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

```
class A {
  void p() { ... }
}
class B extends A {
  void p() { ... }
}
class C extends A {
  void p() { ... }
}
```

Polymorphism

```
class D {
  void m()
  {
    A obj;
    obj = new B();
    obj.p();
    obj = new C();
    obj.p();
  }
}
```

Polymorphism

```
class D {
  void q(A obj)
  {
    obj.p();
  }
  void m()
  {
    A obj;
    obj = new B();
    q(obj);
    obj = new C();
    q(obj);
  }
}
```

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...");
  }
}
```

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
  }
}
```

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)

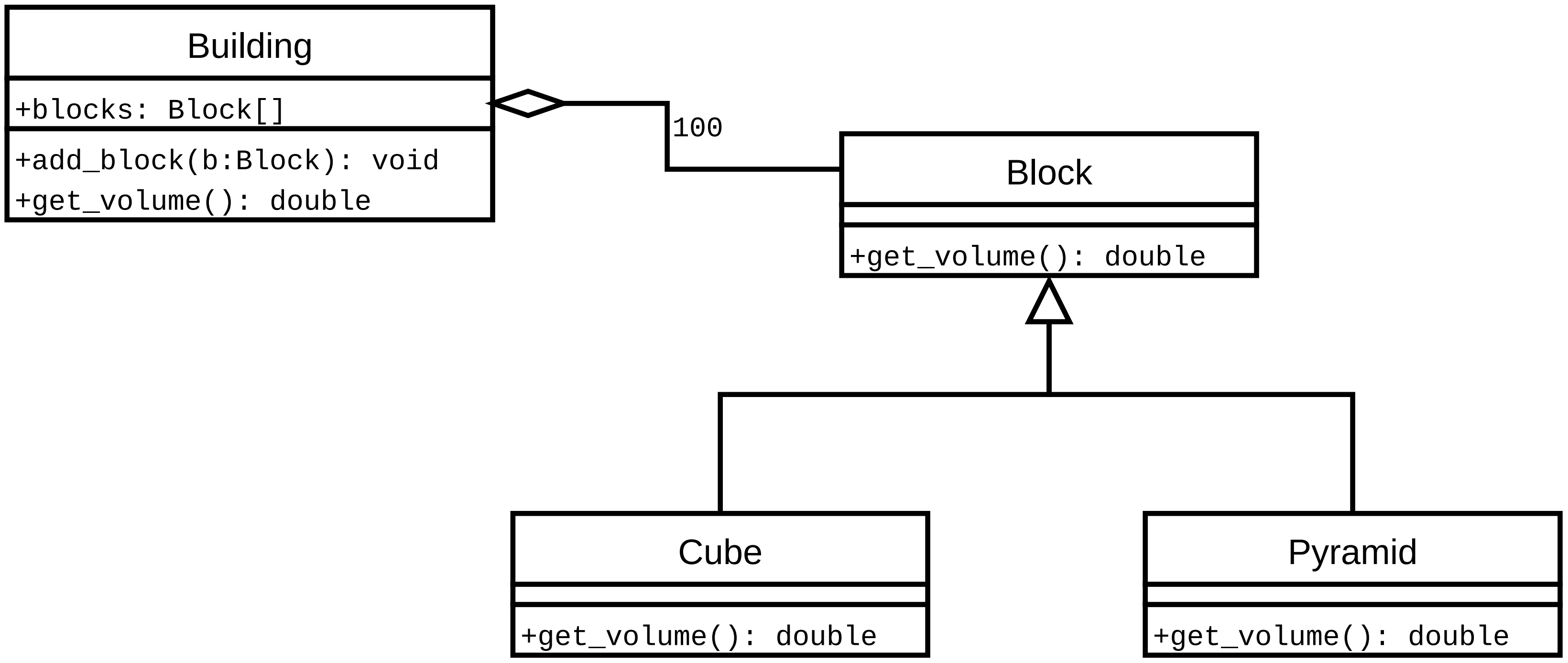
```
class Penguin extends Creature {
  void stumble()
  {
    System.out.println(“Ouch”);
  }
}

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

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_l
        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);
    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;
    }
}
```

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 animate(Creature c)
  {
    if (c instanceof Human)
      ((Human)c).jump();
    else if (c instanceof Martian)
      ((Martian)c).hop();
    c.move();
  }
}
```

McGill

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
  }
}
```

McGill

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, 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.
    }
  }
}
```

Abstract classes

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

Abstract classes

```
class CDPlayer extends MusicPlayer {
  int song;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void ff() { song++; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Abstract classes

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

The end