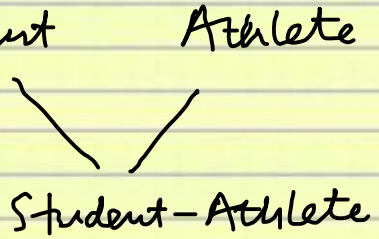
Java chooses methods at run-time, dependant on the type of the object for which the object is called.

```
class Vehicle {
   public void f() { System.out.println("f in Vehicle"); }
}
class Car extends Vehicle {
   public void f() { System.out.println("f in Car"); }
}
```

```
Vehicle v = new Vehicle();
Car c = new Car();
v.f() => prints "f in Vehicle"
c.f() => prints "f in Car"
v = c; => OK - upcasting
v.f(); => still prints "f in Car"
((Vehicle)c).f(); => still prints "f in Car"
c = (Car)v; => down casting OK because v refers to Car
```

Java uses single inheritance – a class may only inherit from one other class.

Multiple inheritance is possible:



This is not possible in Java.

Possible in Java is interface inheritance:

```
public interface I1 {
    public void f1();
    public void f2();
}

public interface I2 {
    public void f3();
    public void f4();
}

public class C implements I1, I2 {
    .... implement all inherited methods
}
```

```
I1 obj = new C();
```

Polymorphism

- overloading of methods, especially constructors; give multiple methods with same name but different parameters (type and/or number)
- late binding of methods

overloading:

```
class C {
  void f() {...}
  void f(int a) {...}
  void f(C1 c) {...} // C1 is any class type
  void f(int a, int b) {...}
}

C c = new C();      c.f(); => calls f()
                    c.f(3); => calls f(int a)
                    c.f(obj) => calls f(C1 c)
```

Also with constructors:

```
class C {
    private int x;
    public C(int x) {
        this.x = x;
    }
    public C() {
        x = 0;
    }
}
```

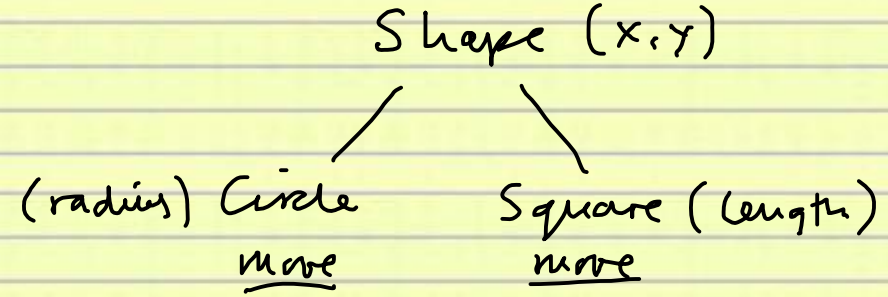
C c = new C(); => calls default ctor

C c = new C(10); => calls C(int x)

Late binding methods

- Late means run-time

Java does not compile code to call a particular method, but leaves it until run-time, when the type of object for which the method is called determines which method.



Could store multiple circles in an array Circle[]
Same for squares.
However, if we want to store both in an array, we could try using Shape instead.

```
Shape [] sa = new Shape[10];
Circle c1 = new Circle(10, 20, 50);
                       ↑  ↑  ↑
                       x  y radius
Square s1 = new Square(20, 30, 25);
                       ↑  ↑  ↑
                       x  y length

sa[0] = c1;
sa[1] = s1;

for (int i = 0; i < 10; i++) {
    sa[i].move(100, 200);
}
```

The appropriate move (Circle or Square) will be called depending on the type of the object.

sa[0].move calls move in Circle
sa[1].move calls move in Square

Collections in Java

- array
- ArrayList
- Stack
- HashSet
- HashMap

} can contain any type of object

array: `double [] da1 = new double[10];`

type is array of doubles

```
List<Double> da2 = new ArrayList<Double>();
```

interface type — class type corresponding to double — class type

```
Set<String> s1 = new HashSet<String>();

Stack<Double> st1 = new Stack<Double>();

Map<String, List<Integer>> map =
    new HashMap<String, List<Integer>>();
```

Operations on Stack: ← converted to Double ("boxing")

```
st.push(12.2);
st.push(13.3);
st.pop();

for (Double d : st) {
    ... d ...
}

Iterator<Double> it = st.iterator();
while (it.hasNext()) {
    ... it.next() ...
}

while (!st.isEmpty()) {
    ... st.pop() ...
}
```