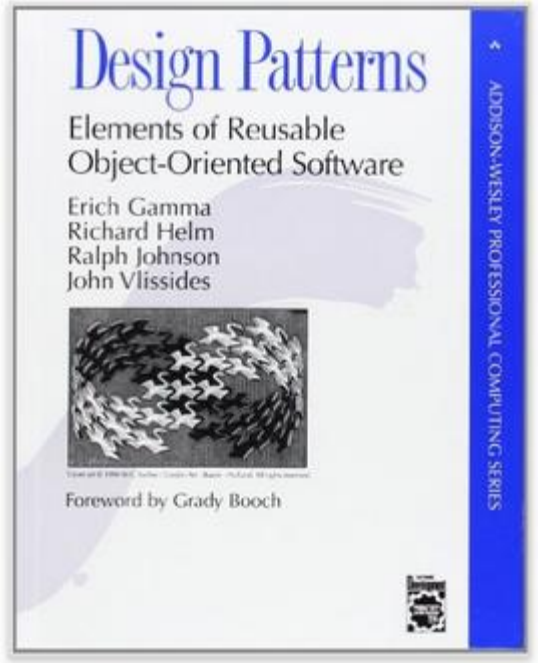
Interface

- Multiple Inheritance
- Abstract Classes
- The “interface”
- Default methods
- “Decorator” Pattern

See Also

https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

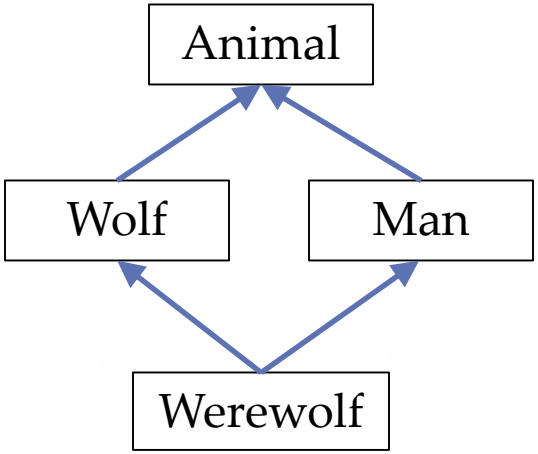
https://en.wikipedia.org/wiki/Design_Patterns



The Diamond of Death

```
class Animal {
    int numLegs;
    Animal(int legs) {
        numLegs = legs;
    }
}

class Wolf extends Animal {
    Wolf() {
        super(4);
    }
    public void sayHi() {
        System.out.println("Arrooo");
    }
}

class Man  extends Animal {
    Man() {
        super(2);
    }
    public void sayHi() {
        System.out.println("Hi");
    }
}
```

```
class Werewolf extends Wolf, Man {
    Werewolf() {
        super(); // Which? Both?
    }
}

Werewolf w = new Werewolf();
w.sayHi();
```



Purely Base Classes

- Some classes are designed as a base class never to be created themselves
- These classes don't represent concrete concepts

```
class Animal {
    public void sayHi() {
    }
    public void sayBye() {
    }
}

class Dog extends Animal {
    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }
}
```

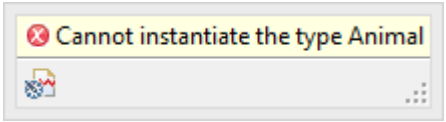
```
static void comeAndGo(Animal a) {
    a.sayHi();
    a.sayBye();
}


Dog d = new Dog();
comeAndGo(d);

Animal a = new Animal(); // What is this?
comeAndGo(a);
```

Abstract Classes

- Keyword "abstract" on a class means "you can't new one"
- The compiler will reject it

```
abstract class Animal {
    public void sayHi() {
    }
    public void sayBye() {
    }
}

class Dog extends Animal {
    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }
}
```

```
static void comeAndGo(Animal a) {
    a.sayHi();
    a.sayBye();
}


Dog d = new Dog();
comeAndGo(d);

Animal a = new Animal(); // What is this?
comeAndGo(a);
```

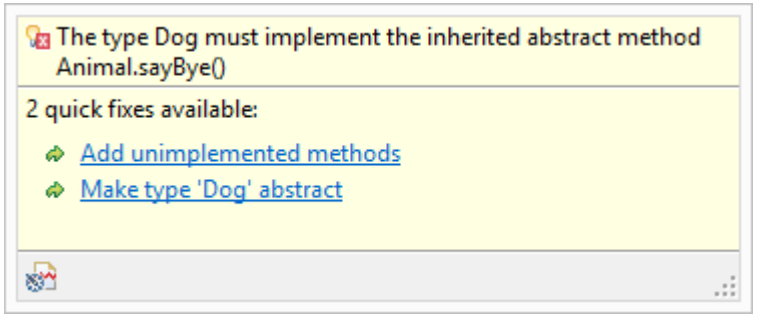


Abstract Methods

- Keyword "abstract" on a method means "no code here"
- Somebody down the inheritance tree must override

```
abstract class Animal {
    public abstract void sayHi();

    public abstract void sayBye();
}

class Dog extends Animal {
    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }
}
```

```
static void comeAndGo(Animal a) {
    a.sayHi();
    a.sayBye();
}
```



Abstractions

- Only classes and non-static methods can be abstract
- You can mix abstract and concrete
- If a class has any abstract methods then it must be explicitly abstract
- You can't "new" a class unless all the abstractions in all the bases are implemented

Diamond Solved?

```
abstract class Animal {
    int numLegs;
    Animal(int legs) {
        numLegs = legs;
    }
    public abstract void sayHi();
    public abstract void sayBye();
}

abstract class Machine {
   int powerRequired;
   Machine(int power) {
       powerRequired = power;
   }
   public abstract void sayHi();
   public abstract void sayBye();
}
```

```
class Robot extends Machine, Animal {
    Robot() {
        super(4); // Legs? Power?
    }
    public void sayHi() {
        System.out.println("WOOF");
    }
    public void sayBye() {
        System.out.println("PANT");
    }
}
```

```
Robot r = new Robot();
r.sayHi();
```

The "interface"

- Multiple "implementation" inheritance has problems
- Multiple "interface" inheritance is super useful
- Java has a keyword to separate the two

```
public interface Animal {
    public void sayHi();
    public void sayBye();
}

public interface Machine {
    public void sayHi();
    public void powerOn();
    public void powerOff();
}

class Robot implements Animal, Machine {

class Robot extends Object implements Animal, Machine {
```

- List the methods without implementations
- No data
- No constructors are added
- Use "implements" instead of "extends"
- Duplicate methods … no problem
- You can still "extend" one base class

Interfaces are Base Classes

- Interfaces behave just like Base Classes
- You can use a pointer to the interface just as you would an "extended" base class

```
Robot r = new Robot();

Animal a = r; // a.sayHi(), a.sayBye()
Machine m = r; // m.sayHi(), a.powerOn(), a.powerOff()

Robot q = (Robot)m;
```

Default Methods

```
public interface Animal {

    public void sayHi();

    public void sayBye();

    public void takeNap();

}
```

```
public class Dog implements Animal {

    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }

}
```

Default Methods

- Use "default" methods to silently extend an interface (new to Java 8)

```
public interface Animal {

    public void sayHi();

    public void sayBye();

    default public void takeNap() {
        System.out.println("ZZZzzz...");
    }

}
```

```
public class Dog implements Animal {

    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }

}
```

More Diamonds

```
public interface Animal {

    public void sayHi();

    public void sayBye();

    default public void takeNap() {
        System.out.println("ZZZzzz...");
    }
}
```

```
public interface Computer {

    public void sayHi();

    default public void takeNap() {
        System.out.println("101010 ...");
    }
}
```

```
public class Robot implements Animal, Computer {

    @Override
    public void sayHi() {
    }

    @Override
    public void sayBye() {
    }

    @Override
    public void takeNap() {
        Computer.super.takeNap();
    }

}
```

Duplicate default methods named takeNap with the parameters () and () are inherited from the types Computer and Animal

Needed for Lambdas

- Support for lambdas in Java 8
- Functional programming
- Single-method classes like for comparator functions and callbacks
- Enhanced interfaces throughout the library

Design Patterns

- The same design problems tend to arise over and over across applications
- A Design Pattern is a formula for solving a common problem
- When you use the name in conversations your peers will know the details

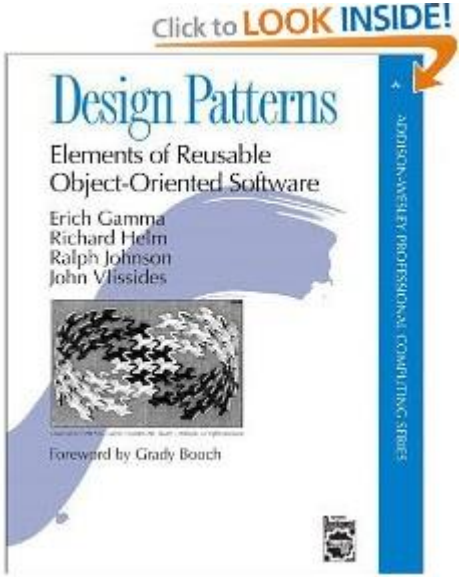
```
      34061
37 | 1260257
    111
     150
     148
       225
       222
         37
         37
          0
```

1. When dividing two numbers, for example, *n* divided by *m*, *n* is the dividend and *m* is the divisor; the answer is the quotient.
2. Find the location of all decimal points in the dividend and divisor.
3. If necess ... ividend by the same number of decimal ... of the last digit.
4. When do ... e tableau.
5. After eac ... e are three possible problems: the multiplication is wrong, the subtraction is wrong, or a greater quotient is needed.
6. In the end, the remainder, *r*, is added to the growing quotient as a fraction, *r/m*.

Long Division

The "Decorator"

- Design Patterns: Elements of Reusable Object-Oriented Software
- Great discussion of Object Oriented programming



The "Decorator"

- We want to decorate the dog's output to make it look like French
- "WOOF" becomes "LE WOOF"

```
interface Animal {
    public void sayHi();
    public void sayBye();
}

class Dog implements Animal {

    @Override
    public void sayHi() {
        System.out.println("WOOF");
    }

    @Override
    public void sayBye() {
        System.out.println("PANT");
    }

}
```

```
Dog dog = new Dog();
Animal a = dog;

Animal a = new Dog();

// This code just works with the
// Animal interface.
a.sayHi();
a.sayBye();
```

By Subclassing

- We could extend Dog into FrenchDog
- What about a FrenchCat? FrenchFrog?
- The "LE " is the same for all of them
- Usually better ways to solve problems than subclassing

```
class FrenchDog extends Dog {
    @Override
    public void sayHi() {
        System.out.print("LE ");
        super.sayHi();
    }
}
```

```
class FrenchCat extends Cat {
    @Override
    public void sayHi() {
        System.out.print("LE ");
        super.sayHi();
    }
}
```

```
class FrenchFrog extends Frog {
    @Override
    public void sayHi() {
        System.out.print("LE ");
        super.sayHi();
    }
}
```

By Composition

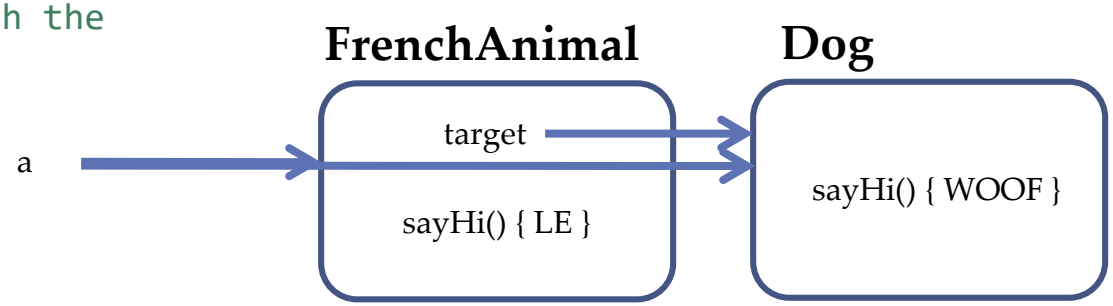
```
class FrenchAnimal implements Animal {

    Animal target;
    FrenchAnimal(Animal other) {
        target = other;
    }

    @Override
    public void sayHi() {
        System.out.print("LE ");
        target.sayHi();
    }

    @Override
    public void sayBye() {
        System.out.print("LE ");
        target.sayBye();
    }

}
```

- Create an object that has a pointer to the "real" object
- Pass the real object to the constructor
- This object becomes the "man in the middle"
- Delegates to the "real" object

Wiring up at Runtime

- Works with ALL Animals
- Can wire in lots of other decorators
- Subclassing is fixed at compile time
- This wiring is flexible … at runtime

```
Animal a = new Dog();
Animal a = new FrenchAnimal(new Dog());
Animal a = new FrenchAnimal(new FrenchAnimal(new Dog()));

// This code just works with the
// Animal interface.
a.sayHi();
a.sayBye();
```



Your Turn

- Code up the Animal interface along with Dog and Cat
- Write a static method to call the methods on Animal
- Create Dogs and Cats and pass them to the static method
- Create SickDog that extends Dog and moans with each method
- Make a SickAnimal decorator instead.
- Test it on the Cat.