We might want to require that every Person can dance. But the way a Lecturer dances is not likely to be the same as the way a Student dances...

```java
class Person {
    public void dance() {
        jiggle_a_bit();
    }
}

class Student extends Person {
    public void dance() {
        body_pop();
    }
}

class Lecturer extends Person {
}
```

Person defines a 'default' implementation of `dance()`

Student overrides the default

Lecturer just inherits the default implementation and jiggles
Student s = new Student();
Person p = (Person)s;
p.dance();

- Assuming Person has a default dance() method, what should happen here??

- **Option 1**
  - Compiler says “p is of type Person”
  - So p.dance() should do the default dance() action in Person

- **Option 2**
  - Compiler says “The object in memory is really a Student”
  - So p.dance() should run the Student dance() method
The Canonical Example I

- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects

**Option 1**

- Keep a list of Circle objects, a list of Square objects,...
- Iterate over each list drawing each object in turn
- What has to change if we want to add a new shape?

<table>
<thead>
<tr>
<th>Circle</th>
<th>+ draw()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>+ draw()</td>
</tr>
<tr>
<td>Oval</td>
<td>+ draw()</td>
</tr>
<tr>
<td>Star</td>
<td>+ draw()</td>
</tr>
</tbody>
</table>
The Canonical Example II

- Option 2
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
for every Shape s in myShapeList
  if (s is really a Circle)
    Circle c = (Circle)s;
    c.draw();
  else if (s is really a Square)
    Square sq = (Square)s;
    sq.draw();
  else if...
```

- What if we want to add a new shape?
The Canonical Example III

- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

What if we want to add a new shape?

```java
For every Shape s in myShapeList
s.draw();
```
Implementations

- Java
  - All methods are polymorphic. Full stop.
- Python
  - All methods are polymorphic.
- C++
  - Only functions marked \textit{virtual} are polymorphic

Polymorphism is an extremely important concept that you need to make sure you understand...
Abstract Methods

- There are times when we have a definite concept but we expect every specialism of it to have a different implementation (like the draw() method in the Shape example). We want to enforce that idea without providing a default method.

- E.g. We want to enforce that all objects that are Persons support a dance() method.
  - But we don't now think that there's a default dance()

- We specify an **abstract** dance method in the Person class.
  - i.e. we don't fill in any implementation (code) at all in Person.

```java
class Person {
    public void dance();
}

class Student extends Person {
    public void dance() {
        body_pop();
    }
}

class Lecturer extends Person {
    public void dance() {
        jiggle_a_bit();
    }
}
```
Abstract Classes

- Before we could write `Person p = new Person()`
- But now `p.dance()` is undefined
- Therefore we have implicitly made the class abstract i.e. It cannot be directly instantiated to an object
- Languages require some way to tell them that the class is meant to be abstract and it wasn't a mistake:

```java
public abstract class Person {
    public abstract void dance();
}
```

Java

```cpp
class Person {
    public:
        virtual void dance()=0;
}
```

C++

- Note that an abstract class can contain state variables that get inherited as normal
- Note also that, in Java, we can declare a class as abstract despite not specifying an abstract method in it!!
Representing Abstract Classes

- Person
  + dance()

Student
  + dance()

Lecturer
  + dance()
What if we have a Lecturer who studies for another degree?

If we do as shown, we have a bit of a problem

- StudentLecturer inherits two different dance() methods
- So which one should it use if we instruct a StudentLecturer to dance()?

The Java designers felt that this kind of problem mostly occurs when you have designed your class hierarchy badly

Their solution? **You can only extend (inherit) from one class in Java**

- (which may itself inherit from another...)
- This is a Java oddity (C++ allows multiple class inheritance)
Java has the notion of an interface which is like a class except:
- There is no state whatsoever
- All methods are abstract

For an interface, there can then be no clashes of methods or variables to worry about, so we can allow multiple inheritance.

```java
interface Drivable {
    public void turn();
    public void brake();
}

interface Identifiable {
    public void getIdentifier();
}

class Bicycle implements Drivable {
    public void turn() {...}
    public void brake() {… }
}

class Car implements Drivable, Identifiable {
    public void turn() {...}
    public void brake() {… }
    public void getIdentifier() {...}
}
Recap

- Important OOP concepts you need to understand:
  - Modularity (classes, objects)
  - Data Encapsulation
  - Inheritance
  - Abstraction
  - Polymorphism
Lifecycle of an Object
You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.

It's a method that gets called when the object is constructed, and it goes by the name of a constructor (it's not rocket science).

We use constructors to initialise the state of the class in a convenient way.

- A constructor has the same name as the class
- A constructor has no return type specified
Constructor Examples

Java

public class Person {
    private String mName;

    // Constructor
    public Person(String name) {
        mName = name;
    }

    public static void main(String[] args) {
        Person p = new Person("Bob");
    }
}

C++

class Person {
    private:
        std::string mName;

    public:
        Person(std::string &name) {
            mName = name;
        }

    public:
        Person(std::string &name) {
            mName = name;
        }

    public static void main(String[] args) {
        Person p("Bob");
    }

    int main(int argc, char ** argv) {
        Person p("Bob");
    }
}
public class Person {
    private String mName;

    public static void main(String[] args) {
        Person p = new Person();
    }
}

- If you specify no constructor at all, the Java fills in an empty one for you
- The default constructor takes no arguments