Concepts of Object-Oriented Programming

Richard Berger

richard.berger@jku.at

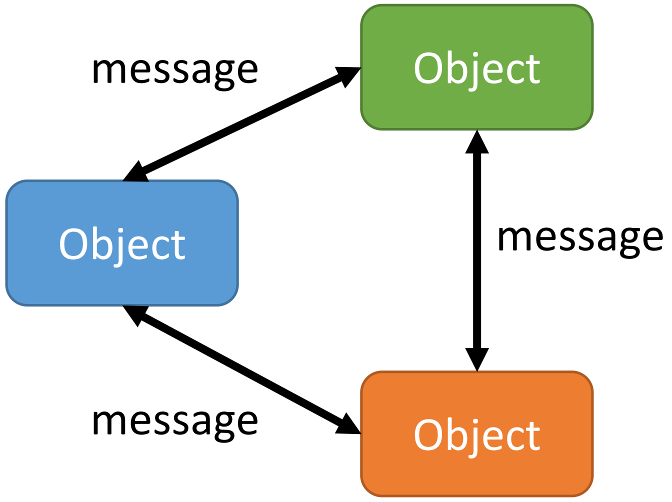
Johannes Kepler University Linz

What this talk is about

- Introduction to Objects & Classes
- Building objects
    - Composition
    - Encapsulation
    - Inheritance
    - Polymorphism
- Best practices
    - Recommendations
    - Design Patterns
- Conclusion
    - Pro & Cons of OOP

What is Object-Oriented Programming?

- OOP started in 1960s (Simula, SmallTalk)
- **Using objects as basic unit of computation**
- Allows to extend type system
  - Usage: just like basic types (int, double, float, char, …)
  - But with own structure and behavior
- **Static Languages** (C++)
  Types are known at compile time
- **Dynamic Languages** (Python)
  Types can be manipulated at runtime



I. Objects & Classes

Defining new objects, object lifetime

Where are these **“objects”**?

- Objects exist in **memory** at runtime
- Just like objects of primitive types (integers, floating-point numbers)
  - We can interpret 4 bytes of data as integer number
  - We can interpret 8 bytes of data as floating point number
- In C we have structs to create composite types containing multiple primitive types
- In C++ and other OOP languages this is further extended by associating behavior to a chunk of data through specifying methods to manipulate that data.

Object

- **State**
  - all properties of an object
- **Behavior**
  - How an object reacts to interactions, such as calling a certain method
  - In OOP speak: Response of an object when sending it messages
- **Identity**
  - Multiple objects can have the same state and behavior, but each one is a unique entity

Structure and Behavior of similar objects is defined by their **class**.

An object is also called an **instance** of a class.

Classes

```
class Vector2D
{
public:
  double x;
  double y;

  double length()
  {
    return sqrt(x*x + y*y)
  }
};
```

Class

- defines memory structure of objects
- how we can interact with objects
- how it reacts to interactions

Member Variable

- A variable in the scope of a class
- All instances allocate memory for their variables

Member Method

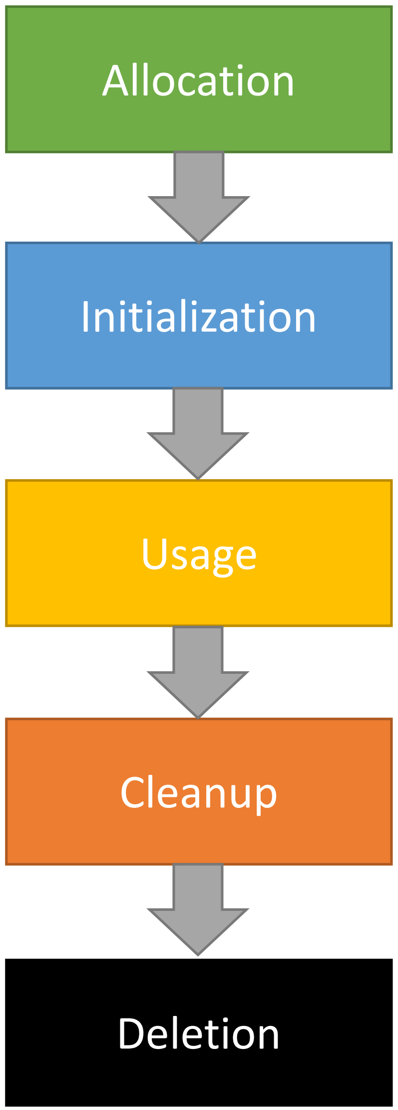
- A method which can be called for an object of the class.
- Can access and modify the object state by manipulating member variables.

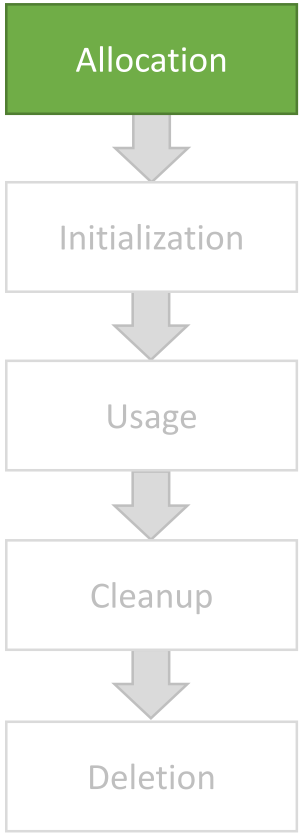
Interface

- All methods which can be called on an object from outside.

Lifetime of an Object

- **Allocation**
  - Allocate enough memory to store object data/state
- **Initialization**
  - Set an initial object state
- **Usage**
  - Interact with objects through methods
  - Access and modify object data
- **Cleanup**
  - Make sure that everything is in order before deletion
- **Deletion**
  - Memory is freed, object ceases to exist



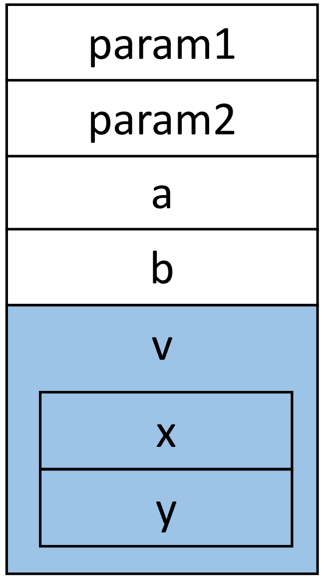


Allocation

Static / Stack Allocation

```
void foo(int param1, int param2)
{
   double a = 30.0;
   double b = 50.0;

   Vector2D v;
   …
}
```

Stack

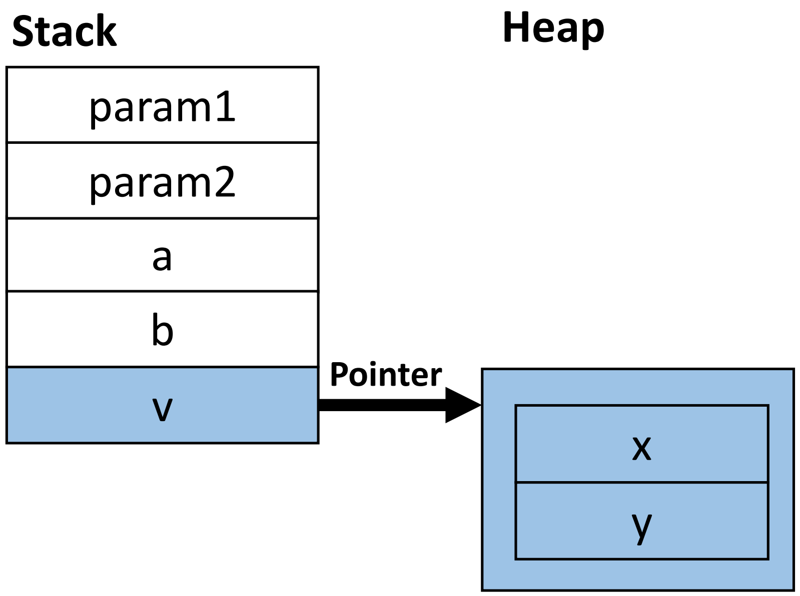


\+ Fast
\+ Automatic cleanup
\- Not persistent
\- “Limited” storage

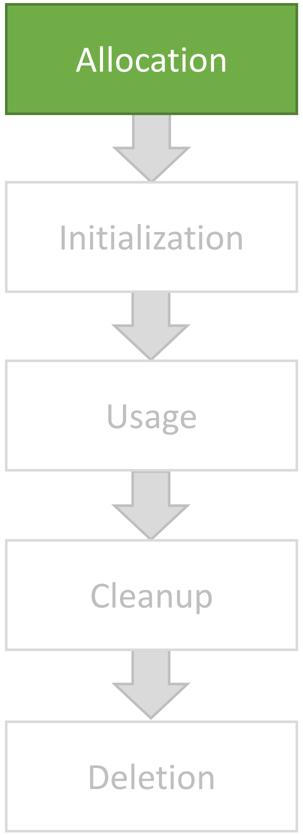
Allocation

Dynamic / Heap Allocation

```
void foo(int param1, int param2)
{
   double a = 30.0;
   double b = 50.0;

   Vector2D * v = new Vector2D;

   …
}
```



+ Persistent
+ “Infinite” storage
- Slow
- Manual cleanup



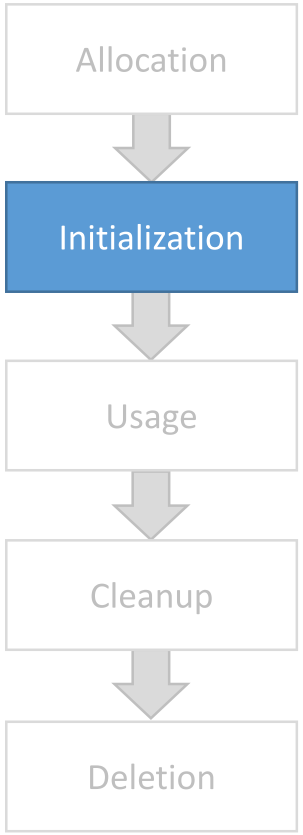
Initialization

```cpp
class Vector2D {
public:
   Vector2D(){
      x = 0.0;
      y = 0.0;
   }
   Vector2D(double x1, double y1){
       x = x1;
       y = y1;
   }
};
```

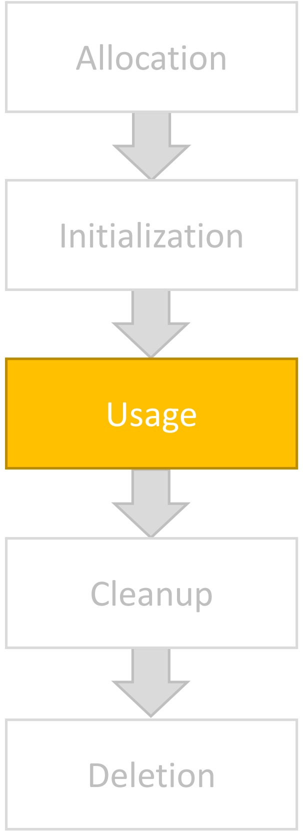
Constructors

- special methods which initialize an instance of a class
- multiple variants with different parameters possible
- initialize member variables

```cpp
Vector2D v1();
Vector2D v2(10.0, 20.0);

Vector2D * v3 = new Vector2D();
Vector2D * v3 = new Vector2D(10.0, 20.0);
```



Usage



Stack Objects

```
Vector2D v;

// access members
v.x = 10.0;
v.y = 20.0;

// call member functions
double len = v.length();
```

Heap Objects

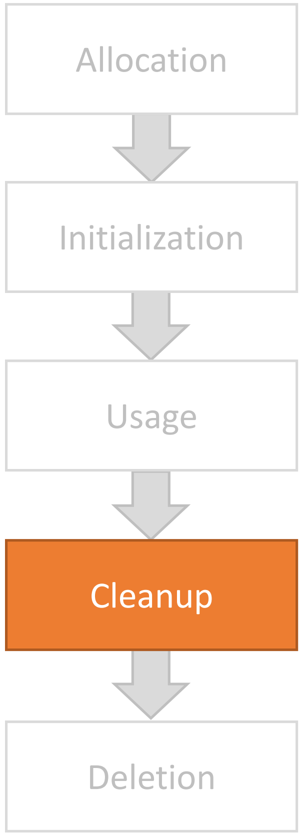
```
Vector2D * pv = new Vector2D();

// access members
pv->x = 10.0;
pv->y = 20.0;

// call member functions
double len = pv->length();
```

Cleanup

```
class MyVector {
  double * data;
public:
  MyVector(int dim){
    data = new double[dim];
  }
  ~MyVector() {
     delete [] data;
  }
};
```

Destructor

- Cleanup before destruction
- Free any acquired resources (file handles, heap memory)

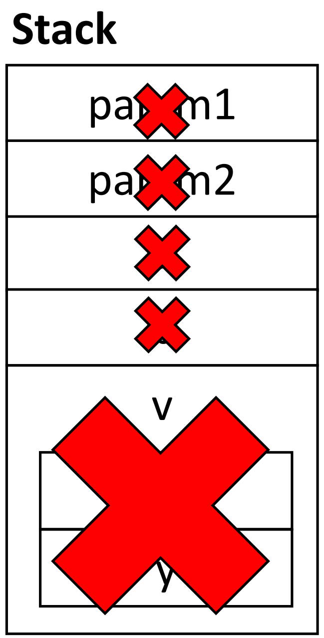


Deletion

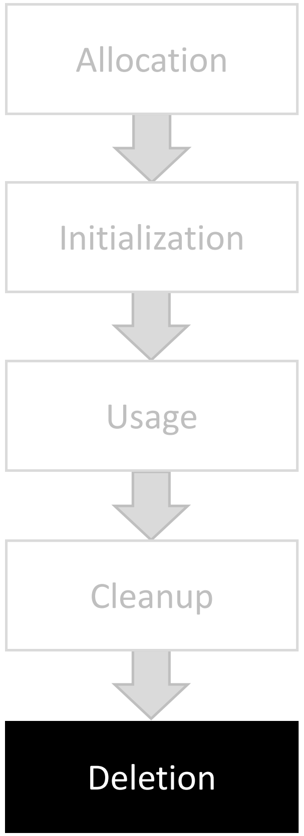
Static / Stack Objects

```
void foo(int param1, int param2)
{
   double a = 30.0;
   double b = 50.0;

   Vector2D v;
   v.x = 10.0;
   v.y = 20.0;

   …
}
```

End of Scope



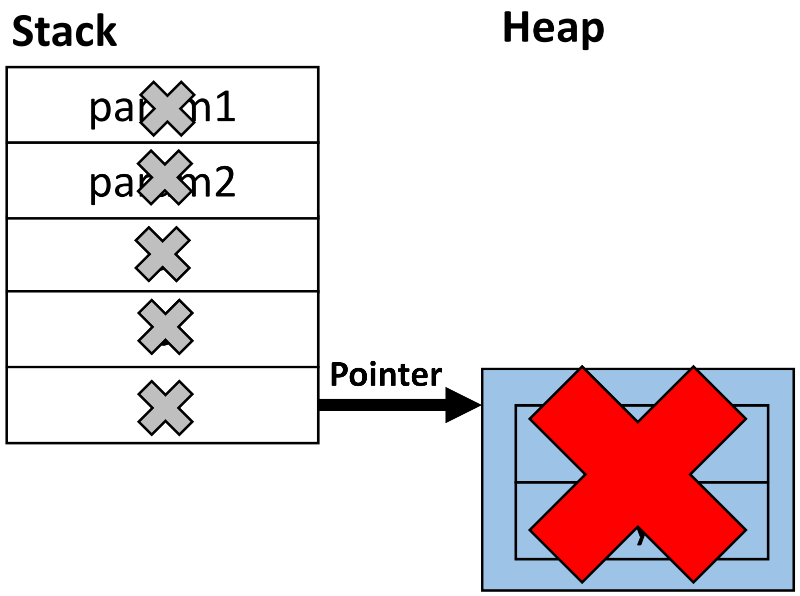
objects on **stack** are automatically deleted at end of scope



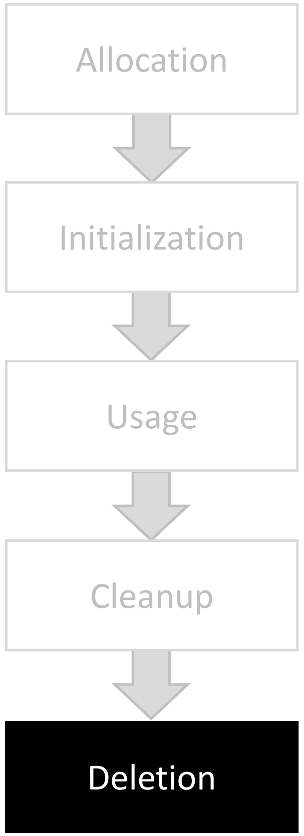
Deletion

Dynamic / Heap Objects

```
void foo(int param1, int param2)
{
   double a = 30.0;
   double b = 50.0;

   Vector2D * v = new Vector2D;
   v->x = 10.0;
   v->y = 20.0;

   …

   delete v;
}
```



objects on **heap** must be deleted **explicitly**



Static class members

```
class Point2D
{
  static int numPoints = 0;
public:
  int identifier;
  double x;
  double y;

  Point2D(double x1, double y1) {
     identifier = numPoints++;
     x = x1;
     y = y1;
  }

  static void reset_count() {
     numPoints = 0;
  }
};
```

Static Member Variable

- Only a single instance of this variables exists
- Much like a global variable
- Can be accessed by all instances
- Access by class name using ::, not instance

```
// Definition in a single .cpp file
int Point2D::numPoints = 0;

// access without having an instance
cout << Point2D::numPoints << endl;
```

Static Method

- A method which is not applied on a class instance, but on the class itself
- Much like a global function
- Can only access and modify static member variables

```
// call static method, no instance required
Point2D::reset_count();
```

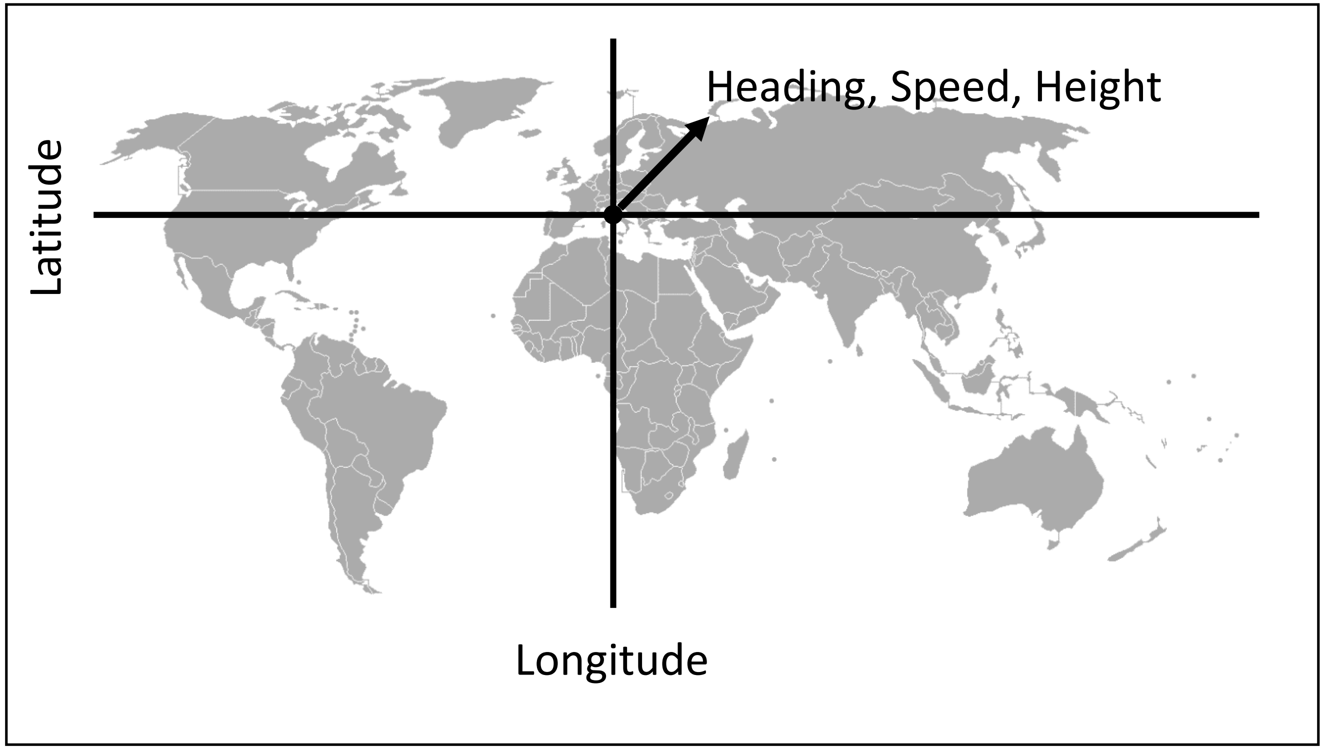
II. Building objects

Composition, Encapsulation, Inheritance, Polymorphism

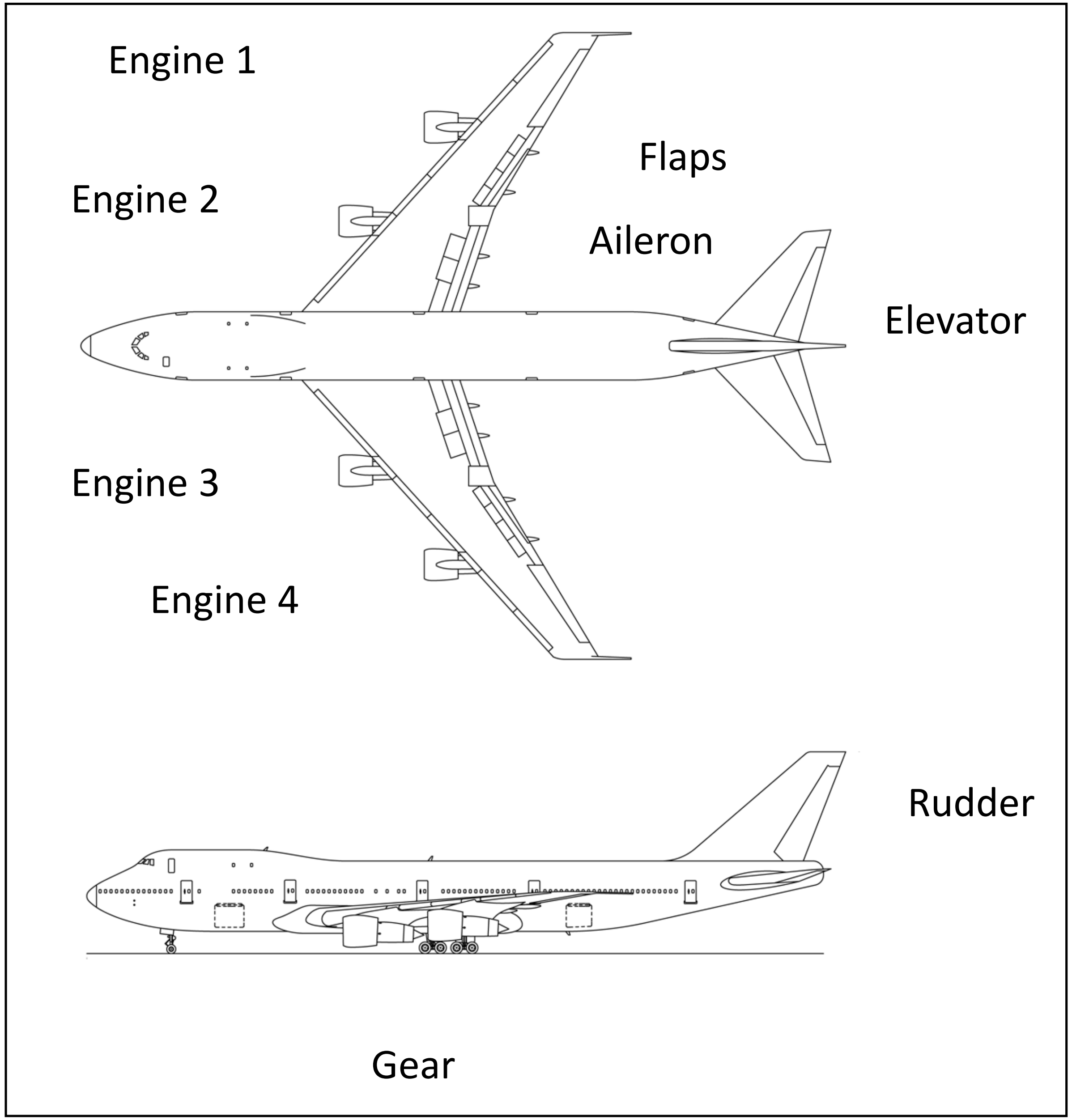
Abstractions

model domain concepts as classes

High-Level Model of an Airplane



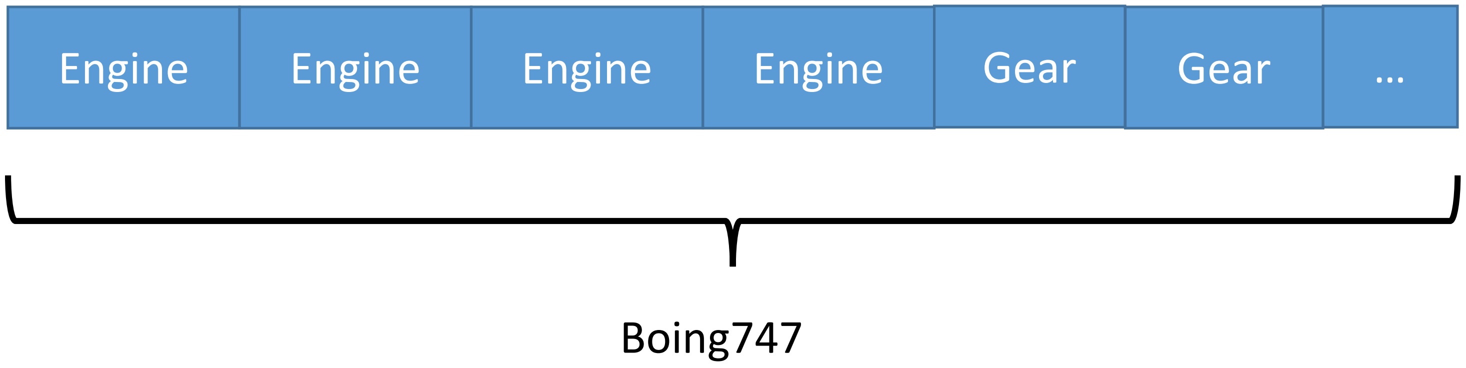
Low-Level Model of an Airplane



Composition

```
class Boeing747
{
      Engine engine1;
      Engine engine2;
      Engine engine3;
      Engine engine4;
      Gear frontGear;
      Gear backGear;
      …
};
```

- natural way of creating new objects is by building them out of existing objects.
- Complex systems are composed out of simpler sub-systems



Encapsulation


```
class Boeing747{
private:
  Gear gear;
public:
  void gearUp() {
    // update physics
    …
    gear.up();
  }
  void gearDown() {
    // update physics
    …
    gear.down();
  }
};
```

hidden (private section)

visible (public section)

- View objects as black box
- Don’t operate directly on internal data of an object
  - Implementation details are hidden behind interface
  - make member variables private
  - Use methods of the interface to perform certain actions
- Some languages, e.g. C++ and Java, help enforce this through specifiers: public, private, protected

Problems without Encapsulation

```
class Boeing747{
public:
  double totalAirFriction;
  Gear gear;
  void gearUp();
  void gearDown();
  double speed();
};

Boeing747 * a = new Boing747();
```

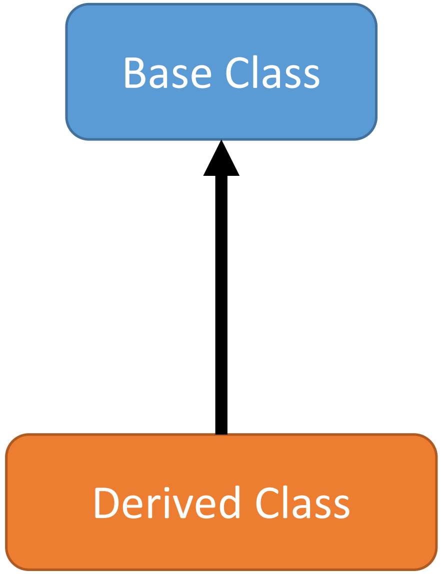
`a->gearDown();` ≠ `a->gear.down();`

Encapsulation ensures that side effects and domain knowledge are kept inside the class which is responsible for them.

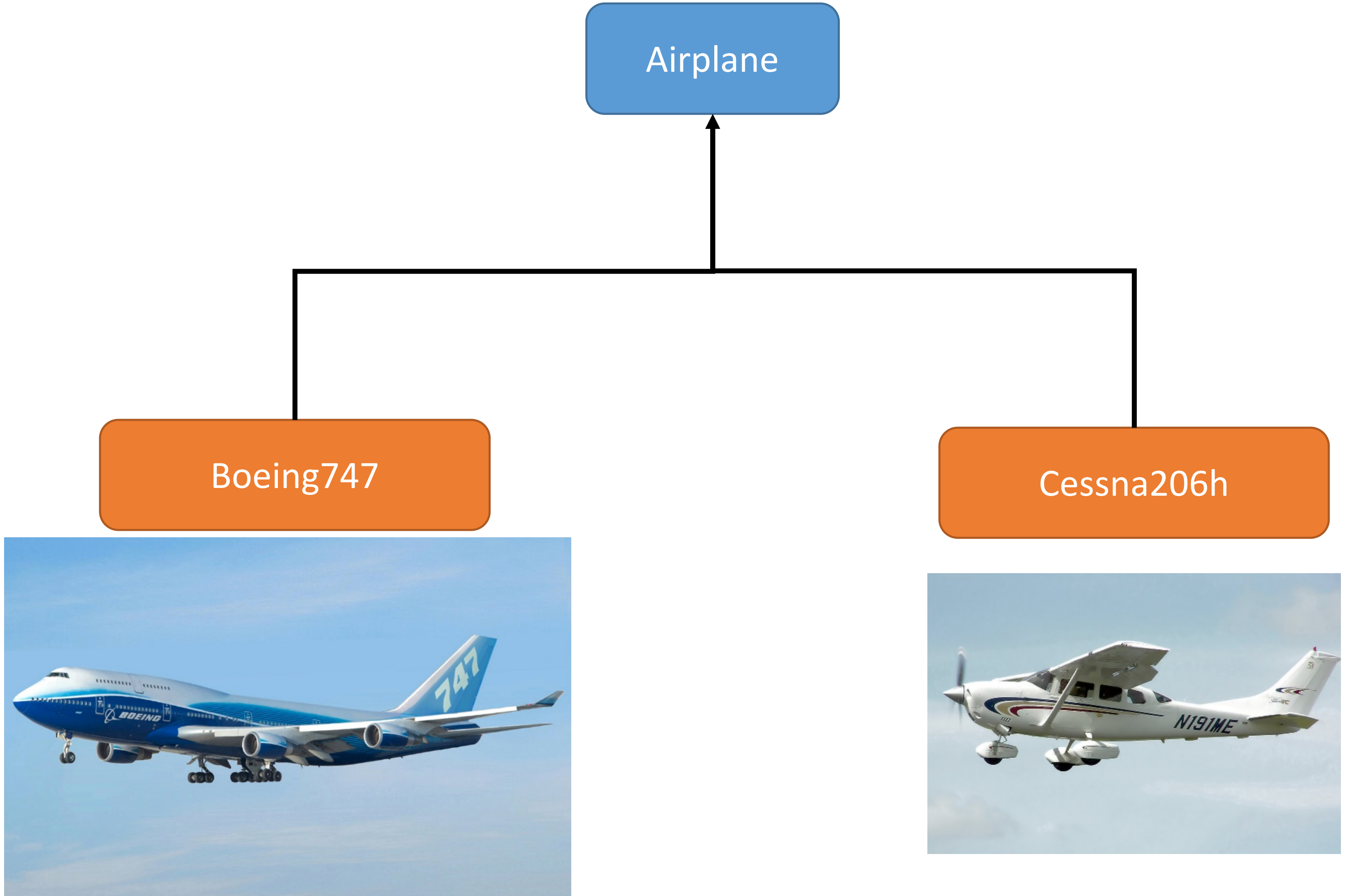
```
void Boeing747::gearDown() {
    if(gear.isUp()) {
        totalAirFriction += 20.0;
        gear.down();
    }
}
void Boeing747::gearUp() {
    if(gear.isDown()) {
        totalAirFriction -= 20.0;
        gear.down();
    }
}
void Boeing747::speed() {
    // function of thrust & friction
    return …
}
```

Type hierarchies and Inheritance

- Smalltalk first added the concept of **inheritance**
- Objects of a class can inherit state and behavior of a **base class** and make adjustments.
- Usages:
  - Extend classes with new functionality
  - Make minor modifications
  - Extract common functionality.



Type hierarchies and Inheritance



Type hierarchies and Inheritance

```
class Boeing747 {
    float longitude, latitude;
    float heading, speed;
    float altitude;
    ...
public:
    void fullThrottle() {
       engine1.maxThrottle();
       engine2.maxThrottle();
       engine3.maxThrottle();
       engine4.maxThrottle();
       speed = …;
    }
}
```

```
class Cessna206h {
    float longitude, latitude;
    float heading, speed;
    float altitude;
    ...
public:
    void fullThrottle() {
       engine1.maxThrottle();
       speed = …;
    }
}
```

Inheritance

```
class Airplane {
public:
    float longitude, latitude;
    float heading, speed;
    float altitude;

};
```

Common **structure**

```
class Boeing747 : Airplane {
...
public:
    void fullThrottle() {
       engine1.maxThrottle();
       engine2.maxThrottle();
       engine3.maxThrottle();
       engine4.maxThrottle();
       speed = …;
    }
}
```

```
class Cessna206h : Airplane {
...
public:
    void fullThrottle() {
       engine1.maxThrottle();
       speed = …;
    }
}
```

Derived classes **inherit structure** of Airplane

Inheritance

```
class Airplane {
public:
    float longitude, latitude;
    float heading, speed;
    float altitude;
    virtual void fullThrottle();
};
```

Common **interface**

```
class Boeing747 : Airplane {
...
public:
    void fullThrottle() {
        engine1.maxThrottle();
        engine2.maxThrottle();
        engine3.maxThrottle();
        engine4.maxThrottle();
        speed = …;
    }
}
```

```
class Cessna206h : Airplane {
...
public:
    void fullThrottle() {
        engine1.maxThrottle();
        speed = …;
    }
}
```

Derived classes **implement** or **extend** interface of Airplane

Polymorphism

```
Boeing747 * a = new Boeing747();
Airplane * b = new Boeing747();
Airplane * c = new Cessna206h();

// as expected: will call Boeing747::fullThrottle
a->fullThrottle();

// NEW! will call Boeing747::fullThrottle
b->fullThrottle();

// NEW! will call Cessna206h::fullThrottle
c->fullThrottle();
```

- Objects of derived classes can be used as base class objects.
- Polymorphism allows us to modify the behavior of base classes and replacing the implementation of methods.
- Polymorphic methods must be declared **virtual** (in C++)

Polymorphism

```
// store objects of different types in a
// datastructure using its base class
Airplane ** airplanes = new Airplane*[10];
...
for(int i = 0; i < 10; i++) {
   // will choose correct implementation
   // dynamically at runtime
   airplanes[i]->fullThrottle();
}


void foo(Airplane & a) {
   // this function will work with
   // any class derived from Airplane
}
```

- Use base class to implement general algorithms and data structures which work with any derived type
- Dynamic dispatch determines the type of an object at **runtime** and executes the correct method

Abstract classes

```
class Airplane {
public:
    float longitude, latitude;
    float heading, speed;
    float altitude;
    virtual void fullThrottle() = 0;
};
```

- Virtual functions without implementation are called **pure-virtual functions.**
- Classes containing pure-virtual functions can not be instantiated and are called ***abstract classes***
- Their behavior **must** be defined in derived classes

Is-A vs. Has-A Relationship

- Use composition to manage complexity
- Use inheritance to extract common functionality
- **Use polymorphism to implement general algorithms which are independent of specific types**
- **Composition = Has-A Relationship**
  E.g. A Boeing 747 has four jet engines.
- **Inheritance = Is-A Relationship**:
  E.g. A Cessna is an airplane. So is a Boeing 747.

III. Best Practices

Recommendations, Design Patterns

Keep your interfaces simple, clean and consistent

- Methods in a class should be as orthogonal as possible
  → avoid method that do almost the same
- Methods with the same name should do similar things
- Use encapsulation
  - This reduces coupling
  - Changing your implementation internals becomes easier
  - Access data with getter / setter methods
- Use **const** to limit what methods can do with your data
  - This lets the compiler help you enforce who can manipulate data
  - Compiler Optimization hint
- Consider using lazy-initialization for costly object properties

Getters & Setters

```
class SomeClass {
   int propertyA;
public:
   int getPropertyA() const {
      return propertyA;
   }

   void setPropertyA(int value) {
      // setters allow you to validate data
      // before accepting it
      if (value > 0 && value < 100) {
          propertyA = value;
      }
   }
}
```

const method does not modify data

Getter function

Setter function

Lazy Initialization

```
class SomeClass {
   LargeDataSet * dataSet;
public:
   SomeClass() : dataSet(nullptr)
   {
   }

   ~SomeClass() {
       delete dataSet;
   }

   int getDataSet() {
      if (!dataSet) {
          dataSet = new LargeDataSet();
      }
      return dataSet;
   }
}
```

Don’t create data set during construction

Won’t do anything if data set is never created

Create data set set on-demand

Base class destructors should be **virtual**

- If you want to allow deletion of objects by using their base class, make sure their destructor is virtual

```
class Base {
public:
   ~Base();
}

class Derived : public Base {
public:
   ~Derived();
}

Base * object = new Derived();
delete object; // undefined behavior!!!
// best case: only ~Base() is called
// potential memory leak
```

```
class Base2 {
public:
   virtual ~Base2(){}
}

class Derived2 : public Base2 {
public:
   virtual ~Derived2(){}
}

Base * object = new Derived();
delete object;
// first ~Derived() is called,
// then ~Base()
```

Don't use **virtual** functions during **construction** and **destruction**

- It's a bad idea. Even if you think you could use it, you will not get what you want.

dynamic dispatch is **disabled** during construction & destruction. These will **always** call the base class version!

```
class Base {
public:
   Base() {
     callVirtual();
   }
   virtual ~Base() {
     cout << "~Base()" << endl;
     callVirtual();
   }
   virtual void callVirtual() {
     cout << "Base::callVirtual" << endl;
   }
}
```

```
class Derived : public Base {
public:
   Derived() : Base() {}
   virtual ~Derived() {
     cout << "~Derived()" << endl;
   }
   virtual void callVirtual() {
     cout << "Derived ::callVirtual" << endl;
   }
}
```

Don’t reinvent the wheel

- Learn about available libraries
  - C++
    - STL → we‘ll have a look at it later this week
    - Boost
    - GUI toolkits if you need them (e.g. Qt)
  - Python
    - Modules
    - pip, setuptools
- OOP is all about writing reusable code, so use code that‘s already there and has been tested by other people
- Learn about design patterns

Design Patterns

- Term was made popular by the book “Design Patterns: Elements of Reusable Object-Oriented Software”, aka. **The Gang of Four** book (1994)
- Collection of generic solutions of common problems in OOP software
- Often used in libraries
- Part of the vocabulary computer scientists use to simplify communication

Design Patterns: Main categories

Creational Patterns	Structural Patterns	Behavioral Patterns
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template
		Visitor

Factory Methods

- Create objects without having to know specific language type
- Circumvent limitations of constructors
  - **No return result**
    only exceptions
  - **Constrained naming**
    e.g. can’t have two constructors with same parameter types
  - **Statically bound creation**
    there is no dynamic binding for constructors, you have to know which type you want to instantiate
  - **No virtual constructors**
- Factory methods can range from very simple implementations to complex selection schemes

Factory Methods – Simple Example

```
class IShape {
public:
    virtual void draw();
};

class Rectangle : IShape;
class Circle : IShape;
class Triangle : IShape;

class ShapeFactory {
public:
    IShape * createShape(const std::string & name);
}
```

```
IShape * createShape(const std::string & name)
{
    if (name == "rectangle") {
        return new Rectangle();
    }

    if (name == "circle") {
        return new Circle();
    }

    if (name == "triangle") {
        return new Triangle();
    }

    return nullptr;
}
```

```
// parse user input
ShapeFactory factory;
string selectedShape = getUserInput();

// create object at runtime
IShape * new_object = factory.createShape(selectedShape);
```

This factory implementation is hard coded. But you can easily write an **extensible factory**.

Adapter

- Used to make an object of one type compatible to another
- Typical use case:
  - You defined your own types of objects with a certain interface
  - You want to use an external library to manipulate your objects
  - However the interface expected by library is different to the one you used
- Instead of rewriting you code, you can create an Adapter class, which maps one interface to another.

Adapter – Example

```
class ForceComputation {
public:
    virtual void compute_force(Vector3D & force);
};
```

```
class LegacyClass {
public:
    virtual void compute_force(double * force);
};
```

```
class ForceComputationAdapter : public ForceComputation {
    LegacyClass * legacy;
public:
    ForceComputationAdapter(LegacyClass * src) : legacy(src) {
    }

    virtual void compute_force(Vector3D & force) {
       double f[3];
       legacy->compute_force(&f[0]);
       force.x = f[0];
       force.y = f[1];
       force.z = f[2];
    }
};
```

Strategy

- Used to keep parts of a larger implementations replacable
- You define a common interface to do a certain task
- Any class which implements that interface can be used in larger implementation
- Allows you to exchange object of that interface during runtime
- Typical use case:
  - Define a common interface to get data
  - Interface can be implemented by classes which use files, databases, web services, etc.

Strategy - Example

```
class IRandomNumberGenerator {
public:
    double getNextDouble() = 0;
}


class MyUncrackableEncryption {
    IRandomNumberGenerator * random;

    void setRandomNumberGenerator(IRandomNumberGenerator * r) {
        random = r;
    }

    void encrypt(char * data, size_t length) {
        double r = random->getNextDouble();
        ...
    }

    void decrypt(char * data, size_t length) {
        ...
    }
}
```

```
class DiceRoll : public IRandomNumberGenerator {
public:
    double getNextDouble() {
        // guaranteed to be random,
        // determined with a fair dice roll
        return 4;
    }
}
```

```
MyUncrackableEncryption e;
DiceRoll d;

e.setRandomNumberGenerator(d);
e.encrypt(…)
```

IV. Conclusion

Pro and Cons of Object-Orientated Programming

Benefits of OOP

- OOP encourages modularity and consistency
- Side effects from changing data are controlled
- Separate interface and implementation
- Control visibility and read/write access to data, violations can be found be the compiler
- Top level code becomes terse (-> less errors)
- Natural semantics for stateful items
- More compile time checking of correct use

Problems of OOP

- Designing good class hierarchies is hard and takes experience
- Bad design is easy
  - Objects get bloated by unneeded members
  - Inconsistent implementations (methods that have the same name don't do the same thing)
- Overhead of dynamic dispatch
- Inefficient data access for caching, vectorization
- Flow of control scattered across classes, especially with very deep class hierarchies
- Implicit actions (copy constructor, assignment operator) can become very expensive

Final Recommendations

- Use OOP in moderation
  - use OOP where it helps modularity
  - but not everything that can be an object needs to be
- At the upper level(s) imperative programming (using collections of objects) is often cleaner
- Use abstraction where details need not to be known, but do not hide what is important
- Object oriented programming is not bound to a specific programming language; some require less code to be written; the important part is sticking to the established conventions