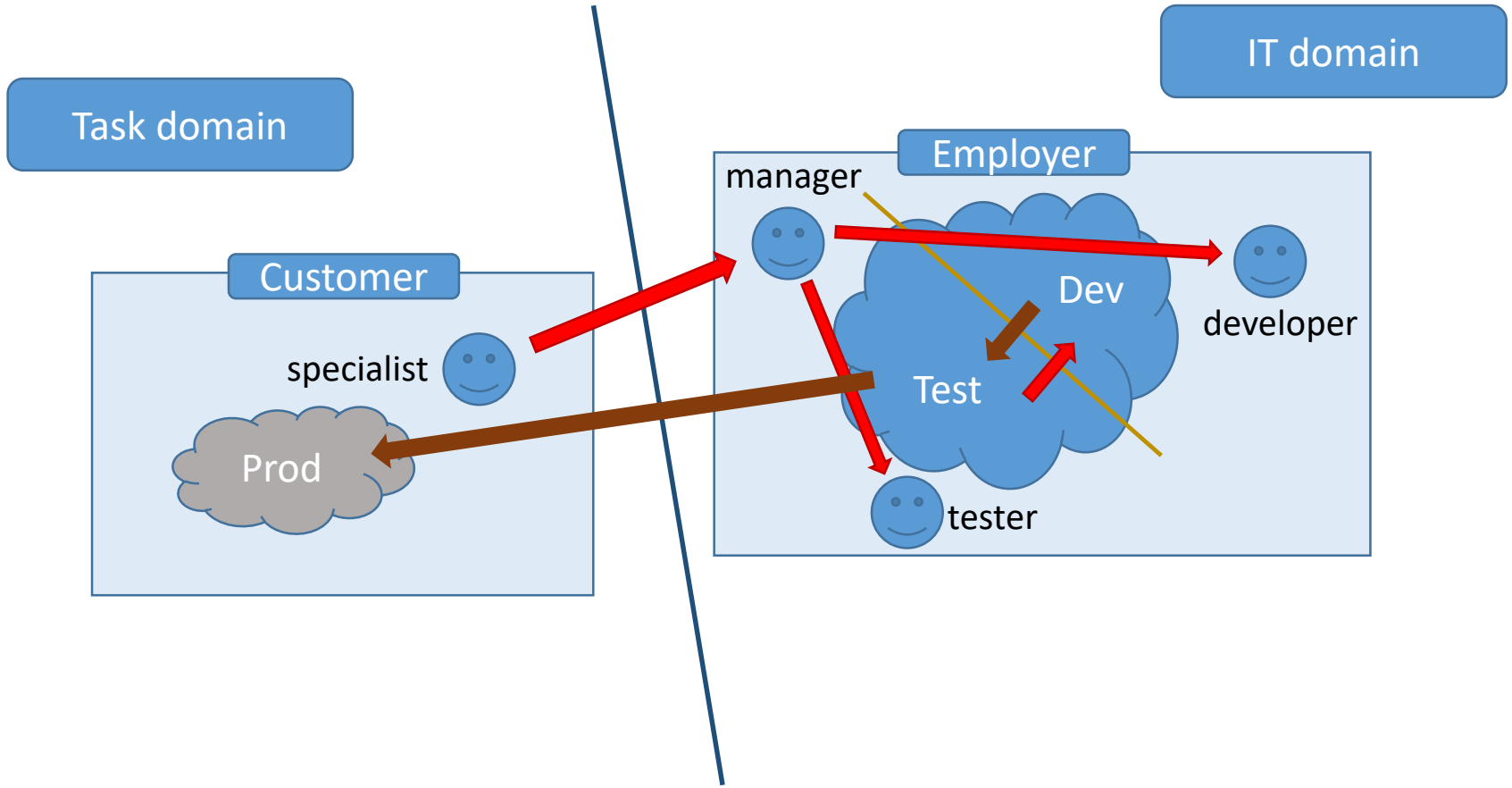
Object oriented programming

(OOP)

Tamás Storcz

OOP motivation
Business structure



OOP motivation

- human:
  - Interesting – burnout
  - Less effort – Laziness – do not work too much…
- business:
  - employer – quick work
  - employee – interesting tasks
  - customer – accurate solution (errorless)

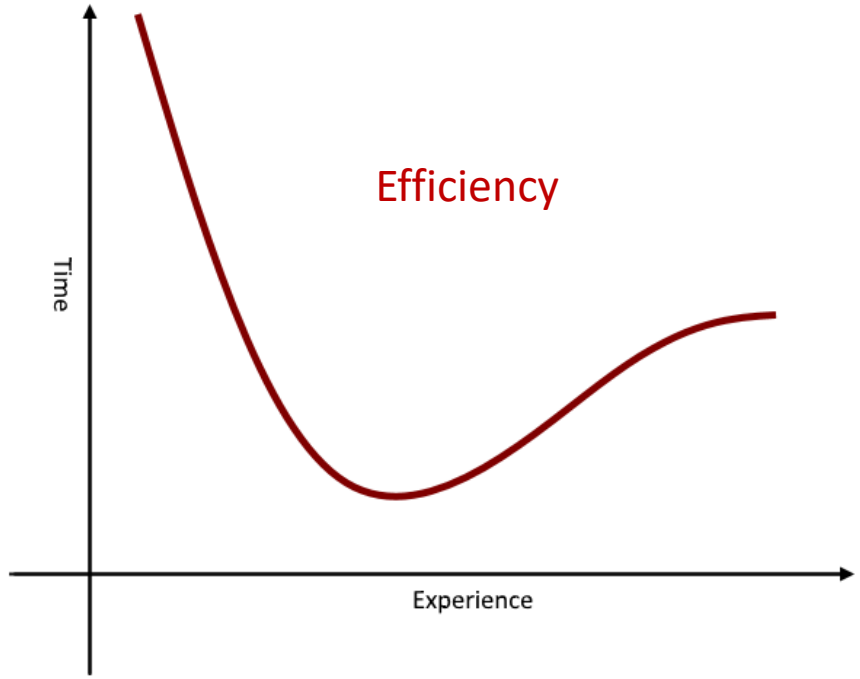
Efficiency

Interesting – flow

Burn-out

- Repeated tasks
- Same level of complexity
- No joy

- No measure of performance
- No progress



Efficient programming - perspectives

What is efficient programming?

- Using minimal number of instructions
- Quickest execution
- Lowest memory consumption
- Maximum security
- Continuous availability
- Shortest development time
- Sustainable source code

Compiler

API

Developer

Efficient programming

Aim: optimal resource utilization

What is the resource bottleneck?

From examination of projects with different size and complexity, considering solution quality and, project sustainability, de development bottleneck is the **developer worktime**.

Main tasks of developers

- Creating new from scratch
  - Design and implementation
- Modifying an existing solution – usually part of a framework
  - Find and fix errors – error reports
  - Modify existing operation – requirement changes
  - Extend functionality – new requirements

Understand requirements, find solution, implement, test, deliver

Efficient programming

To program efficiently, use **as little** developer resource, as possible for

- understanding task domain knowledge
- interpreting of **previous implementations**
- writing **boilerplates**
- updating documentations

Efficient programming – coding

Rate of reading : writing code when programming

10:1

Modules and names → Readable code

Common languge and thinking schema → GOOD names

Safe reuse of components (API + extensions)

Readable code – Monolithic

```
do {
    n = s.nextInt();
} while (n <= 0 || n > max);
int f = 1;
for (int i = 1; i <= n; i++) {
    f *= i;
}
System.out.println(n + "!=" + f);
```

Readable code – Modular + names

```
int n = getPositiveIntFromConsole(
   message: "Please enter n!",
   max: 10
);
System.out.println(
   n + "!="
   + calculateFactorial(n)
);
```

Modelling

Aim: to define a closed system, simplier than reality, in which the task can be solved well enough (time, accuracy, resources).

Model: A simplified interpretation of reality.

Model design:

- Specify interesting properties, behaviors
- Classification and class relations
- Algorithms and data structures

Objects and Classification

Objects are *classified* based on **type and meaning** of their properties and behaviors – If objects have same properties and behaviors, they belong to the same class.

Objects are instances of classes.

Objects are *distinguished* by **their place of existence (unique id)**. Values of their important properties could be the same.

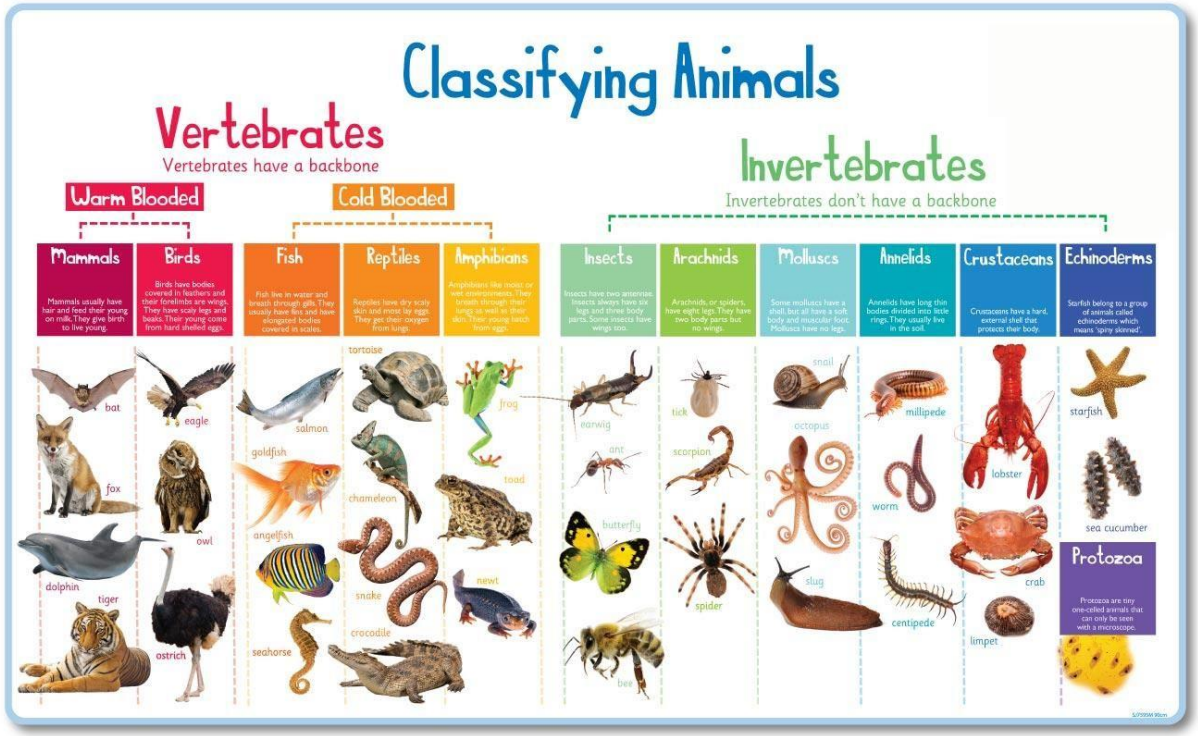
Abstraction

The main tool of object creation and classification.

Abstraction of properties: Prescinding from property values, classification can be done. (Based on properties and behaviors.)

Abstraction level: On higher abstraction levels prescinding from less important properties and behaviors, class hierarchy creation is available.

Abstraction – example



Object oriented programming

Object oriented programming is a programming methodology.

Most commonly used paradigm of nowadays.

It is focusing on design and implementation of connected program **components** – **objects**, **hierarchically** classified by their properties and behaviours.

Object oriented programming – for who?

- Processor instruction set is very well defined
- Low level programming:
  Work with processor instruction sets
- High level programming: avoid boilerplates, use APIs
  Also translated to processor instructions to execute

Programming paradigms are created for programmers only – not for the machines

OOP executable is NOT better!!!

Message in OOP

- When an object receives a message, it executes a behavior
- Message can have details, which the behavior receives and processes

- This is indeed the fancy name for method calls…

OOP principals – meaning

- **Encapsulation**

  Encapsulate logically related data and their operation into a closed unit.
- **Inheritance**

  Derived classes inherit all properties and behaviors of parents, and they can define new ones.
- **Polymorphism**

  Inherited behaviors of derived classes can be modified, therefore they can response differently to the same *message*.

OOP principals - benefits

- **Encapsulation**

  Single responsibility, consistency, abstraction, safe reuse
- **Inheritance**

  Functions of a reused item can be extended
- **Polymorphism**

  Specialization of operation (specialized tasks of narrowing abstraction)

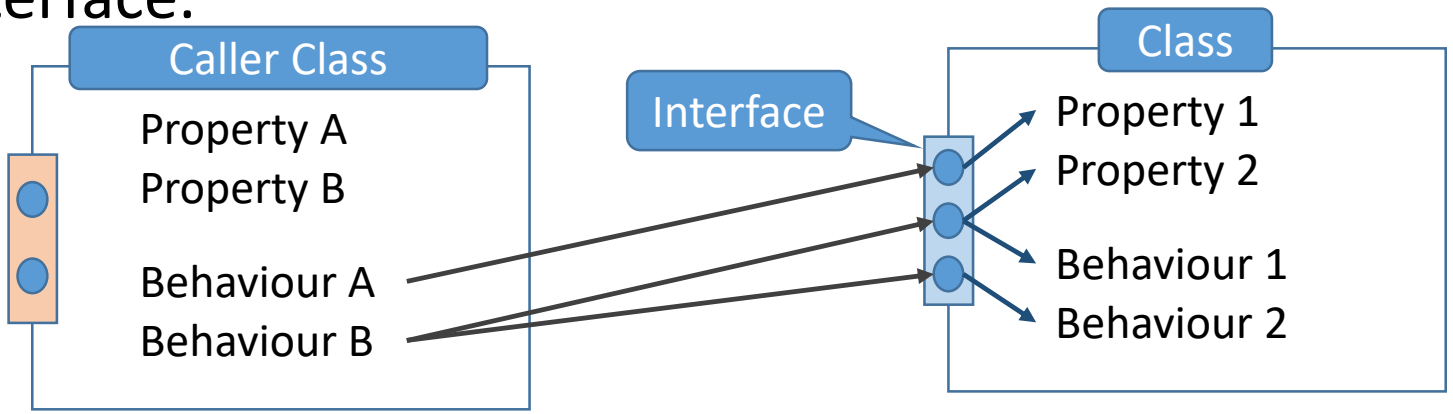
  Exceptions of reuse – behave differently

Interface

Structural definition of a one direction connection surface. Declares the usage modes of the program component – object – which ***implements*** the interface.

Abstract descriptor of a one direction communication channel.

A contract about expected behaviors of a class, which ***implements*** the given interface.



Encapsulation

Logically connected data and operations working on them are handled as one closed unit.

Only a surface is provided to utilize its services by *semi-explicit* interface definition.

Benefits

- Consistency – No contradiction in internal state descriptors – controlled state transitions
- Abstraction – Interface and usage is independent from implementation
- Secure reuse – Objects of same class are expected to work the same way
- Single responsibility – On a given abstraction level, the object is responsible for only one thing

Data encapsulation

Classic C structures (struct)

- Logically related data encapsulated in a distinct unit.

Operation

- Declaration
- Initialization
- Uncontrolled state changes

C structure

```
struct Document {
        char* title;
        char* description;
        char* content;
} Document;

void appendLine(
        Document doc, char*
newLine
) {…}
```

Example from a medical experiment

```
struct Person{
    int age;
    int bmi;
    bool male;
    int numberOfPragnancy;
    bool diabetes;
} Person;
```

Uncontrolled state change

Data & behavior encapsulation

Class

A template of objects with same descriptors and behaviors. Internal state of objects of the same class could differ, but descriptors match.

The template contains:

- Internal state descriptors (logically related data)
- Predefined **initial state**
- **Behaviors** (projectors or state transitions)

Class declaration

```
class Document {
      public String title;
      pubilc String
description;
      public String content;

      public void appendLine(
            String newLine
      ) {…}
}
```

```
class Person{
   …
   bool male;
   int numberOfPragnancy;
   …
   void setPragnancy(int count)
   {
      if(male && count > 0)ERROR!
   }
}
```

Controlled state change

Consistent internal state

- Why internal?
  - The state is object specific, exactly not known by the environment, because state descriptors are not accessible
- Why consistent?
  - State descriptors always depict a valid state
- How to provide consistency?
  - Predefined initial state
  - Controlled state transitions – controlled state descriptor modifications

Initialization – Special methods

Handlers of lifecycle events:

- Creation – Constructor – A method for object initialization
- Termination – Destructor – there is no destructor in Java

Constructor

```
class Document {
        public String title;
        pubilc String description;
        public String content;
        //constructor
}
```

```
public Document() {
        title = "";
        description = "";
        content = "";
}
```

- Same name as the class
- No return type

Default constructor

JVM requires constructor for object creation

- Programmer can define constructor explicitly
- A default constructor is added implicitly

```
class Document {
    public Document() {}
}
```

Encapsulation – Consistency

Data hiding: Which state descriptors should be hidden from environment?

Hide all to keep the access really abstract. (independent from implementation)

Use the resource via this abstract interface in a controlled way.

Data hiding

How to hide data?

- Use `private` modifier to define a component available only inside the object
- Use `public` modifier to define a component available both inside and outside the object

Data access

- Internal state descriptors are hidden – called: data members
- Properties available via methods – no direct data property
- Public getter methods

```
public <property type> get<property name>()
```

- Public setter methods

```
public void set<property name>(<property type>
newValue)
```

Getters and setters

```
class Document {
	private String title;
	public String getTitle() {
		return title;
	}
	public void setTitle(String newTitle) {
		title = newTitle;
	}
}
```

Implicit class interface

Access from outside the class:

```
class Document {
      String getTitle();
      void setTitle(String newTitle);
}
```

Abstract interface

Depends on implementation

```
class Point {
        public double X;
        public double Y;
        public double getRadius();
        public double getAngle();
}
```

Independent from implementation

```
class Point {
    private double X;
    private double Y;
    public double getX();
    public double getY();
    public double getRadius();
    public double getAngle();
}
```

Reuse I – Classes and objects

Classes are templates for objects of same type (same properties and behaviors).

This is the first level of code reuse.

Objects are **instance**s of classes

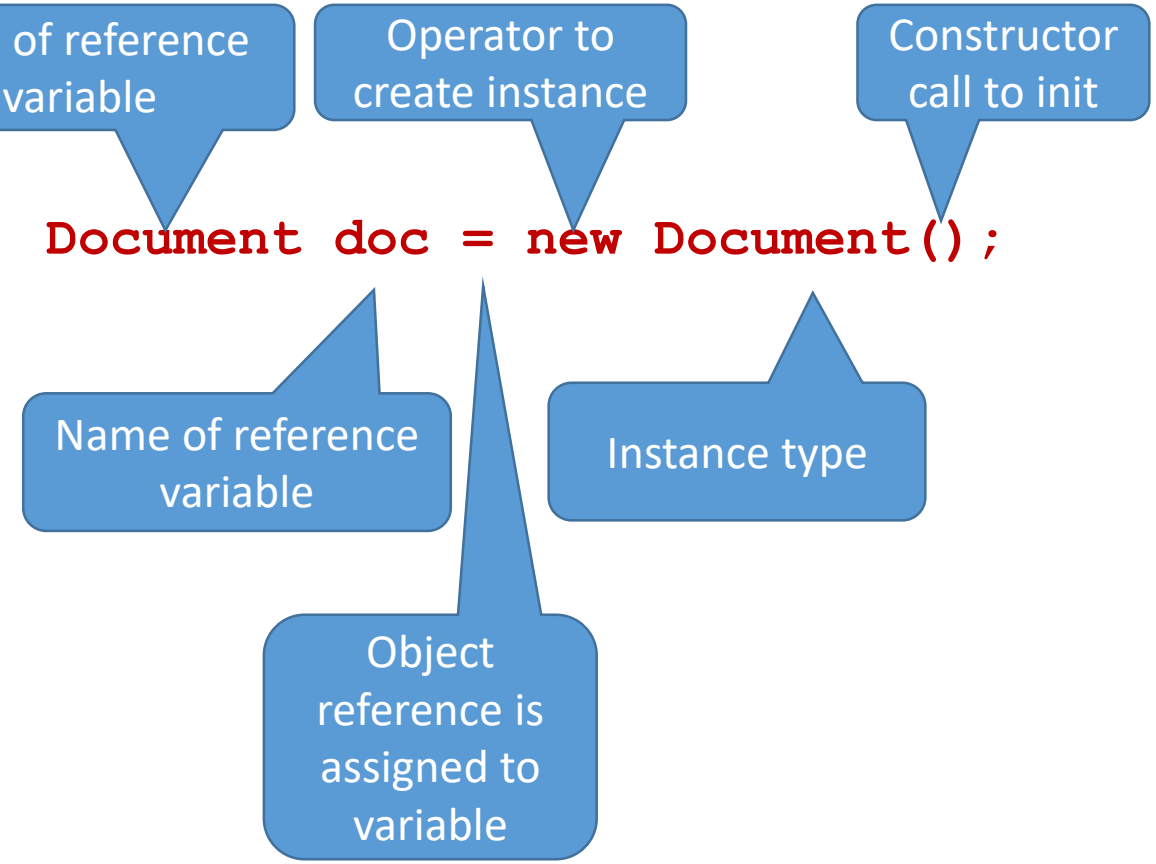
All objects of the same type have:

- Custom values of common properties – instance data, instance state
- Shared methods, working on instance data
- Controlled state transitions – consistent internal state descriptors

Instantiation

```
class Document {
        public String title;
        pubilc String description;
        public String content;

        public Document() {…}
}
```



Single responsibility

On a **certain level** of abstraction, the object is responsible for only a **single thing** – When the class has to be changed?

One of the most challenging parts of OOP! Could change through development.

Store users: User, User collection, Serialization, Storage management

Coherence

Class

- *prop_1, prop_2*, *prop_3*
- method_1 – *prop_1, prop_2*
- method_2 – *prop_2*
- method_3 – *prop_3*

Class_1

- *prop_1, prop_2*
- Method_1 – *prop_1, prop_2*
- Method_2 – *prop_2*

Class_2

- *prop_3*
- Method_3 – *prop_3*

struct vs. class

struct

- Value type
- Limited functionality
- On assignment, value is duplicated
- Requires space only for data
- When storing lots of data, space-saving
- Java has no struct

Class

- Reference type
- Full functionality
- On assignment, value is NOT duplicated
- Requires space for reference also
- Reorder is quicker
- **Java has class**

OOP – Summary

Target

Only developers by source code organization

Tools

- Define secure template – Encapsulation
- Extend functionality – Inheritance
- Specialize functionality – Polymorphism

Aim

Preparation of readable, maintainable, testable code which is reliable and safely reusable