Review

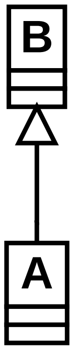
- Inheritance:
  - Represents the “is-a” relationship between classes
  - Represents specialization of classes (subsets)
  - Represents a way of describing alternatives (alternative subclasses)
  - Is a mechanism for reusability

Inheritance

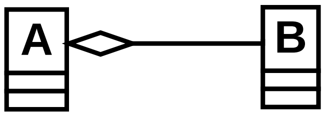
- Whenever we have a situation which states that "every A is a B", we model this as

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

- All attributes and methods from the parent class (or super class) B are "inherited" by the subclass (or derived class) A.
- Class A can have (and usually does have) additional attributes and methods.



represents:
"every A is a B"
(inheritance)



represents:
"every A has a B"
(aggregation)

Inheritance

- The silogism "if every A ***is a*** B and every B ***has a*** C then every A ***has a*** C", means that all the attributes that B has, are also attributes of A. A may have other attributes as well which B doesn't. A is more specific or specialized than B.

```
class C { ... }
class B {
  C v;
  // ...
}
class A extends B {
  // Has an implicit C v;
  // ...
}
```

Inheritance

- Inheritance represents also a spectrum of possibilities or alternatives, given by the subclasses of a class
- If every B is an A and every C is an A, and nothing else is an A, then an A is either a B or a C
  - (e.g. if every racing car is a car, and every sedan is a car, and nothing else is a car, then a car is either a racing car or a sedan.)

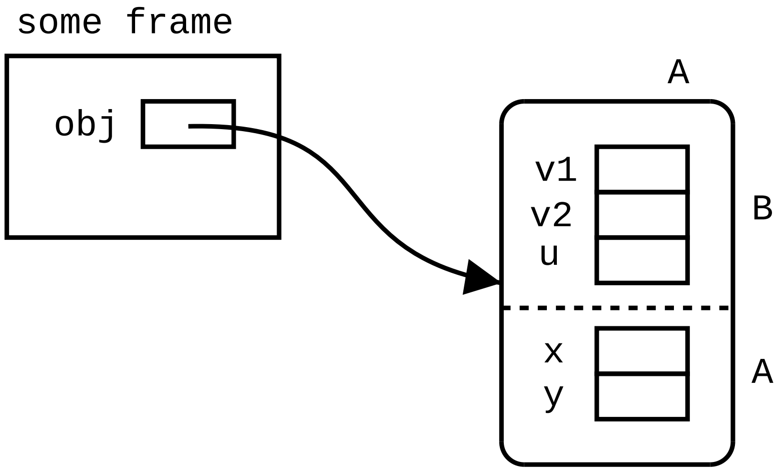
```
class Animal { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
class Bird extends Animal { ... }

// In some client
Animal a1 = new Dog();
Animal a2 = new Cat();
Animal a3 = new Bird();
Dog d = new Animal(); // Wrong!
```

Inheritance

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... }
}
class A extends B {
  E x;
  C y;
  void p() { ... }
  void s() { ... }
}
```

Inheritance

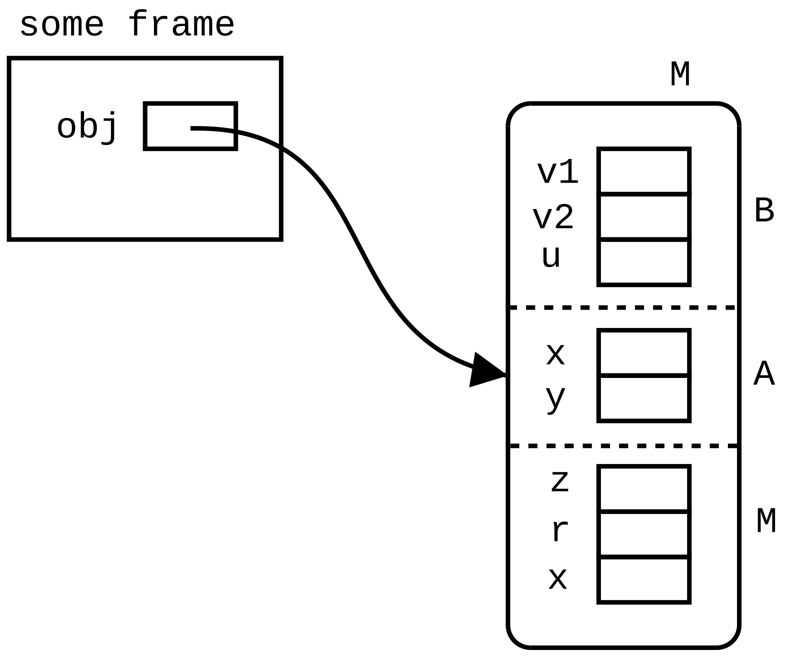
```
// In some client
A obj = new A();
obj.p();
obj.m();
// We can refer to ... obj.x ... obj.y ...
// ... obj.u ... obj.v1 ... obj.v2 ...
```



Shadowing variables

- An attribute or instance variable can be redefined in a subclass. In this case we say that the variable in the subclass *shadows* the variable in the parent class.

```
class M extends A {
  E z;
  D r, x;
  void q() { ... }
}
```



Accessing variables from the super class

- The `super` reference is used to access an attribute or method in a parent class.

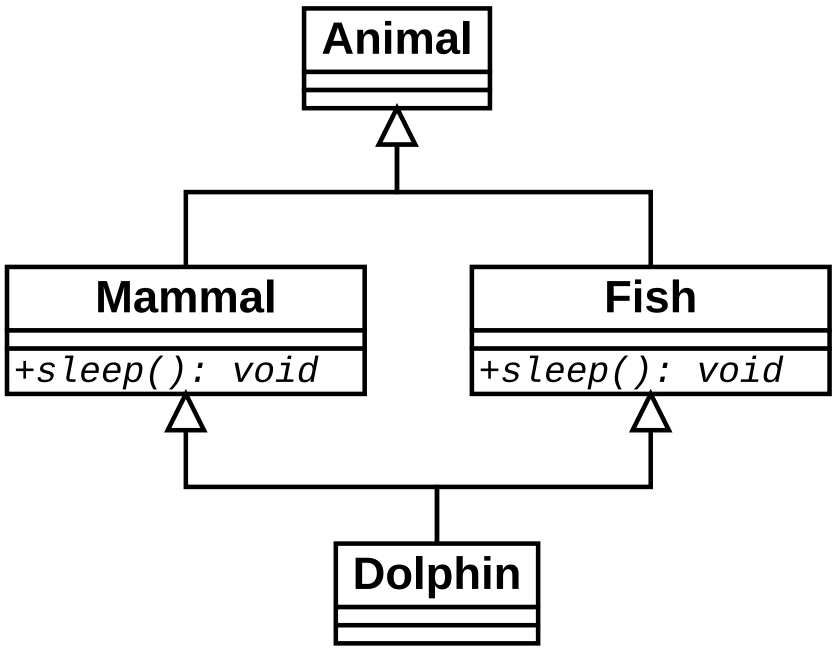
```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
}
```

Overriding methods

- A method can be redefined in a subclass. This is called *overriding* the method.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
  void p()
  {
    ...
  }
}
```

Multiple inheritance



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

- Java does not support multiple inheritance

Polymorphism

- Polymorphism means “many forms.”
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- 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)

```
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)

```
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

```
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
    // move *must* be defined in class Creature
  }
}

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);
  }
}
```

Accessing super

```
class Human extends Creature {
  void move()
  {
    super.move();
    System.out.println(“Walking...”);
  }
}
class Martian extends Creature {
  void move()
  {
    super.move()
    System.out.println(“Crawling...”);
  }
}
```

Casting and instanceof

- Casting is like putting a special lens on an object
- A casting expression is of the form

  (*type*) *expr*

  where `type` is any type (primitive or user-defined) and `expr` is an expression which evaluates to an object reference whose type is compatible with `type`.
- Not all casts are possible

```
(int) ‘‘Hello’’
(Engine) yannick
```

Casting

- If a variable is a reference of type A, it can be assigned any object whose type is a subclass of B.

```
Human greg = new Human();
Creature c = greg;
```

- But a reference of type B cannot be assigned directly reference of type A, if B is a subclass of A (because A has less attributes than required by B):

```
Creature d = new Creature();
Martian m = d;       // Error
```

- ...however, if we know that a reference x of type A points to an object of type B (and B is a subclass of A,) then we can force to see x as being of type B by using a casting expression:

```
Creature e = new Martian();
Martian f = (Martian)e;
```

Casting

```
class Creature {
  boolean alive;
  // ...
}
class Martian extends Creature {
  int legs, wings;
  // ...
}

// Somewhere else...
Creature d = new Creature();
Martian m = d;
int k = m.legs + m.wings; // Error!
// ...because d does not have legs or wings
```

Casting

```
class Creature {
  boolean alive;
  void move() { ... }
}
class Martian extends Creature {
  int legs, wings;
  void move() { ... }
  void hop() { ... }
}

// Somewhere else...
Creature d = new Creature();
Martian m = d;
m.hop(); // Error!
// ...because d cannot hop
```

Casting

- ...however, if we know that a reference x of type A points to an object of type B (and B is a subclass of A,) then we can force to see x as being of type B by using a casting expression:

```
Creature e = new Martian();
Martian f = (Martian)e;
int n = f.legs * f.wings;
((Martian)e).hop(); // same as f.hop();
```

Checking the type of a reference

- To find out whether a reference `r` is an instance of a particular class `C` we use the boolean expression:

```
r instanceof C
```

- This is normally used whenever we do casting:

```
class Human extends Creature {
  void move()
  {
    System.out.println(“Walking...”);
  }
  void jump()
  {
      System.out.println("Up and down");
  }
}
```

Checking the type of a reference

```
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
class Zoo {
  void move(Creature c)
  {
    if (c instanceof Human)
      ((Human)c).jump();
    else if (c instanceof Martian)
      ((Martian)c).hop();
    c.move();
  }
}
```

Narrowing and Widening casts

- Suppose class A has B as a subclass.
- Narrowing casts make a reference to a B object into an A object

```
B z = new B();
A w = (A)z; // Narrowing; Same as A w = z;
```

- Widening casts make a reference to an A object into a B object

```
A x = new B();  // Narrowing
B y = (B)x;  // Widening
```

- Sometimes it is necessary to make an explicit narrowing conversion if we want to force an object to behave as one of its ancestors, for example to access some overriden method.

Narrowing and Widening casts

```
class FlyingMartian extends Martian {
  void move()
  {
    System.out.println(“Gliding...”);
  }
}

class ZooTest {
  public static void main(String[] args)
  {
    FlyingMartian peng = new FlyingMartian();
    peng.move();
    ((Martian)peng).move();
    ((Creature)peng).move();
    ((Human)peng).move(); // Error peng is not Hum
  }
}
```

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 run-time 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.

Object

- Object is a class in the standard Java library which is a superclass to all.
- It contains methods

```
public boolean equals(Object o)
protected Object clone()
public String toString()
public Class getClass()
```

- A method whose argument is of type Object can receive any object from any class as argument. (maximum possible polymorphism.)
- Whenever an object appears in a String expression, the method toString is invoked automatically

Object

```
class Human {
  String name;
  public String toString()
  {
    return ‘‘My name is ‘‘+name;
  }
}
class Test {
  public static void main(String[] args)
  {
    Human h = new Human();
    h.name = ‘‘Kelly’’;
    String s = ‘‘’’+h;
    // Same as String s = ‘‘’’+h.toString();
  }
}
```

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();
}
```

McGill

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, hungry;
  abstract void move();
  void eat()
  {
    System.out.println(“Hmmm...”);
    hungry = false;
  }
}
```

Interfaces

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

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

  is the same as

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

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println(“I’m walking...”);
  }
  void eat()
  {
    System.out.println(“I’m eating...”);
  }
  void jump()
  {
    System.out.println(“Up and down...”);
  }
}
```

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 TapeRecorder {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

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

Interfaces

```
class CDPlayer implements MusicPlayer {
  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;
  }
}
```

Interfaces

```
class TapeRecorder implements MusicPlayer {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

```
class PlayerTest {
  static void test(MusicPlayer p)
  {
    p.play();
    p.ff();
    p.pause();
    p.play();
    if (p instanceof TapeRecorder) {
      ((TapeRecorder)p).record(new Tape());
    }
    p.stop();
  }
}
```

Interfaces

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

The end