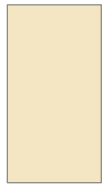
Polymorphism

Summary

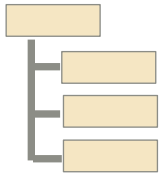
- You will know a lot about polymorphism and virtual methods
- You will recognize dynamic binding from static method matching
- You will know to apply the above

A Good Software Design

- *A single large procedure?*
  - 1000 Lines of code?



Breaking a complex problem/system into parts easier to conceive, understand, program, and maintain.

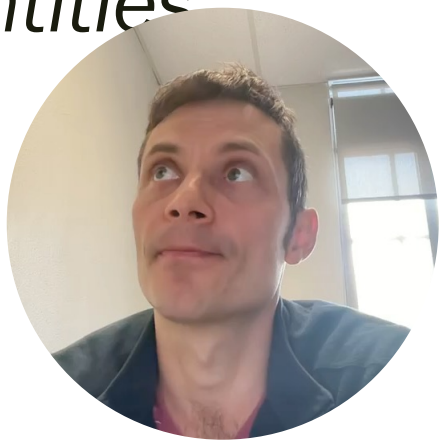


- Structural decomposition
- Separation of concerns
  - *Design principle*
    - Separating a computer program into distinct sections
    - Each section addresses a separate concern
  - *A **concern** is a set of information affecting computer program*
    - Performance, Security, Logging, Transactional behavior, etc.
- What if we cannot effectively separate certain concerns?
  - *Results in tangled/spaghetti code – or a model!*

Decomposition

- Structured programming
  - *breaks a process down into well-defined steps*

- Structured analysis
  - *breaks down a software system from the system context level to system functions and data entities*



Types of decomposition

- **Functional decomposition**
  - technique for mastering the complexity of the function of a model
- **Abstract Data type**
- **Object-oriented decomposition**

Object-oriented decomposition

- Breaks a large system down into progressively smaller **classes or objects** that are responsible for some part of the problem domain.

- Java EE
  - *Component decomposition*
    - Object with a specific meaning (~component)

Class abstraction and encapsulation

- Class abstraction separates its intent from the implementation
  - *from the use of the class*

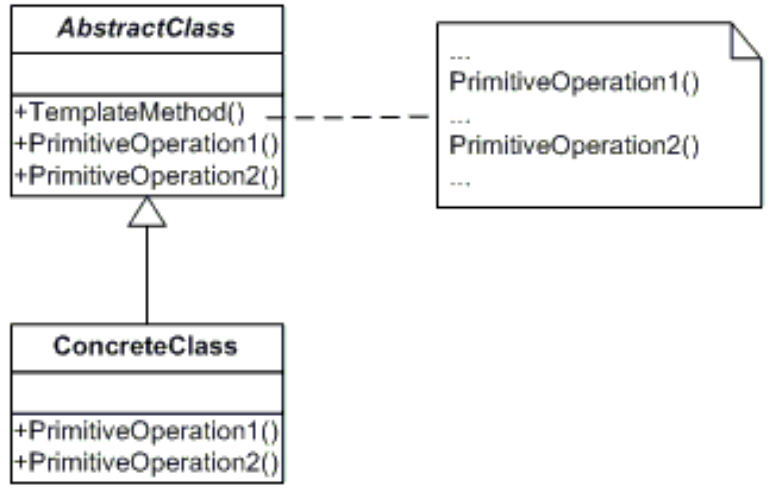
- Class description is provided in the abstraction and the developer knows how to use it
- The developer does not need to know how is the particular class implemented
- Implementation details are encapsulated and intentionally hidden from the developer
  - *private methods or variables*

  - *Consider*
    - List<Person> list = new ArrayList<>();

Class composition and inheritance

- Two ways to define one class in terms of the other
  - *Composition: one class composed of other classes*
  - *Inheritance: one class is a subclass of another class*

Polymorphism



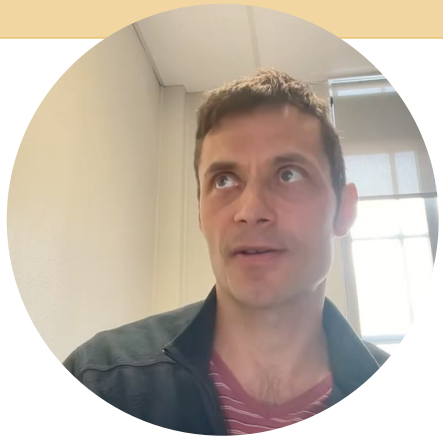
- Polymorphism means that:
  - ***a variable of a supertype can refer to a subtype object.***
  - *See image above: when we call templateMethod(), what gets called?*

- A class defines the type.
- A type defined by a subclass is called a subtype.
- A type defined by its superclass is called a supertype.
  - *Doberman is a subtype of German Dog that is a subtype of a Dog.*
  - *Both German Dog and Dog are supertypes of Doberman.*
  - *Dog can bark, can others do?*

Polymorphism

```
public class Dog {
        public void bark() {
                System.out.println("bark");
        }
}
public class GermanDog extends Dog {
        public void bark() {
                System.out.println("kläffen");
        }
}
public class Doberman extends GermanDog {
        public void bark() {
                System.out.println(
                        "does not bark, bites");
        }
}
```

```
public class Test {
     public static Dog init() {
             int rnd = (int)(Math.random()*3);
             if (rnd == 0) {
                     return new Dog();
             } else if (rnd == 1) {
                     return new GermanDog();
             } else {
                     return new Doberman();
             }
     }

  public static void provoke(Dog aDog) {
          aDog.bark();
  }


  public static void main(String[] args) {
         Dog aDog = init();
         provoke(aDog);
   }
}
```



Polymorphism

```
public class Dog {
	public void bark() {
		System.out.println(“bark”);
	}
}
public class GermanDog extends Dog {
	public void bark() {
		System.out.println(“kläffen”);
	}
}
public class Doberman extends GermanDog {
	public void bark() {
		System.out.println(“does not bark, bites”);
	}
}
```

Polymorphism

```
public static Dog init() {
  int rnd = (int)(Math.random()*2);
  if (rnd == 0) {
       return new GermanDog();
  } else {
       return new Doberman();
  }
}
```

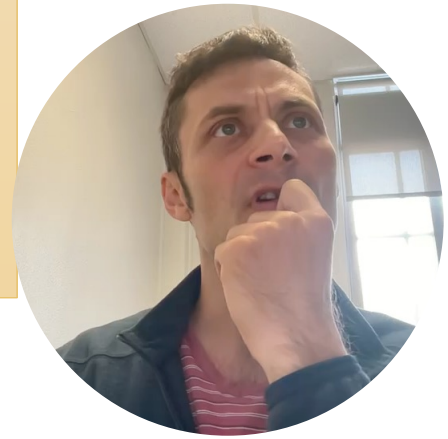
```
public abstract class Dog {
	public void bark() {
		System.out.println("bark");
	}
}
public class GermanDog extends Dog {
	public void bark() {
		System.out.println("kläffen");
	}
}
public class Doberman extends GermanDog {
	public void bark() {
		System.out.println("does not bark, bites");
	}
}
```

Polymorphism

```
public abstract class Dog {
	public abstract void bark();
}
public class GermanDog extends Dog {
	public void bark() {
		System.out.println(“kläffen”);
	}
}
public class Doberman extends GermanDog {
	public void bark() {
		System.out.println(“does not bark, bites”);
	}
}
```

Polymorphism

```
public interface class Dog {
	void bark();
}
public class GermanDog implements Dog {
	public void bark() {
		System.out.println(“kläffen”);
	}
}
public class Doberman extends GermanDog {
	public void bark() {
		System.out.println(“does not bark, bites”);
	}
}
```



Polymorphism

- Do all dogs have a color? How to enforce it for all subtypes?
- Even shi-tsu, or chihuahua

```
public abstract class Dog {
        public abstract void bark();
        public abstract String color();
}
```

- Java conventions: Action vs. Property (getter)

```
public abstract class Dog {
        public abstract void bark();
        public abstract String getColor();
}
```

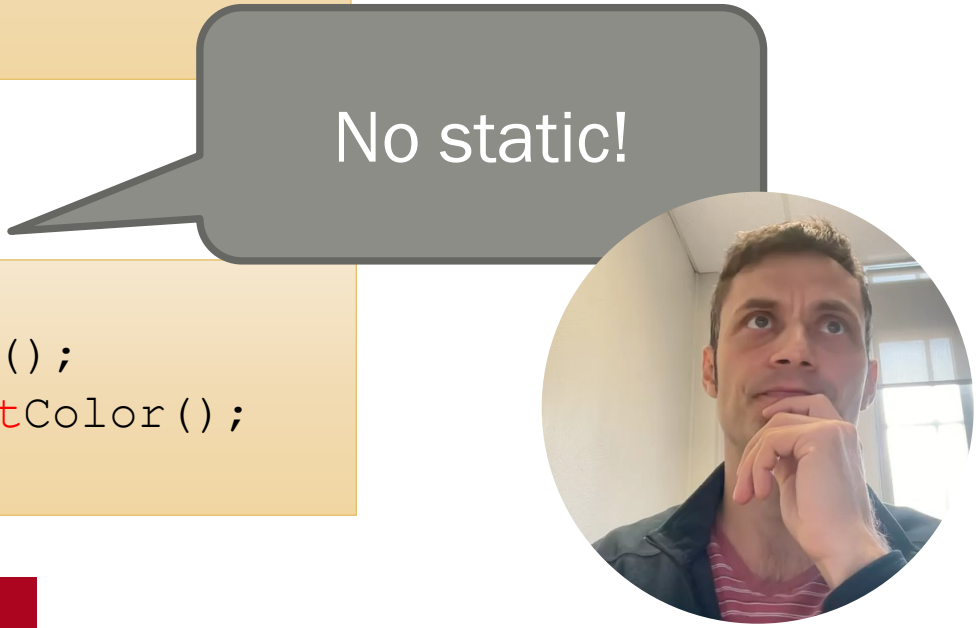
Polymorphism

- Do all dogs have a color? How to enforce it for all subtypes?
- Even shi-tsu, or chihuahua

```
public abstract class Dog {
        public abstract void bark();
        public abstract String color();
}
```

- Java conventions: Action vs. Property (getter)

```
public abstract class Dog {
        public abstract void bark();
        public abstract String getColor();
}
```



Polymorphism

```
public class Test {
    public void main(..) {
     ..
    aDog.heartBeat()
    // can I access and see heartBeat??
```

- Accessor?

Public? is your SSN public?

```
public abstract class Dog {
        public abstract void bark();
        public abstract String getColor();
        private void heartBeat(){..};
}
public class GermanDog extends Dog {
    public void bark() {..}
    public String getColor() {..}
    // can I access and see heartBeat??

}
```

Polymorphism

```
public class Test {
    public void main(..) {
     ..
    aDog.heartBeat()
    // can I access and see heartBeat??

}
```

- Accessor?

```
public abstract class Dog {
        public abstract void bark();
        public abstract String getColor();
        protected void heartBeat(){..};
}
public class GermanDog extends Dog {
    public void bark() {..}
    public String getColor() {..}
    // can I access and see heartBeat??
    // can I modify the method?
}
```

Polymorphism

```
public class Test {
    public void main(..) {
     ..
    aDog.heartBeat()
    // can I access and see heartBeat??

}
```

- Accessor?

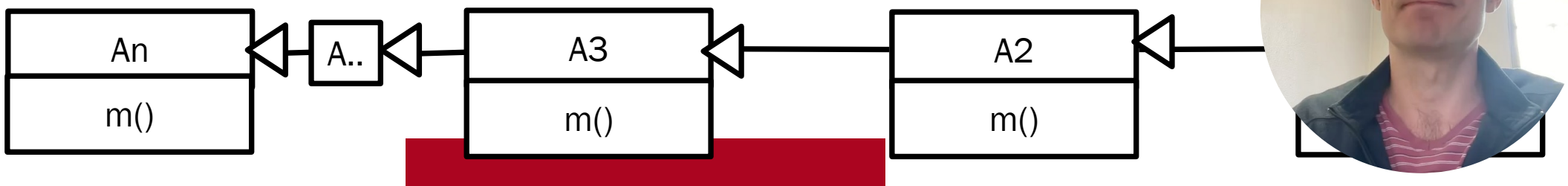
```
public abstract class Dog {
       public abstract void bark();
       public abstract String getColor();
       final protected void heartBeat(){..};
}
public class GermanDog extends Dog {
    public void bark() {..}
    public String getColor() {..}
    // can I access and see heartBeat??
    // can I modify the method?
}
```

Next, note the difference

- Polymorphism
  - *Dynamic binding*
- Method matching
  - *Nothing to do with polymorphism!*

How does the program solves polymorphism?

- Dynamic binding
  - `An a` *is an instance of A1.. but also {A2, A3, .. An}*
  - *A1 is a subclass of A2, A2 is a subclass of A3..*
  - *What is the most general class? An* *(In Java ~ Object)*
  - *What is the most specific? A1*
- When we invoke a method `m` on `a` ~ `a#m()`
  - *JVM searches implementation for* `m` *in {An,.. A3, A2, A1} in this order until found!*



Method matching vs binding

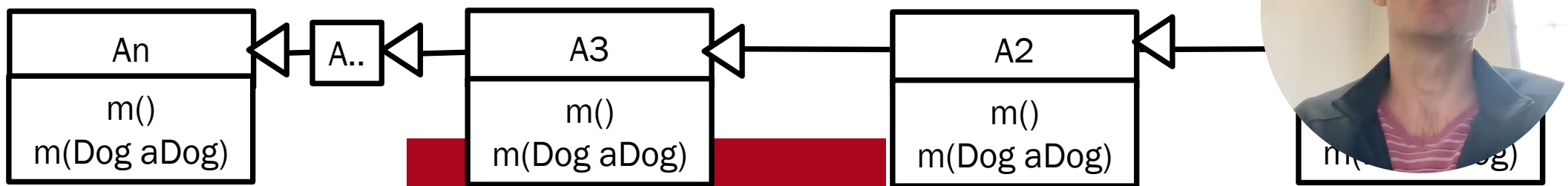
```
class An {
        public void m();
        public void m(int i);
        public void m(int i, int i);
        public void m(Dog);
}
```

- Two issues:
  - *Compiler finds a matching method according a parameter type, their count and order*
  - *All done at compile time!!!!!!*

A particular method can be implemented (override) in multiple subclasses

- *The method to execute found by dynamic binding*
- *JVM find the method implementation at runtime!!!!*
  - JVM finds the appropriate subclass

```
class An {
        public void m(Dog aDog);
} ..
class A3 extends A4  {
        public void m(D
} ..
class A2 exten
```



Generic programming

- Polymorphism allows methods to be used generally for polymorphic arguments

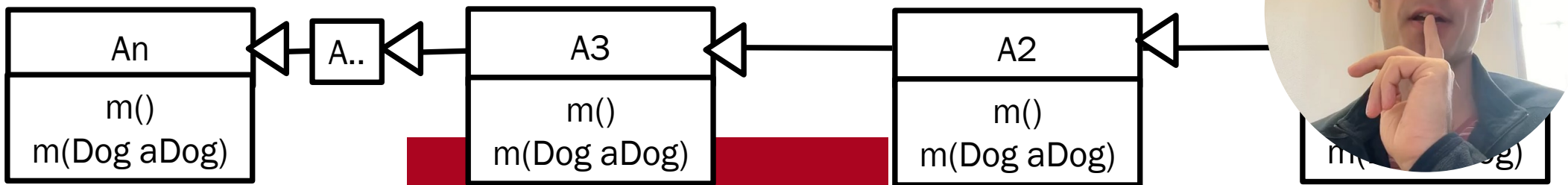
```
An a = initAnyAType();// any subtype of An
a.m(initDog());       // we do not know what concrete A and what Dog we have!
```

- and that is what is it about..
  - *If you don't think this is beautiful then change major!*

- If a method parameter is a supertype, you can pass any subclass

```
class An {
        public void m(Dog aDog);
} ..
class A3 extends A4  {
        public void m(Dog aDog);
} ..
class A2 extends A3
```

- The particular method implementation to invoke is determined dynamically

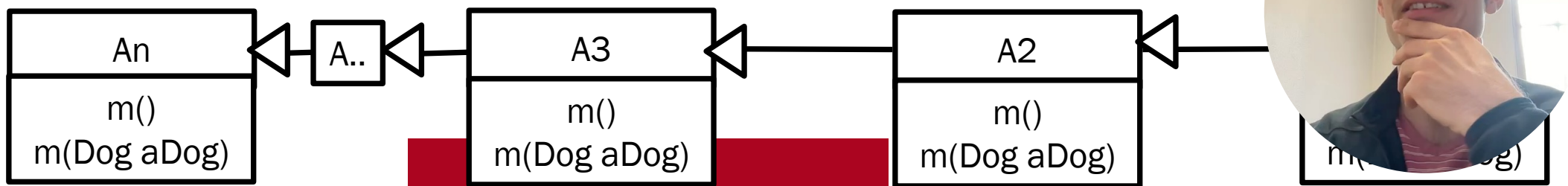


Generic programming

```
An a = initAnyAType();// any subtype of An
a.m(initDog());       // we do not know what concrete A and what Dog we have!
```

Can you notice the two dimensions for extensions?

```
class An {
        public void m(Dog aDog);
} ..
class A3 extends A4  {
        public void m(Dog aDog);
} ..
class A2 extends A3
```



Let me test you #1

Who barks?

```
public class Dog {
        public void bark(){..}|
        public String color(){..}

}
public class GermanDog extends Dog {
         public void bark(){..}
}
public class Doberman extends GermanDog {
         public void bark(){..}
}
```

```
public class Test {
 public static Dog init() {
    int rnd = (int)(Math.random()*3);
    if (rnd == 0) {
        return new Dog();
    } else if (rnd == 1) {
        return new GermanDog();
    } else {
        return new Doberman();
    }
  }
  public static void provoke(Dog aDog) {
    aDog.bark();
  }

  public static void main(String[] args) {
    Dog aDog = init();
    aDog.bark();
    provoke(aDog);
  }
}
```

Let me test you #2

Which method is called?

```
public class Dog {
        public void bark(){..}|
        public String color(){..}

}
public class GermanDog extends Dog {
         public void bark(){..}
}
public class Doberman extends GermanDog {
         public void bark(){..}
}
```

```
public class Test {
 public static Dog init() {
    int rnd = (int)(Math.random()*3);
    if (rnd == 0) {
        return new Dog();
    } else if (rnd == 1) {
        return new GermanDog();
    } else {
        return new Doberman();
    }
  }

..

  public static void identify(Dog aDog) {
    System.out.println("Dog")
  }
  public static void identify(GermanDog aDog) {
    System.out.println("GermanDog")
  }
  public static void identify(Doberman aDog) {
    System.out.println("Doberman ")
  }

  public static void main(String[] args)
    Dog aDog = init();
    aDog.bark();
    identify(aDog); // what happens now?
  }
}
```

Let me test you
#1 Dynamic binding

- dynamic,
- determined at run-time

```
public class Dog {
        public void bark(){..}|
        public String color(){..}

}
public class GermanDog extends Dog {
         public void bark(){..}
}
public class Doberman extends GermanDog {
         public void bark(){..}
}
```

```
public class Test {
 public static Dog init() {
    int rnd = (int)(Math.random()*3);
    if (rnd == 0) {
        return new Dog();
    } else if (rnd == 1) {
        return new GermanDog();
    } else {
        return new Doberman();
    }
  }


  ..


  public static void main(String[] args) {
    Dog aDog = init();
    aDog.bark();
    ..
  }
}
```

Let me test you
#2 Method matching

- Static! Compile time

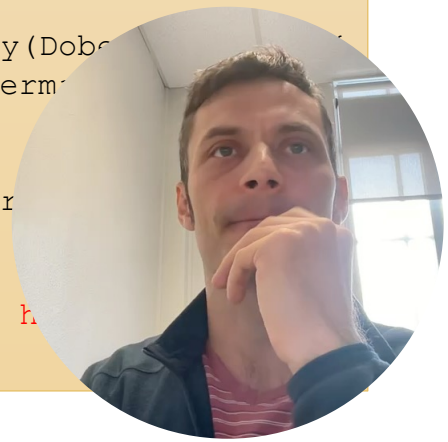
```
public class Dog {
        public void bark(){..}|
        public String color(){..}

}
public class GermanDog extends Dog {
         public void bark(){..}
}
public class Doberman extends GermanDog {
         public void bark(){..}
}
```

```
public class Test {
 public static Dog init() {
    int rnd = (int)(Math.random()*3);
    if (rnd == 0) {
        return new Dog();
    } else if (rnd == 1) {
        return new GermanDog();
    } else {
        return new Doberman();
    }
  }
 ..
  public static void identify(Dog aDog) {
    System.out.println("Dog")
  }
  public static void identify(GermanDog aDog) {
    System.out.println("GermanDog")
  }
  public static void identify(Doberman aDog) {
    System.out.println("Doberman ")
  }

  public static void main(String[] args) {
    Dog aDog = init();
    aDog.bark();
    identify(aDog); // what happens now?
  }
}
```

This is what you need to understand!

- Can we still make this happen to work as we want??
  - *Yes*
    - What options we have?

```
public class Test {
 public static Dog init() {
     int rnd = (int)(Math.random()*3);
     if (rnd == 0) {
          return new Dog();
     } else if (rnd == 1) {
          return new GermanDog();
     } else {
          return new Doberman();
     }
  }
  ..
  public static void identify(Dog aDog) {
     System.out.println("Dog")
  }
  public static void identify(GermanDog aDog)
{
     System.out.println("GermanDog")
  }
  public static void identify(Dob
     System.out.println("Doberm
  }

  public static void main(Str
     Dog aDog = init();
     aDog.bark();
     identify(aDog); // what h
  }
}
```



To pass this course this is what you need to understand!

- Can I still make this happen?
  - *Yes*
    - What options we have?
    - #1 move identify to the dog classes
    - But we lose centralization!!!!!!!!

```
class Dog {
        public void identify(){};
}
 class Germ.. extends Dog {
         public void identify(){};
} ..
class Dob extends Germ {
```

aDog.identify()

```
public class Test {
 public static Dog init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
            return new Dog();
      } else if (rnd == 1) {
            return new GermanDog();
      } else {
            return new Doberman();
      }
  }

   ..

  public static void identify(Dog aDog) {
      System.out.println("Dog")
  }
  public static void identify(GermanDog aDog) {
      System.out.println("GermanDog")
  }
  public static void identify(Doberman aDog)
      System.out.println("Doberman ")
  }

  public static void main(String[] args) {
      Dog aDog = init();
      aDog.bark();
    identify(aDog); // what happens now?
  }
}
```

To pass this course this is what you need to understand!

- Can I still make this happen?
  - *Yes*
    - What options we have?
    - #1 make it dispatch to the dog classes
    - Lets trick it and dispatch twice

```
class Dog {
        public void dispatch(Test t){};
}
 class Germ.. extends Dog {
         public void dispatch(Test t){};
} ..
class Dob extends Germ {  ..
        public void dispatch(Test t){};
} ..
```

```
public class Test {
 public static Dog init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
            return new Dog();
      } else if (rnd == 1) {
            return new GermanDog();
      } else {
            return new Doberman();
      }
  }

  ..

  public void identify(Dog aDog) {
      System.out.println("Dog")
  }
  public void identify(GermanDog aDog) {
      System.out.println("GermanDog")
  }
  public void identify(Doberman aDog) {
      System.out.println("Doberman ")
  }

  public static void main(String[] args) {
      Dog aDog = init();
      aDog.bark();
      aDog.dispatch(new Test());
  }
}
```

To pass this course this is what you need to understand!

- Can I still make this happen?
  - *Yes*
    - What options we have?
    - #1 make it dispatch to the dog classes
    - Lets trick it and dispatch twice

System.out.print(this); ???

```
class Dog {
        public void dispatch(Test t){
}
 class Germ.. extends Dog {
         public void dispatch(Test t){};
} ..
class Dob extends Germ {  ..
        public void dispatch(Test t){};
} ..
```

```
public class Test {
 public static Dog init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
                  new Dog();
                  nd == 1) {
                  new GermanDog();
                  return new Doberman();
      }
  }

  ..

  public void identify(Dog aDog) {
      System.out.println("Dog")
  }
  public void identify(GermanDog aDog) {
      System.out.println("GermanDog")
  }
  public void identify(Doberman aDog) {
      System.out.println("Doberman ")
  }

  public static void main(String[] args) {
      Dog aDog = init();
      aDog.bark();
      aDog.dispatch(new Test());
  }
}
```

To pass this course this is what you need to understand!

- Can I still make this happen?
  - *Yes*
    - What options we have?
    - #2 double dispatch
      - *combine what is polymorphic*
      - *with what is method match*

```
class Dog {
        public void dispatch(Test t){
                t.identify(this)
        };
}
 class GermanDog extends Dog {
         public void dispatch(Test t){
                t.identify(this)
        };
} ..
class Doberman extends Germ {  ..
```

```
public class Test {
 public static Dog init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
            return new Dog();
      } else if (rnd == 1) {
            return new GermanDog();
      } else {
            return new Doberman();
      }
  }

  ..

  public void identify(Dog aDog) {
      System.out.println("Dog")
  }
  public void identify(GermanDog aDog) {
      System.out.println("GermanDog")
  }
  public void identify(Doberman aDog) {
      System.out.println("Doberman ")
  }

  public static void main(String[] args) {
      Dog aDog = init();
      aDog.dispatch(new Test());
  }
}
```

To make it even better?

- Make interface for Test and call it a Visitor with the methods identify(...)

Now we know dynamic binding

- How is it implemented?

Back to C++
virtual method table

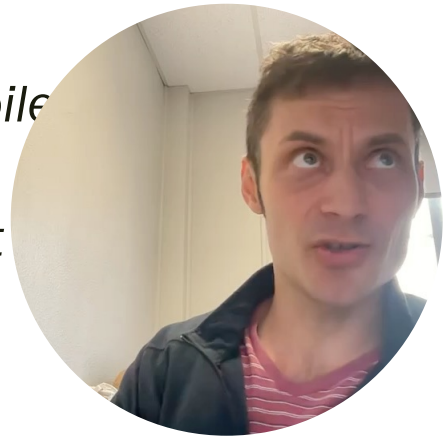
- A **virtual method table** (**VMT**)
  - AKA: **virtual function table**, **virtual call table**, dispatch table, **vtable**, or **vftable, etc.**
  - a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).

Back to C++
virtual method table

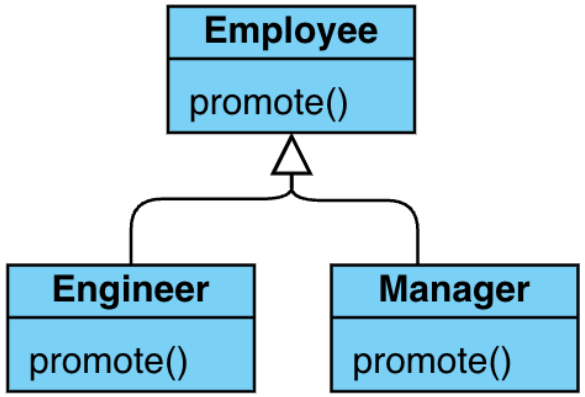
- Whenever a class defines a virtual method, most compilers add a hidden member variable to the class which points to an array of pointers to (virtual) methods called the virtual method table (VMT or Vtable).
  - These pointers are used at runtime to invoke the appropriate function implementations, because **at compile time it may not yet be known** if the base function is to be called or a derived one implemented by a class that inherits from the base class.
- There are many different ways to implement such dynamic dispatch, but the vtable (virtual table) solution is especially common among C++

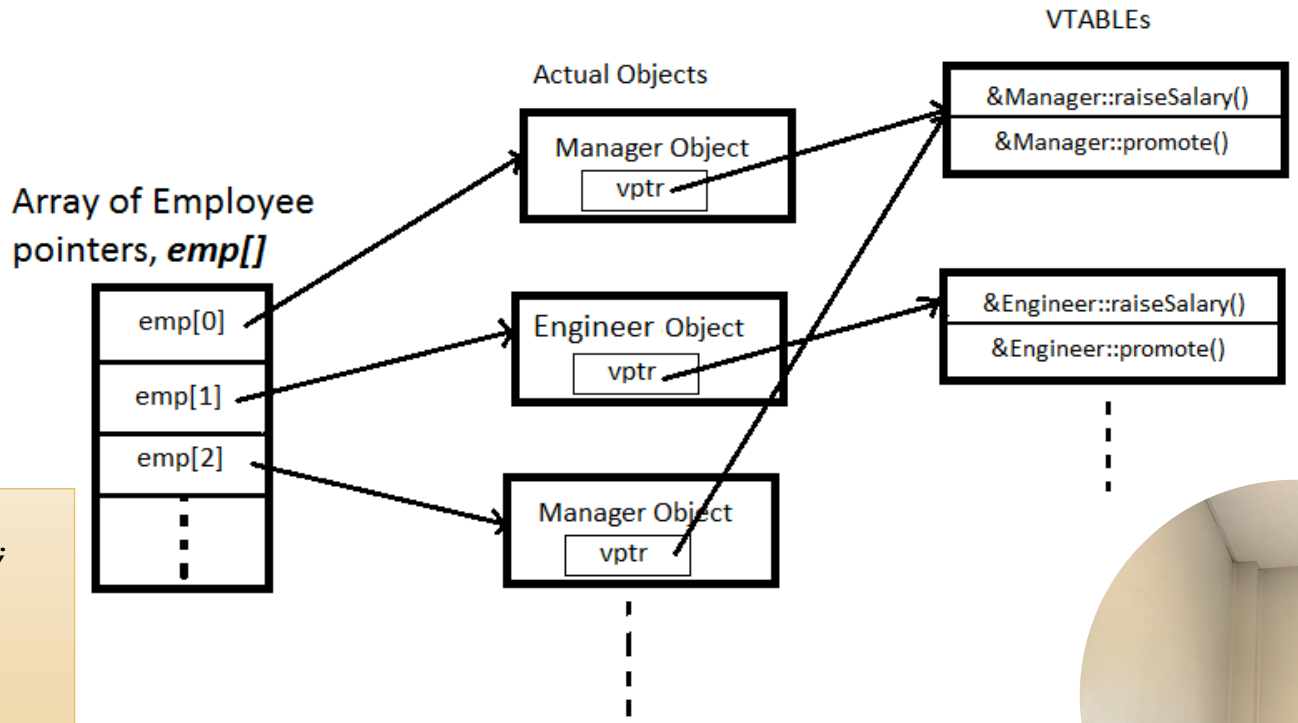
VMT

- Suppose a program contains several classes in an inheritance hierarchy: a superclass, Cat, and two subclasses, HouseCat and a Lion.
  - *Class Cat defines a virtual function named* `speak`*, so its subclasses may provide an appropriate implementation (e.g. either meow or roar or none!).*
- When the program calls the `speak` function on a Cat reference (which can refer to an instance of Cat, or an instance of HouseCat or a Lion), **the runtime must determine** which implementation of the function the call should be ***dispatched* to**.
  - *This depends on the* ***actual*** *class of the object,*
    - *it is* ***declared*** *in the class Cat.*
  - *The particular class can not generally be determined statically (at compile the compiler cannot decide which function to call at that time.*
  - *The call must be dispatched to the right function dynamically (that is, at instead.*



VMT - Employee, Manager, Engineer
promote()






```
..
     List<Employee> list = new ArrayList<>();
     for(int i=0;i<10;i++) {
          list.add(initEmployee());
      }
..
     for(int i=0;i<10;i++) {
          list.get(i).promote();
      }
..
```

Common example - a collection of something polymorphic

- Collection of things to process

- We avoided IF/ELSE/SWITCH!
- Good use of polymorphism
- Do animals know about shelter details?

```
class Dog extend Animal {
        public void dispatch(Shelter shelter){
                shelter.putToCage(this);
        };
}
class Cat extend Animal {
        public void dispatch(Shelter shelter){
                shelter.putToCage(this);
        };
} ..
class Flea extend Animal { ..
    public void  dispatch(Shelter shelter){
```

```
public class Shelter {
 public static Animal init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
            return new Dog();
      } else if (rnd == 1) {
            return new Cat();
      } else {
            return new Flea();
      }
  }
  ..
  public void putToCage(Dog g) {}
  public void putToCage(Cat g) {}
  public void putToCage(Flea g) {}
  ..

  public static void main(String[] args) {
     List<Animal> list = new ArrayList<>();
     for(int i=0;i<10;i++) {
          list.add(init());
     }
      ..
      Shelter cageDivider = new Shelte
      for(Animal animal : list) {
          animal.dispatch(cageDivider
      }

  }
}
```

Extracting Visitor interface.. loosing coupling

```
public interface Visitor {

  ..
  public void visit(Dog g) {}
  public void visit(Cat g) {}
  public void visit(Flea g) {}
  ..

}
```

- Do animals know about shelter details?

```
class Dog extend Animal {
        public void dispatch(Visitor v){
                v.visit(this);
        };
}
class Cat extend Animal {
        public void dispatch(Visitor v){
                v.visit(this);
        };
} ..
class Flea extend Animal { ..
    public void  dispatch(Visitor v){
```

```
public class Shelter implements Visitor{
 public static Animal init() {
      int rnd = (int)(Math.random()*3);
      if (rnd == 0) {
            return new Dog();
      } else if (rnd == 1) {
            return new Cat();
      } else {
            return new Flea();
      }
  }
  ..
  public void visit(Dog g) {}
  public void visit(Cat g) {}
  public void visit(Flea g) {}
  ..

  public static void main(String[] args) {
     List<Animal> list = new ArrayList<>();
     for(int i=0;i<10;i++) {
          list.add(init());
      }
       ..
      Visitor v = new Shelter();
      for(Animal animal : list) {
          animal.dispatch(v)
      }

  }
}
```

Inheritance

- Methods are virtual!
- What is it?

- Question:
  - *Do we use Override or Overload?*
  - *Is it static or dynamic?*

Test

- Methods are virtual!
- What is it?

- Question:
  - *Do we use Override or Overload?*
  - *Is it static or dynamic?*

```
public abstract class A {
  public static void main(String[] args) {
     A a = new B();
     a.test(5);
  }

 public void test (int p) {
     a();
  }
  public abstract void a();

  public void b() {
       out.print("A#b");
  }
}
```

```
public class B extends A {
 public void b() {
       out.print("B#b");
                  super.b();
                  a();
  }
}
```

Evaluation

```
public abstract class A {
  public static void main(String[] args) {
     A a = new B();
     a.test(5);
  }

 public void test (int p) {
     a();
  }
  public abstract void a();

  public void b() {
       out.print(“A#b”);
  }
}
```

```
public class B extends A {
  public void a() {
       out.print(“B#a”);
       super.b();
  }

 public void b() {
       out.print(“B#b”);
       super.b();
       a();
  }
}
```

Evaluation

```
public abstract class A {
  public static void main(String[] args) {
     A a = new B();
     a.test(5);
  }

 public void test (int p) {
     a();
  }
  public abstract void a();

  public void b() {
       out.print("A#b");
  }
}
```

```
public class B extends A {

  public void test (int p) {
     a();
     if (p > 1) {
          test(p-1);
     }
  }
  public void a() {
       out.print("B#a");
       super.b();
  }

 public void b() {
       out.print("B#b");
       super.b();
       a();
  }
}
```

Evaluation

```
public abstract class A {
  public static void main(String[] args) {
     A a = new B();
     a.test(5);
  }

 public void test (int p) {
     a();
  }
  public abstract void a();

  public void b() {
       out.print("A#b");
       b();
  }
}
```

```
public class B extends A {
  public void a() {
       out.print("B#a");
       super.b();
  }

 public void b() {
       out.print("B#b");

  }
}
```

Evaluation

```
public abstract class A {
  public static void main(String[] args) {
     A a = new B();
     a.test(5);
  }

 public void test (int p) {
     a();
  }
  public abstract void a();

  public void b() {
       out.print("A#b");
       b();
  }
}
```

```
public class B extends A {
  public void a() {
       out.print("B#a");
       super.b();
  }

 public void b() {
       out.print("B#b");

  }
}
```

If you understand this, all the rest is easy!

The rest of entire course is mostly about this!

...??? .. ?