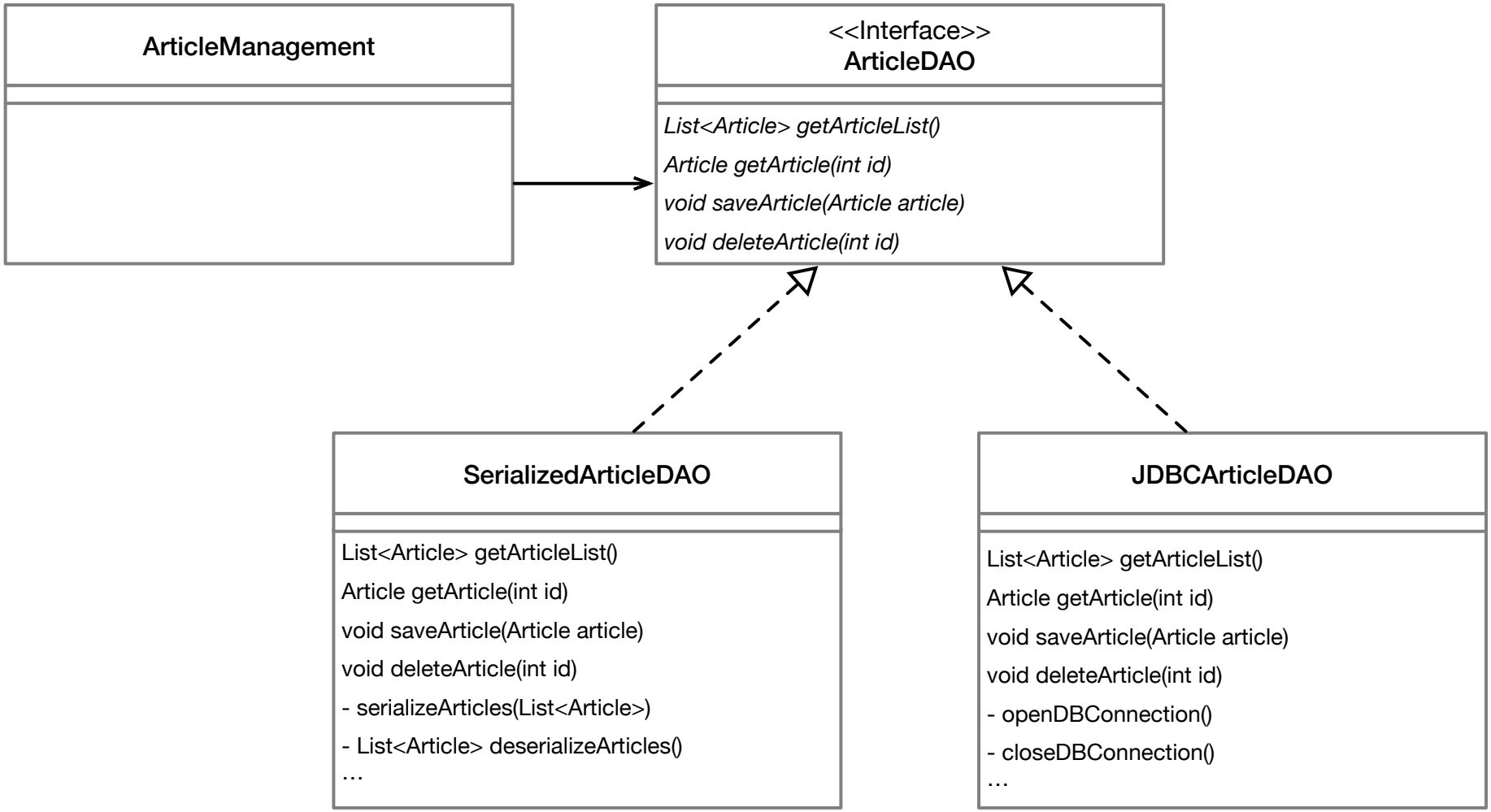
Interfaces

- Interfaces play an important role in the Java API
  - Collections: **`Collection`**, **`Set`**, **`List`**, **`Queue`**, **`Iterator`**, **`Enumeration`**, **`List`**, …
  - Serialization: **`Serializable`**
  - Eventhandling: **`EventListener`**, **`ActionListener`**, **`KeyListener`**, **`MouseListener`**, **`MouseMotionListener`**, **…**
  - Multithreading: **`Runnable`**

"Program to an interface, not to an implementation."

Interfaces – DAO Pattern

- The Data Access Object (DAO) pattern decouples business logic from the implementation details of the persistence layer (RDBMS, object store, ...).



DAO Pattern

- How is the implementation of the DAO interface provided?

```
public class ArticleManagement {
  private SerializedArticleDAO articleDAO;
  public ArticleManagement(String filename) {
    this.articleDAO = new SerializedArticleDAO(filename);
  }
...
}
```

- Selection of implementation class is hard-coded in **`ArticleManagement`**.
- Better solution?
- → **Dependency Injection**

DAO Pattern – Dependency Injection

- Constructor of **`ArticleManagement`** parameterized with DAO interface.
- Selection of implementation class delegated to **`ArticleCLI`**.
- **`ArticleManagement`** is now independent of DAO implementation.

```
public class ArticleManagement {
  private ArticleDAO articleDAO;
  public ArticleManagement(ArticleDAO articleDAO) {
    this.articleDAO = articleDAO;
  } …
```

```
public class ArticleCLI {
  private ArticleManagement mgmt;
  ...
    mgmt = new ArticleManagement(
              new SerializedArticleDAO(filename));
```

Dependency Injection

- Dependency injection (DI) is the dynamic provision of information about required classes (components) at runtime by a framework or container.

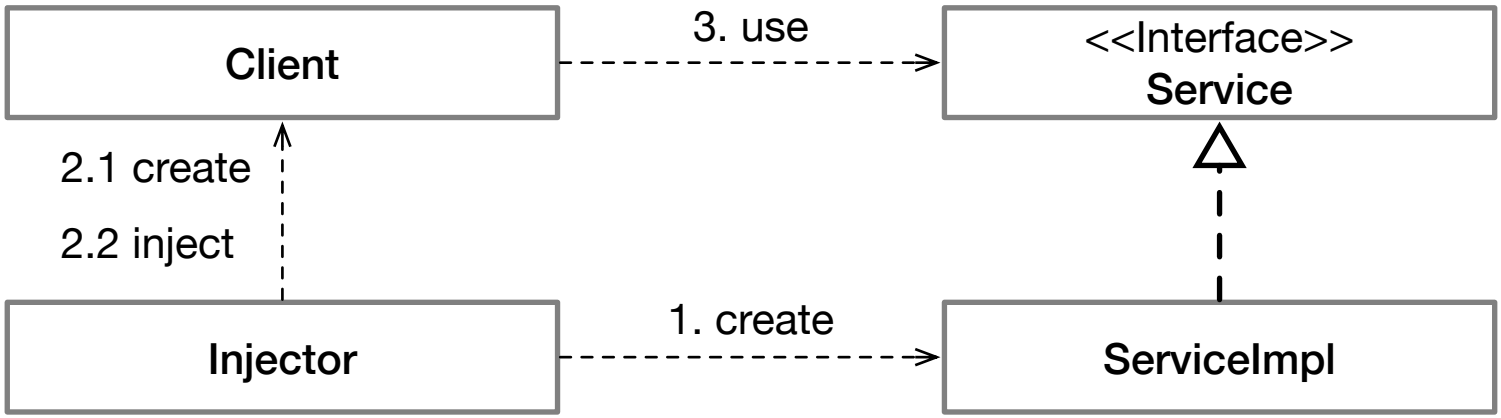
- Types of dependency injection
  - Constructor Injection
  - Setter Injection
  - Interface Injection

- Examples of containers/frameworks that support DI
  - Jakarta Enterprise Beans (Enterprise Java Beans)
  - Pico Container
  - Spring Framework

See: https://martinfowler.com/articles/injection.html

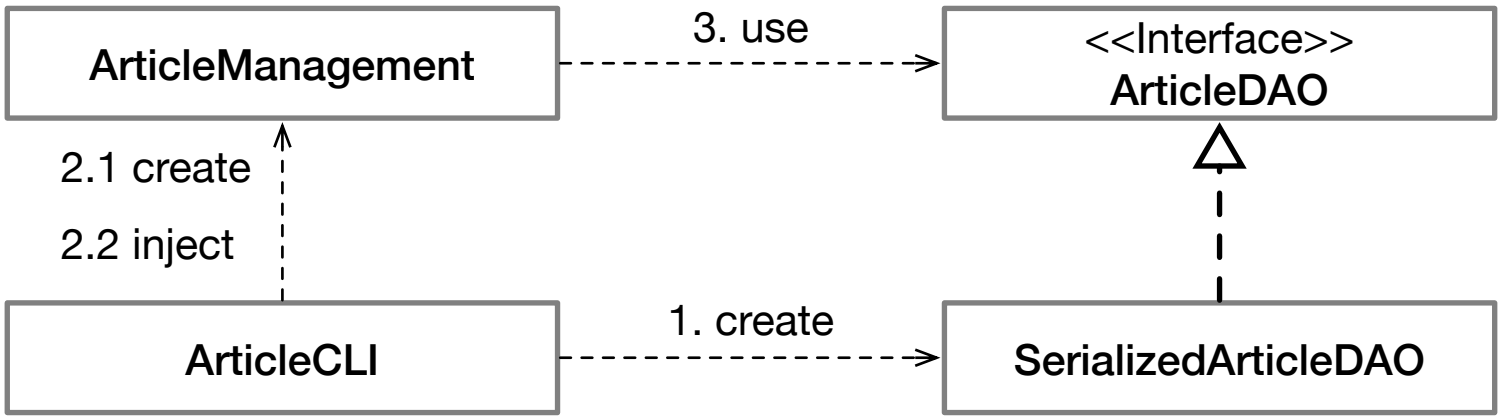
Dependency Injection

General Scheme



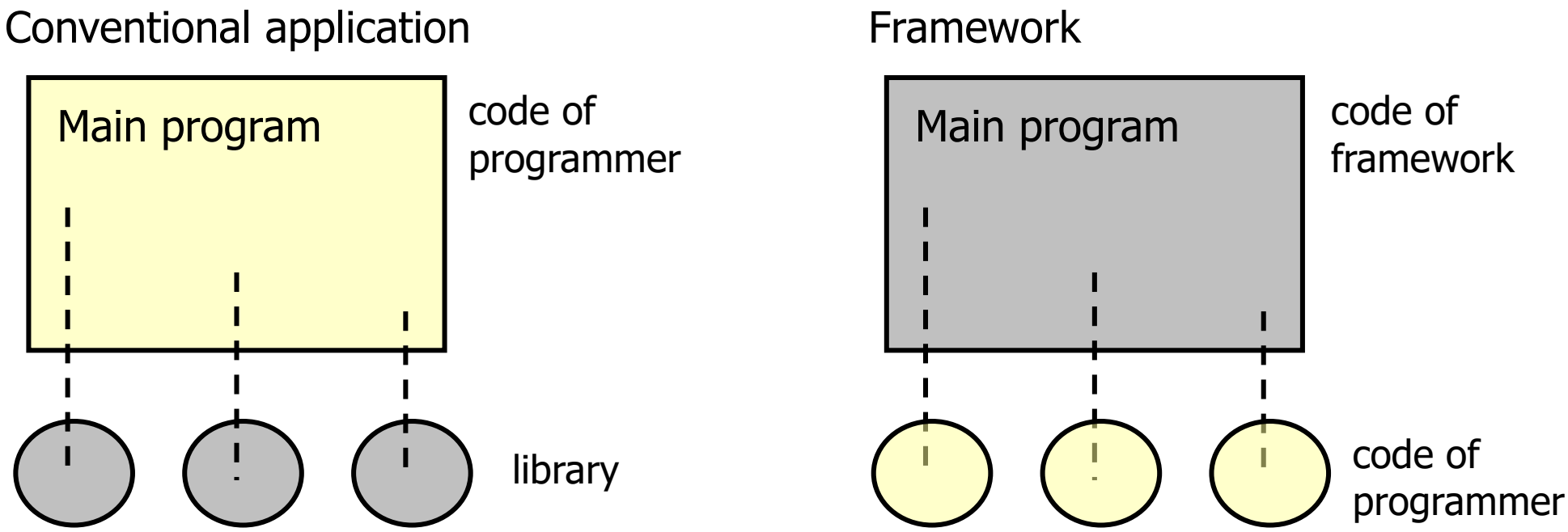
Dependency Injection

Example: Assignment 1



Inversion of Control

- Dependency Injection is often referred to Inversion of Control (IoC).
- IoC is a concept where control over the application logic and application objects is delegated to a framework or container.



Hollywood Principle: "Don't call us, we'll call you."

Setter Injection

```
public class OrderService {
   IOrderDB orderDB = new OrderDB(…);     // class to access DB
                                          // is hardcoded
   public Order saveOrder(Order order) {
          orderDB.save(order);
   }
}
```

```
public class OrderServiceDI {
   IOrderDB orderDB;

   public void setOrderDB(IOrderDB orderDB) { // obtain ref.
          this.orderDB = orderDB;             // at runtime
   }
   public Order saveOrder(Order order) {
          orderDB.save(order);
   }
}
```

Dependency Injection – Jakarta Enterprise Beans

- Injecting EJBs, resource objects, JNDI objects, etc. via annotations.
- The @Resource annotation injects service objects and JNDI objects

```
public class FooDao {
    @Resource (name="DefaultDS")
    DataSource myDb;
    ...
}
```

direct field variable injection

```
@Resource
public void setSessionContext (SessionContext ctx) {
    sessionCtx = ctx;
}
```

Setter injection

Dependency Injection

- Decoupling of object locators and object creators from business logic
- Loosely coupled architecture
- Better maintainability
- Better reuse
- Simplified software testing (mock objects)