Inheritance

- Inheritance:

```
class B { ... }
class A extends B { ... }
```

- A is a *subclass* of B, or equivalently, A is *derived from* B, A is a *child of* B, or B is a *superclass of* A, or B is a *parent of* A.

- Means that the set of A objects is a subset of the set of B objects.

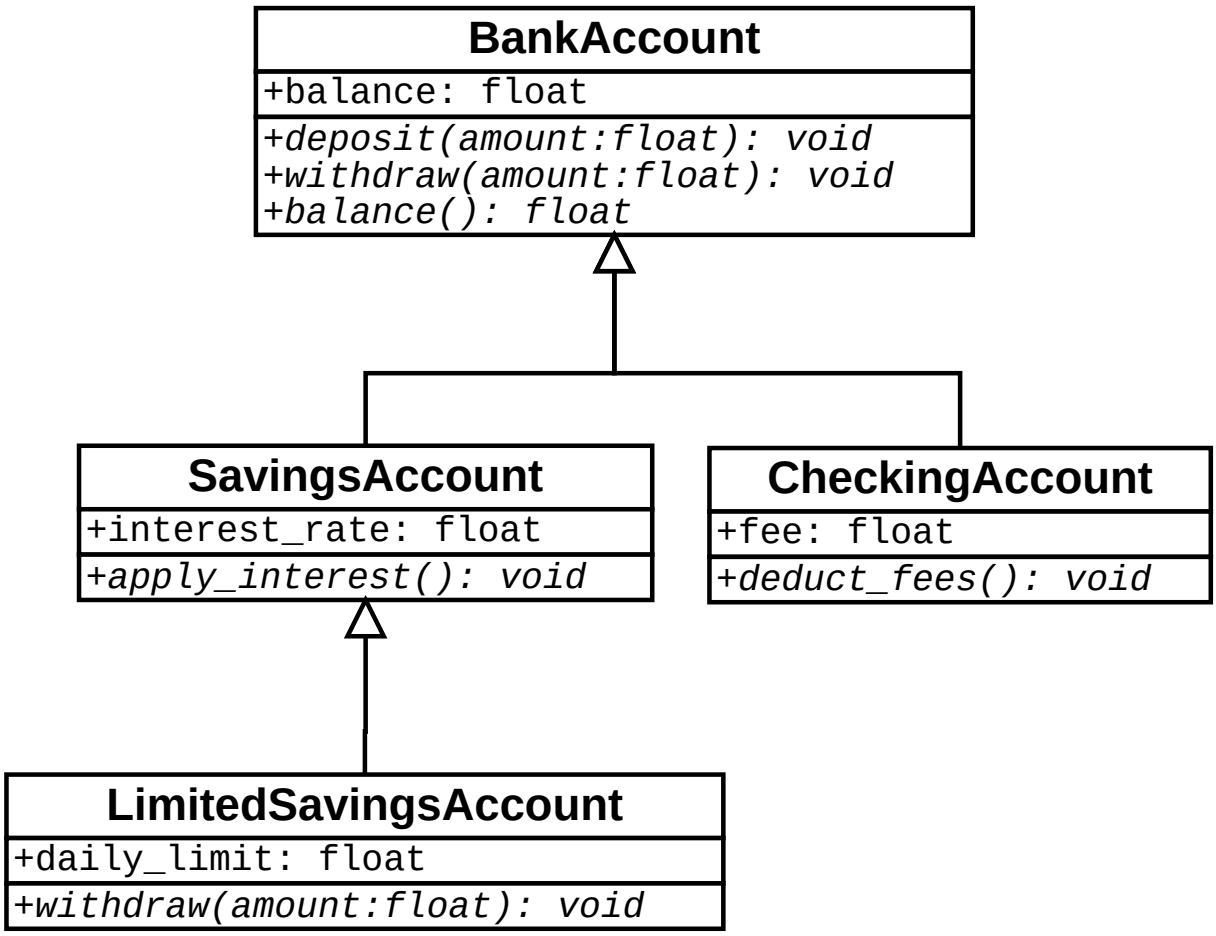
```
class Labrador extends Dog { ... }
```

- Inheritance represents the "is-a" relationship

Inheritance

- Roles of inheritance:
  - Represents the *is-a* relationship between classes
  - Describes *specialization*
  - Describes *"being a subset of"*
  - Describes *alternatives* (an Animal is a Dog *or* a Cat *or* a Bird, etc.)
  - A parent class describes the things that all its subclasses have in *common*
  - Allows us to *reuse* definitions by extending classes

Inheritance



Inheritance

```
class SavingsAccount extends BankAccount {
  private float interest_rate;
  public SavingsAccount(float initial_balance,
                        float rate)
  {
    super(initial_balance); // Calls superclass
                            // constructor
    interest_rate = rate;
  }
  public void apply_interest()
  {
    balance = balance
            + balance * interest_rate/100.0;
  }
}
```

Inheritance

```
class CheckingAccount extends BankAccount {
  private float fee;
  public CheckingAccount(float initial_balance,
                         float fee)
  {
    super(initial_balance);
    this.fee = fee;
  }
  public void deduct_fee()
  {
    balance = balance - fee;
  }
}
```

Overriding methods

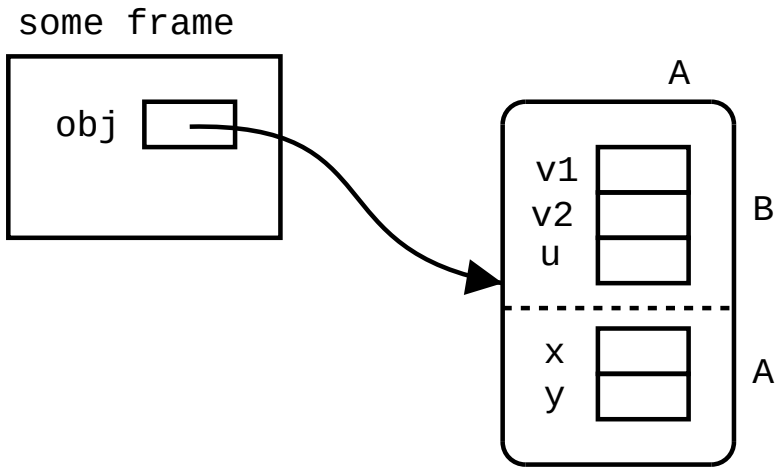
```
class LimitedSavingsAccount
extends SavingsAccount {
  private float daily_limit;
  public LimitedAccount(float initial_balance,
                        float rate, float limit)
  {
    super(initial_balance, rate);
    daily_limit = limit;
  }
  public void withdraw(float amount)
  {
    if (amount < daily_limit)
      balance = balance - amount;
  }
}
```

Inheritance

```
class B {
  int v1, v2;
  String u;
  void m() { ... }
}
class A extends B {
  double x;
  boolean y;
  void p() { ... }
  void s() { ... }
}
```

Inheritance

```
// In some client
A obj = new A();
obj.p();
obj.m();
// We can refer to ... obj.x ... obj.y ...
// ... obj.u ... obj.v1 ... obj.v2 ...
```



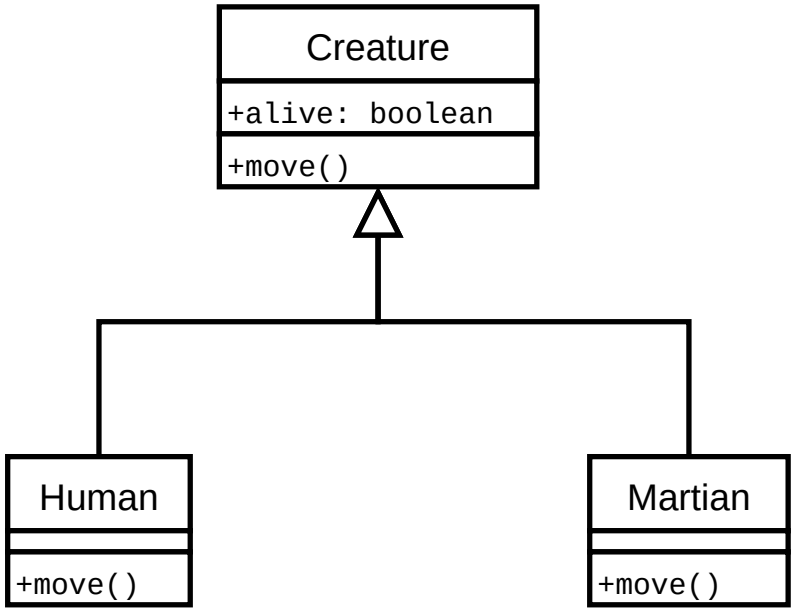
Object Oriented Programming

- The execution of an OO program consists of
  - Creation of objects
  - Interaction between objects (message-passing)
- Defining features of an OO language:
  - Class definitions (describing the types of objects and their structure,)
  - Object instantiation (creation,)
  - Message-passing (invoking methods,)
  - Aggregation (object structure, *has-a* relationships)
  - Encapsulation (objects as abstract units, hiding,)
  - Inheritance,
  - Polymorphism

Polymorphism

- Polymorphism means "many forms."
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- If a class A has a method m we could give different meaning to m by defining subclasses that override m, and therefore the result of executing m depends on the context, since the context decides which subclass is instantiated.

Polymorphism



Polymorphism

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
}
class Human extends Creature {
  void move()
  {
    System.out.println("Walking...");
  }
}
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Polymorphism

```
public class ZooTest
{
  public static void main(String[] args)
  {
    Human yannick = new Human();
    Martian ernesto = new Martian();
    ernesto.move();
    yannick.move();
  }
}
```

Polymorphism

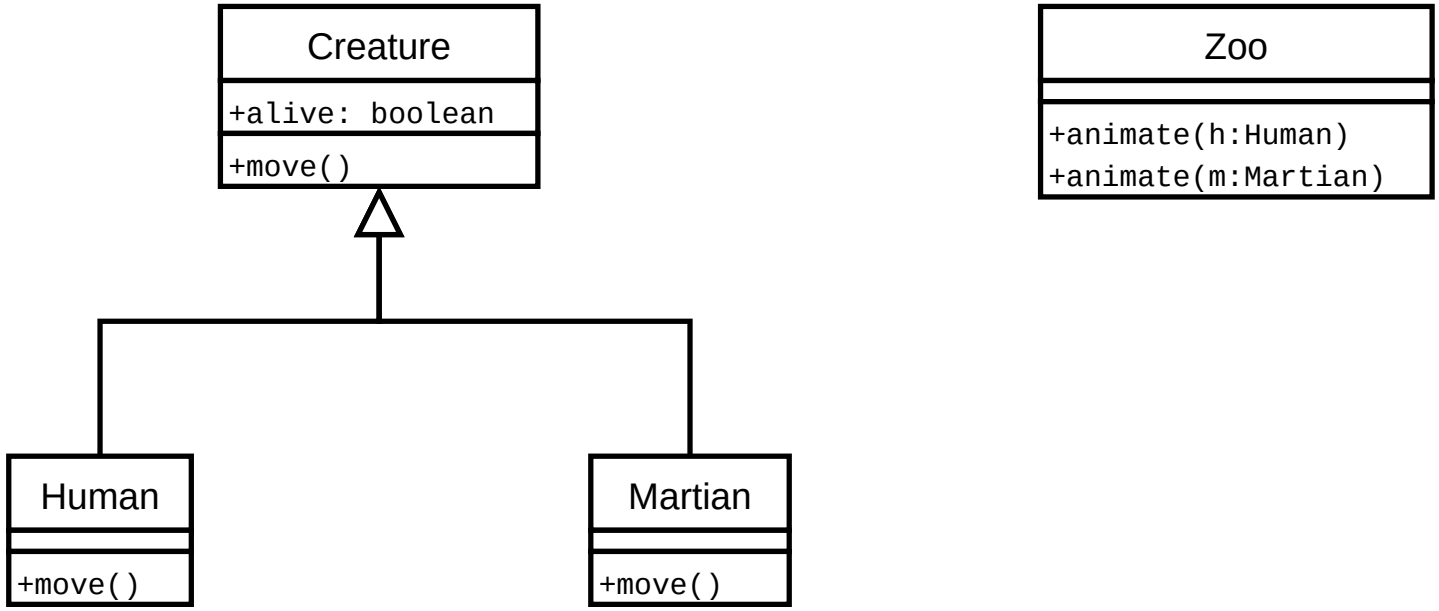
- A polymorphic method is a method which can accept more than one type of argument

- Kinds of polymorphism:
  - Overloading (Ad-hoc polymorphism): redefining a method in the same class, but with different signature (multiple methods with the same name.) Different code is required to handle each type of input parameter.
  - Parametric polymorphism: a method is defined once, but when invoked, it can receive as arguments objects from any subclass of its parameters. The same code can handle different types of input parameters.

Polymorphism

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
}
class Human extends Creature {
  void move()
  {
    System.out.println("Walking...");
  }
}
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Ad-hoc Polymorphism (Overloading)

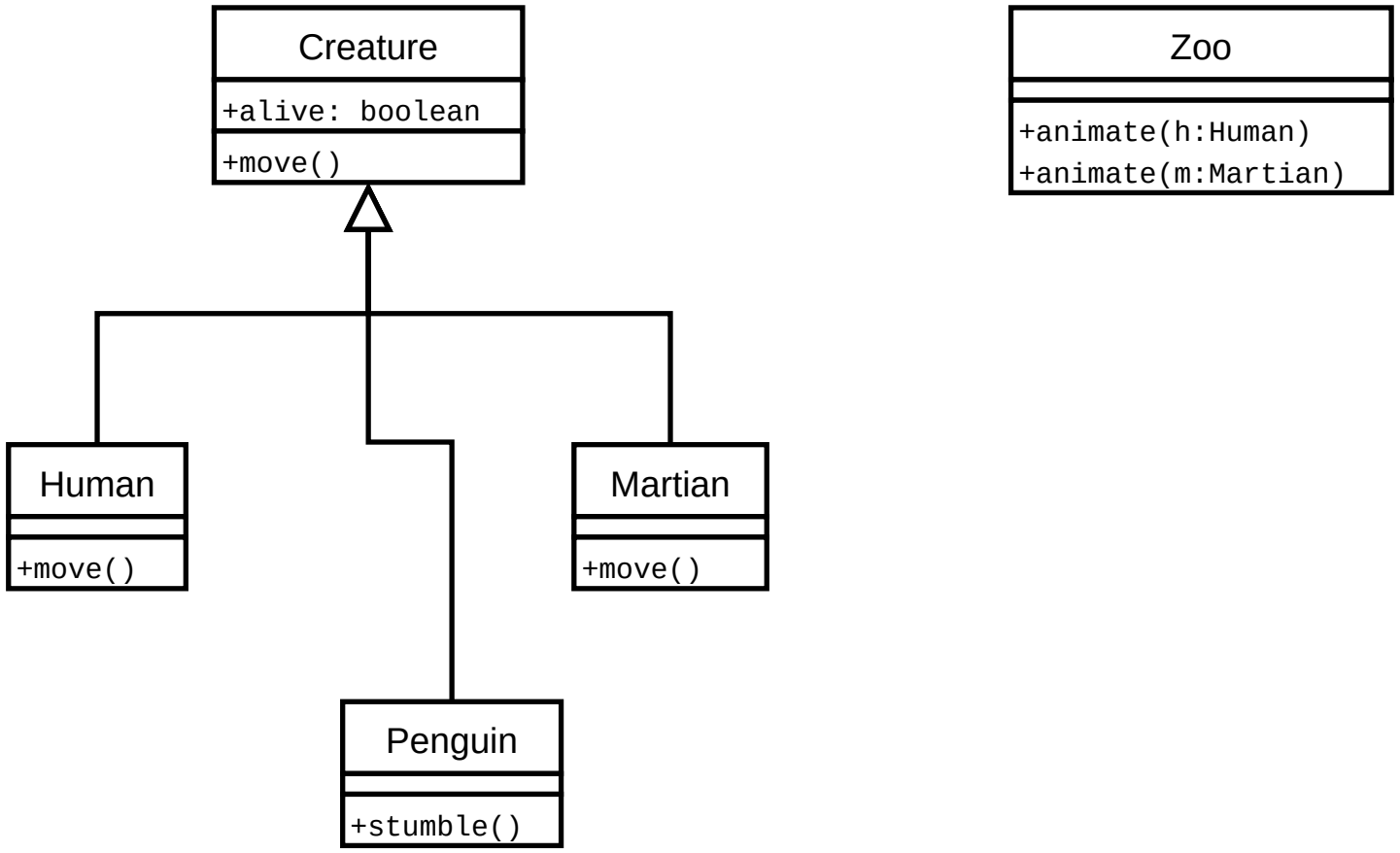


Ad-hoc Polymorphism (Overloading)

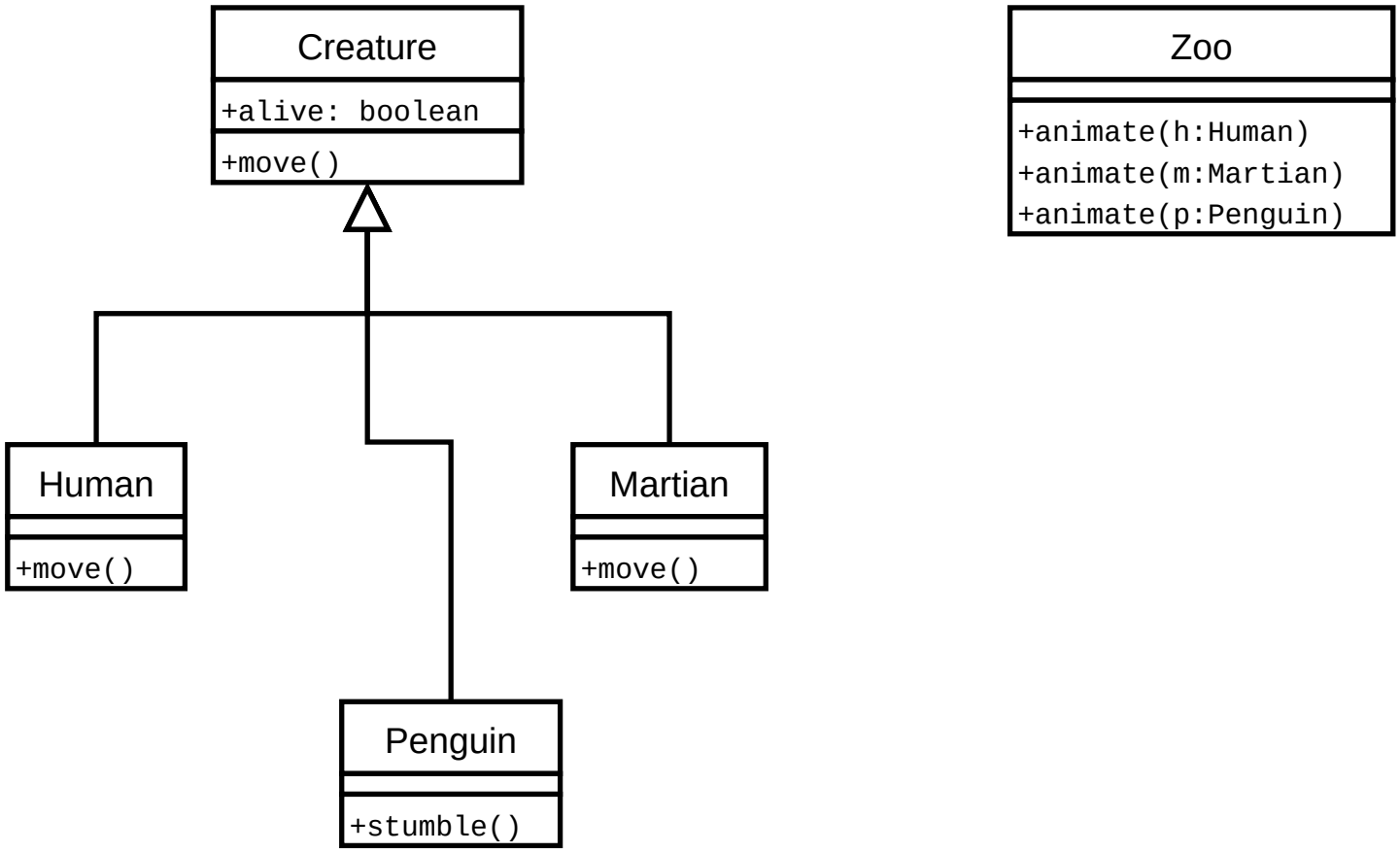
```
class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Ad-hoc Polymorphism (Overloading)



Ad-hoc Polymorphism (Overloading)

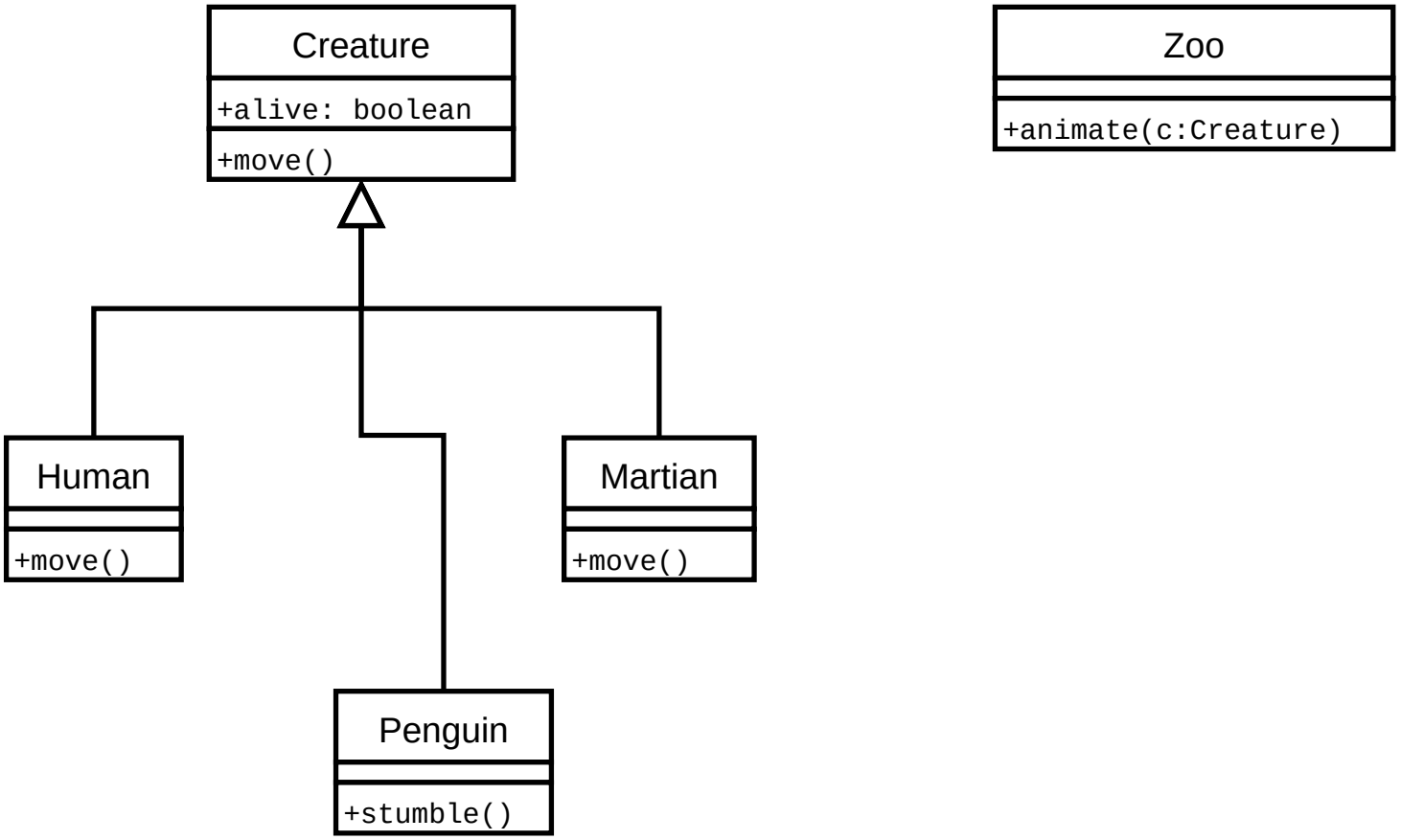


Ad-hoc Polymorphism (Overloading)

```
class Penguin extends Creature {
  void stumble()
  {
    System.out.println(“Ouch”);
  }
}

class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
  void animate(Penguin p)
  {
    p.move();
  }
}
```

Parametric Polymorphism



Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move(); // Dynamic-dispatch
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick); // Polymorphic call
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move(); // Dynamic-dispatch
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    Penguin paco = new Penguin();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick); // Polymorphic call
    my_zoo.animate(paco);    // Polymorphic call
  }
}
```

Polymorphism

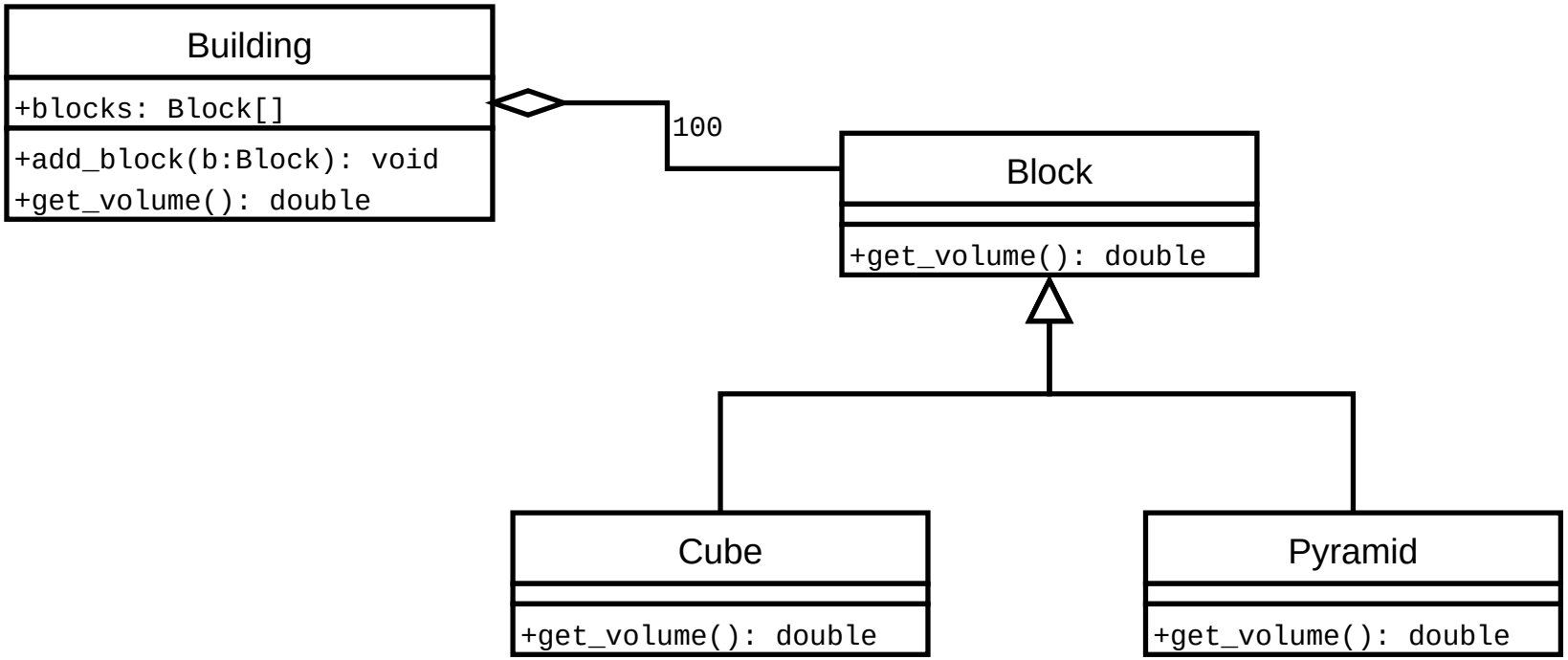
- Polymorphism is a tool that permits abstraction and reusability
- A polymorphic method is a method which can receive as input any object whose class is a subclass of the methods's parameter.
- Ad-hoc polymorphism is overloading (providing separate methods for each expected parameter type)
- Parametric polymorphism relies on dynamic-dispatching. Dynamic-dispatching is the process by which the runtime system directs the message of an object to the appropriate subclass.
- A dynamic-dispatch can be decided only at run-time, not at compile-time, because the compiler cannot know which is the actual object passed as argument to a polymorphic method. Furthermore, the same method might be called with different objects from different classes during the execution of the program.

Example

Suppose that a *building* is made up of up to 100 *blocks*, where each block is either a *cube* or a *pyramid*.

- Each block, regardless of its type, has a *volume*, and it should be possible to obtain the volume of a given block.
- A cube is a block that has a side length l. The volume of a cube is l^3.
- A pyramid is a block that has a base side-length b, and a height h. The volume of a pyramid is $\frac{1}{3}b^2h$.
- A building has up to 100 blocks. We can *add* blocks to a building, and we can compute the total volume of the bulding. The total volume is the sum of the volumes of all the blocks in the building.

Design


Building
+blocks: Block[]
+add_block(b:Block): void
+get_volume(): double
100
Block
+get_volume(): double
Cube
+get_volume(): double
Pyramid
+get_volume(): double

Implementation

```
class Block
{
  protected double volume;
  public double get_volume() { return volume; }
}
```

Implementation

```
class Cube extends Block
{
  private double side_length;
  public Cube(double s)
  {
    side_length = s;
  }
  public double get_volume()
  {
    volume = side_length * side_length * side_lengt
    return volume;
  }
}
```

Implementation

```
class Pyramid extends Block
{
  private double base, height;
  public Pyramid(double b, double h)
  {
    base = b;
    height = h;
  }
  public double get_volume()
  {
    volume = (1.0 / 3) * base * base * height;
    return volume;
  }
}
```

Implementation

```
public class Building
{
  private Block[] blocks;
  private int first_available;
  public Building()
  {
    blocks = new Block[100];
    first_available = 0;
  }
  public void add_block(Block b)
  {
    if (first_available < 100) {
      blocks[first_available] = b;
      first_available++;
    }
  }

    // Continues...
```

```
    public double get_volume()
    {
      double volume = 0.0;
      for (int i=0; i<first_available; i++) {
        volume += blocks[i].get_volume();
      }
      return volume;
    }
  }
```

Implementation

```
public class Test {
  public static void main(String[] args)
  {
    Building b = new Building();
    Cube c1, c2;
    Pyramid p1;
    c1 = new Cube(3);
    c2 = new Cube(5);
    p1 = new Pyramid(4, 5);
    b.add_block(c1);
    b.add_block(c2);
    b.add_block(p1);
    double v = b.get_volume();
  }
}
```

Implementation

```
public class Building
{
  private Block[] blocks;
  private int first_available;
  private double volume;

  public Building()
  {
    blocks = new Block[100];
    first_available = 0;
    volume = 0;
  }
  public void add_block(Block b)
  {
    if (first_available < 100) {
      blocks[first_available] = b;
      first_available++;
      volume = volume + b.get_volume();
    }
  }
    // Continues ...
```

```
    public double get_volume()
    {
      return volume;
    }
  }
```

Implementation

```
class Block
{
  protected double volume;
  public double get_volume() { return volume; }
}
```

Implementation

```
class Cube extends Block
{
  private double side_length;
  public Cube(double s)
  {
    side_length = s;
  }
  public double get_volume()
  {
    volume = side_length * side_length * side_lengt
    return volume;
  }
}
```

Implementation

```
class Cube extends Block
{
  private double side_length;
  public Cube(double s)
  {
    side_length = s;
    volume = side_length * side_length * side_lengt
  }
}
```

Implementation

```
class Pyramid extends Block
{
  private double base, height;
  public Pyramid(double b, double h)
  {
    base = b;
    height = h;
  }
  public double get_volume()
  {
    volume = (1.0 / 3) * base * base * height;
    return volume;
  }
}
```

Implementation

```
class Pyramid extends Block
{
  private double base, height;
  public Pyramid(double b, double h)
  {
    base = b;
    height = h;
    volume = (1.0 / 3) * base * base * height;
  }
}
```

Implementation

```
class Cylinder extends Block
{
  private double radius, height;
  public Cylinder(double r, double h)
  {
    radius = r;
    height = h;
    volume = Math.PI * radius * radius * height;
  }
}
```

Implementation

```
public class Test {
  public static void main(String[] args)
  {
    Building b = new Building();
    Cube c1, c2;
    Pyramid p1;
    c1 = new Cube(3);
    c2 = new Cube(5);
    p1 = new Pyramid(4, 5);
    b.add_block(c1);
    b.add_block(c2);
    b.add_block(p1);
    double v = b.get_volume();

    Cylinder s1;
    s1 = new Cylinder(3, 5);
    b.add_block(s1);
  }
}
```

Inheritance

- A method in a superclass can access *indirectly* the attributes and methods of a subclass.

```
class Parent {
    void m()
    {
        System.out.print("1 ");
    }
    void q()
    {
        System.out.print("2 ");
        m();
    }
}
class Child extends Parent {
    void m()
    {
        System.out.print("3 ");
    }
}
```

Inheritance

- A method in a superclass can access *indirectly* the attributes and methods of a subclass.

```
public class Inh1
{
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        Child obj2 = new Child();
        obj1.m();   // Prints "1"
        obj2.m();   // Prints "3"
        obj1.q();   // Prints "2 1"
        obj2.q();   // Prints "2 3"
    }
}
```

Polymorphism

```
class Creature
{
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    move();
  }
}
```

Polymorphism

```
class Human extends Creature
{
  void move()
  {
    System.out.println("Walking...");
  }
}

class Martian extends Creature
{
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.dance();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick); // Polymorphic call
  }
}
```

Parametric Polymorphism

```
class Penguin extends Creature
{
  void move()
  {
    System.out.println(“Stumble”);
  }
}
```

Parametric Polymorphism

```
public class ZooTest
{
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    Penguin paco = new Penguin();
    my_zoo.animate(ernesto);
    my_zoo.animate(yannick);
    my_zoo.animate(paco);
  }
}
```

Polymorphism

```
class Creature
{
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    move();  // Dynamic-dispatch
  }
}
```

Polymorphism

```
class Creature
{
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    this.move();  // Dynamic-dispatch
  }
}
```

Abstract classes

- A class with default behaviour:

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println(“Here we go...”);
  }
}
```

- An abstract class: subclasses must provide implementation

```
abstract class Creature {
  boolean alive;
  abstract void move();
}
```

Abstract classes

- An abstract class is a class that has at least one abstract method
- An abstract method is a method which is not implemented (i.e. has no body) and must be overriden (i.e. must implemented by the subclasses.)
- An abstract class is used to represent an abstract concept which captures the common structure and behaviour of several classes, but leaves some detail to the subclasses.
- Abstract classes force the use of parametric polymorphism.

Abstract classes

- There cannot be instances of abstract classes.

```
Creature kowe = new Creature(); // Wrong!
//because
kowe.move(); // What would be executed here?
```

- The abstract methods *must* be implemented in the subclasses of an abstract class (unless the subclass itself is also abstract.) This is, there is no default behaviour for an abstract method.

Abstract classes

- An abstract class can have non-abstract methods (which usually represent the "default behaviour" of a method:)

```
abstract class Creature
{
  boolean alive;
  abstract void move();
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    move();
  }
}
```

Interfaces

- Interfaces are (equivalent to) purely abstract classes, i.e. classes where all the methods are abstract

```
interface Creature
{
  void move();
  void dance();
}
```

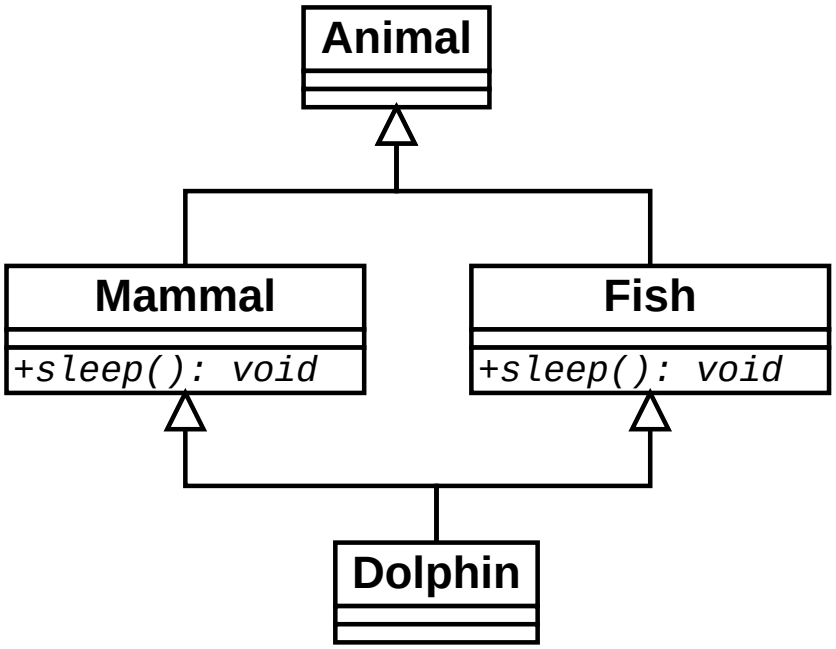
is the same as

```
abstract class Creature
{
  abstract void move();
  abstract void dance();
}
```

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println(“I’m walking...”);
  }
  void dance()
  {
    System.out.println(“1-2-3-spin-1-2-3-step...”)
  }
  void jump()
  {
    System.out.println(“Up and down...”);
  }
}
```

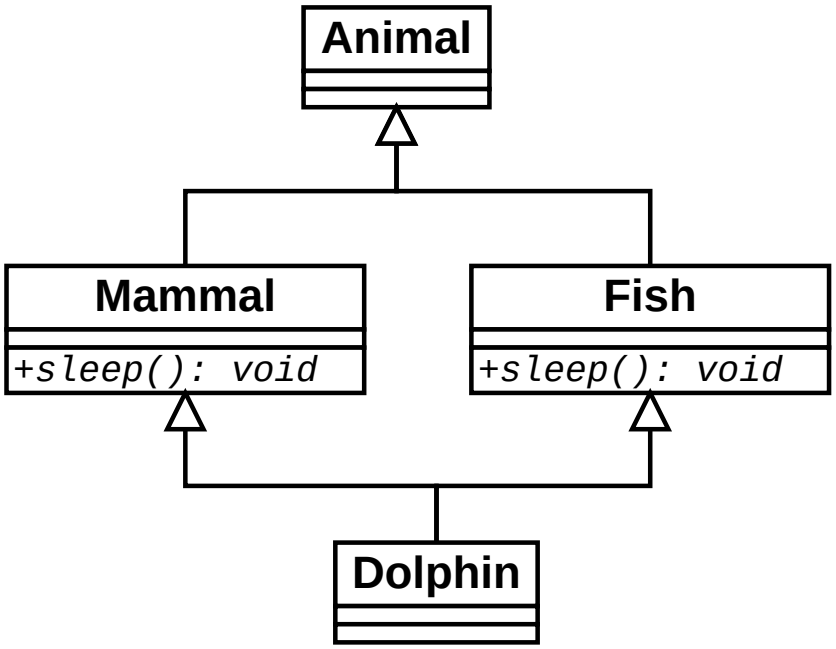
Multiple inheritance




```
class B { ... }
class C { ... }
class A extends B, C { ... } // Error
```

- Java does not support multiple inheritance

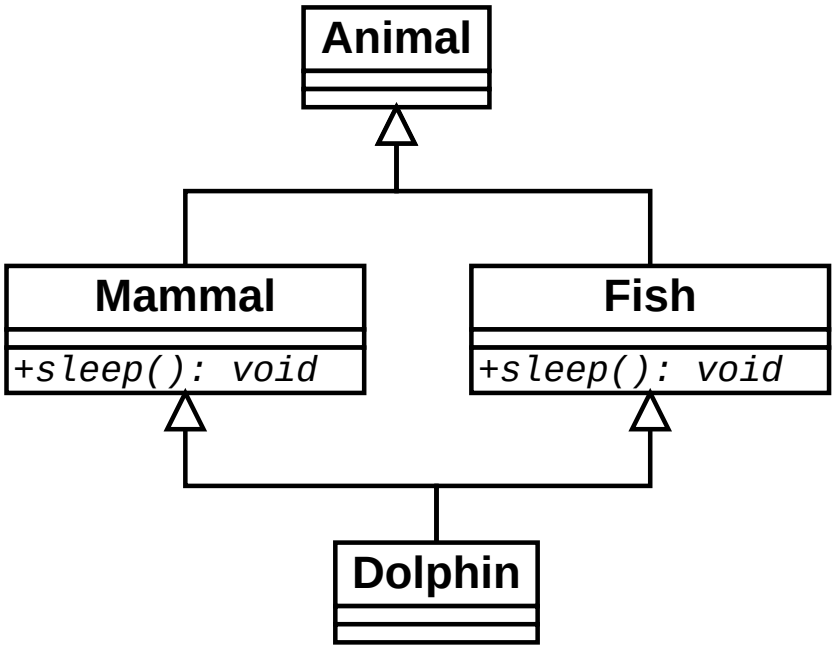
Multiple inheritance




```
interface B { ... }
interface C { ... }
class A implements B, C { ... } // OK
```

- Java does supports multiple inheritance for interfaces

Multiple inheritance




```
class B { ... }
interface C { ... }
class A extends B implements C { ... } // OK
```

- You can extend a class and implement interfaces

Designing with Interfaces

- Abstraction: (hiding details)
  - Separation between interface and implementation
  - An interface can be implemented by many classes
  - The client of the interface ...
  - ...doesn't need to know how it is implemented
  - ...only needs to know what are its methods

Designing with Interfaces

- *Design by contract*: declaring an interface is like making a contract
  - Agreement between the client of the class and its implementer
  - What is agreed: the *method signatures*
  - *Method signature*:
    * method name,
    * number and type of parameters, and
    * return type

Using interfaces for generalization

```
class CDPlayer
{
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Using interfaces for generalization

```
class DVDPlayer
{
  boolean stopped, recording;
  DVDPlayer() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record() { recording = true; }
}
```

Interfaces

```
interface MediaPlayer
{
  void play();
  void ff();
  void pause();
  void stop();
}
```

Using interfaces for generalization

```
class CDPlayer implements MediaPlayer
{
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Using interfaces for generalization

```
class DVDPlayer implements MediaPlayer
{
  boolean stopped, recording;
  DVDPlayer() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record() { recording = true; }
}
```

Interfaces

```
class PlayerTest
{
  static void test(MediaPlayer p)
  {
    p.play();
    p.ff();
    p.pause();
    p.play();
    p.stop();
  }
}
```

Interfaces

```
class SoundStudio
{
  public static void main(String[] args)
  {
    MediaPlayer[] players = { new CDPlayer(),
                              new DVDPlayer(),
                              new CDPlayer() };
    for (int i = 0; i < players.length; i++)
    {
      PlayerTest.test(players[i]);
    }
  }
}
```

Abstract classes

```
abstract class MediaPlayer
{
  boolean stopped;
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  abstract void stop();
}
```

The end