Department of Computer Science
University of Bristol

COMSM0086 – Object-Oriented Programming

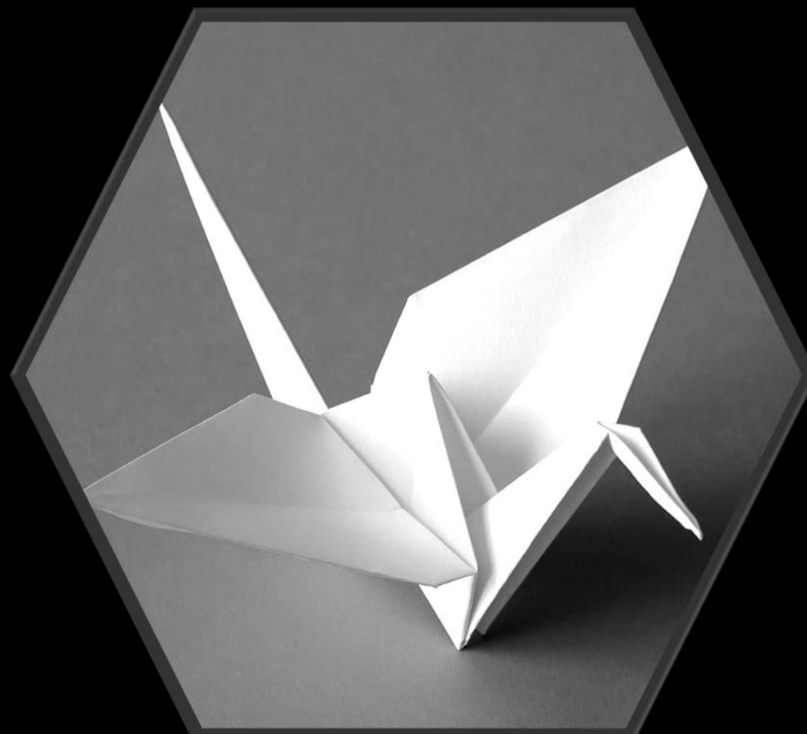
POLYMORPHISM AND VISITOR

Sion Hannuna | sh1670@bris.ac.uk
Sion Lock | simon.lock@bris.ac.uk

“Polymorphism is very useful for practical programming because it allows the **uniform manipulation** of objects of different, but related sub-classes using methods of a common super-class.”

Jürgen Winkler

Recap: Abstract Classes



Abstract Classes, Abstract Methods

- to prevent us from making instances of a class we apply the **abstract** keyword to the class
- abstract classes are often ones that are purely conceptual without any instances (e.g. a **mammal**, a generic **Shape**, an **AbstractRobot**)

AbstractRobot.java

```
abstract class AbstractRobot extends Robot {
  abstract void greet(AbstractRobot other);
  abstract void greet(TranslationRobot other);
  abstract void greet(CarrierRobot other);
}
```

no instance of AbstractRobot is ever allowed

abstract methods provide no implementation in the class, however, sub-classes may provide implementations

- usually an **abstract** class contains **abstract** methods, that is methods which are declared, but supply no implementation (any non-abstract sub-class is forced to implement **all** these methods)
- a class with one or more abstract methods **must be** declared abstract itself

Recap: Polymorphism

Recap: Sub-Classing & Polymorphism

- a **sub-class** can be understood as a sub-type that supports both **inheritance** (i.e. sub-classes receive all features for free from the parent) and **polymorphism** (i.e. features of sub-classes can be used in place of a feature of a class)

SUB-CLASS = SUB-TYPE with INHERITANCE + POLYMORPHISM

```
abstract class AbstractRobot extends Robot {
  ...
}
```

```
class TranslationRobot extends AbstractRobot {
  ...
}
```

```
  ...
  Robot c3po = new Robot();
  TranslationRobot c4po = new TranslationRobot("e");
  ...
  Robot c5po = new TranslationRobot("e");
  ...
  // BOGUS: TranslationRobot c6po = new Robot("e");
  ...
```

‘extends’ signals inheritance from Robot class

a reference can point to an object of a sub-class of the reference type

Recap: Double / multiple Dispatch

Double Dispatch

- if we want to make the selection of method dynamic in more than one type we need to implement **multiple dispatch**
- Java does not explicitly supply a single mechanism for it
- however, we can be cunning and utilise single dispatch **twice**
- to do this, we need to dynamically dispatch on a receiver as before, but also turn the otherwise static parameter of the call into a dynamic receiver itself within the method that is dynamically dispatched

different options are selected during runtime

AbstractRobot.java

```
abstract class AbstractRobot extends Robot {
  abstract void greet(AbstractRobot other);
  abstract void greet(TranslationRobot other);
  abstract void greet(CarrierRobot other);
}
```

CarrierRobot.java

```
class CarrierRobot extends AbstractRobot {
  ...
  void greet(TranslationRobot other) {
    talk("'Hello from a TranslationRobot to a CarrierRobot.'"); }

  void greet(CarrierRobot other) {
    talk("'Hello from a CarrierRobot to another.'"); }

  void greet(AbstractRobot other) {
    other.greet(this);
} }
```

2nd dispatch dynamically using the incoming parameter

TranslationRobot.java

```
public class TranslationRobot extends AbstractRobot {
  ...
  void greet(TranslationRobot other) {
    talk("'Hello from a TranslationRobot to another.'"); }

  void greet(CarrierRobot other) {
    talk("'Hello from a CarrierRobot to a TranslationRobot.'"); }

  void greet(AbstractRobot other) {
    other.greet(this);
} }
```

DispatchWorld.java

```
class DispatchWorld {
  public static void main (String[] args) {
    AbstractRobot c3po = new TranslationRobot("e");
    AbstractRobot c4po = new TranslationRobot("o");
    AbstractRobot c5po = new CarrierRobot();
    AbstractRobot c6po = new CarrierRobot();
    c3po.greet(c4po);
    c5po.greet(c4po);
    c4po.greet(c5po);
    c5po.greet(c6po);
} }
```

1st dispatch dynamically on receiver

Visitors

(... a first Meeting with the 'Pattern' Family ...)

(... Standard solutions to common problems ...)

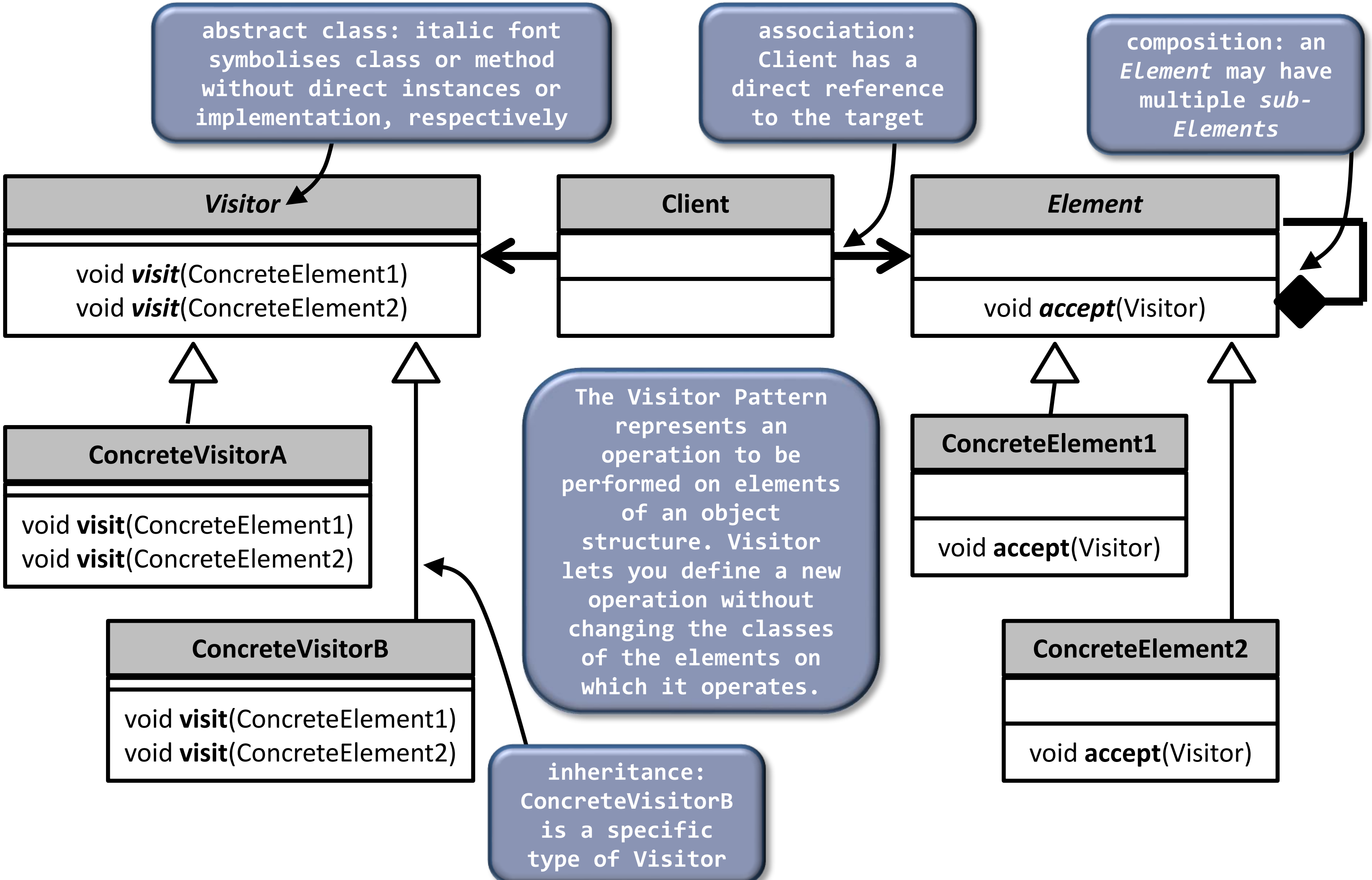
How is the Visitor Pattern useful?

The Visitor Pattern facilitates the addition of new operations to existing object structures without modifying those structures *(maybe look up open closed principle)*.

A visitor class is created that implements all of the appropriate specializations.

The visitor takes the instance reference as input, and implements the goal through double dispatch.

A Version of the Visitor Pattern



Toy example – mammals getting visited

For this example, try and recognise the various elements of the visitor pattern and understand their interactions

- *Visitor (abstract superclass)*
  - *Concrete Visitor(s)*
- *Element (abstract superclass)*
  - *Concrete Element(s)*
- *Client (coordinates things in this example)*

CODE WALK THROUGH

Real world example – cash back offers

A bank offers 3 types of credit card which offer annual subscription fees vs cashback offers: as trade-offs

	Bronze (free)	Silver (£250)	Gold (£500)
Fuel	1%	2%	3%
Tesco	0.5%	1%	2.5%
Cycle republic	0%	1%	5%

Based on: https://youtu.be/TeZqKnC2gvA

Real world example – cash back offers

Consider what classes you will have for the pattern's components:

- *Visitor*
  - *Concrete Visitor(s)*
- *Element*
  - *Concrete Element(s)*
- *Client*

	Bronze (free)	Silver (£250)	Card (£500)
Fuel	1%	2%	3%
Tesco	0.5%	1%	2.5%
Cycle republic	0%	1%	5%

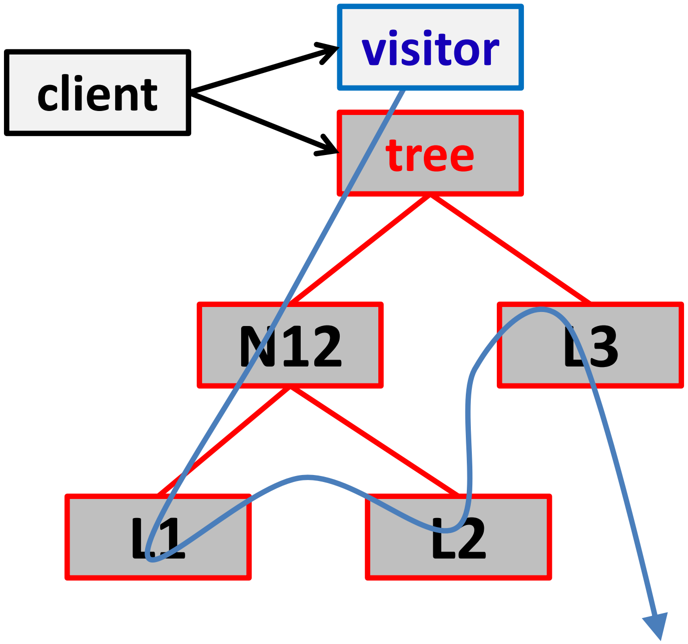
CODE WALK THROUGH

Based on: https://youtu.be/TeZqKnC2gvA

Example: A Binary Tree that can be visited...

- consider the following situation:

  - we have a target object structure (for instance a binary **Tree** where every node is either a **Leaf** or a **Fork** with references to two **Tree** objects)
  - other objects, known as **clients**, would like to perform operations that require information from possibly all sub-objects of the target structure (for instance summing up values from all the leaves of the tree structure)
  - however, we would like any operations to be defined **independently** from the object structure itself
  - the operations should therefore be encapsulated in a separate object (which we shall call the **Visitor**)




```
abstract class Tree { // the client
  abstract void accept(Visitor v);
}
```

```
class Fork extends Tree {
  Tree l;
  Tree r;
  ...
} }
```

```
class Leaf extends Tree {
  int value;
  ...
} }
```

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

all Tree objects know how to accept a Visitor object

all Visitor objects know how to visit all types of trees

The Tree-side of Things ... that can be visited ...

- we need an abstract class **Tree** that is parent to two non-abstract specialisations of **Tree**: the **Fork** and **Leaf** classes
- we also demand that all trees should accept a **Visitor** object (by calling its **visit** method with itself as parameter)

Tree.java

```
abstract class Tree {
  abstract void accept(Visitor v);
}
```

any Tree object must be able to accept a Visitor object, non-abstract sub-classes MUST implement this method

Fork.java

```
class Fork extends Tree {
  Tree l;
  Tree r;

  Fork(Tree l, Tree r) {
    this.l = l;
    this.r = r;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

being a Fork object means to hold references to two Tree objects

Leaf.java

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

being a Leaf object means to have a value

accepting a visitor object means to call its visit method handing this as parameter

VisitorWorld: Interaction of the Tree and the Visitor

- we also have an abstract **Visitor** object that knows how to visit **Fork** and **Leaf** objects
- a particular, non-abstract **SumVisitor** implements **visit**
- now we can message a **Tree** to accept a **SumVisitor**...

Visitor.java

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

SumVisitor.java

```
class SumVisitor extends Visitor {
  int total;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  } // forward to sub-trees

  @Override
  void visit(Leaf leaf) {
    total += leaf.value;
} }
```

Tree.java

```
abstract class Tree {
  abstract void accept(Visitor v);
}
```

```
class VisitorWorld {

  public static void main (String[] args) {
    Tree l1 = new Leaf(1);
    Tree l2 = new Leaf(2);
    Tree l3 = new Leaf(3);
    Tree n12 = new Fork(l1, l2);
    Tree tree = new Fork(n12, l3);
    SumVisitor visitor = new SumVisitor();
    tree.accept(visitor);
    System.out.println( "total: " +
                        visitor.total);
} }
```

Leaf.java

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

Fork.java

```
class Fork extends Tree {
  Tree l;
  Tree r;

  Fork(Tree l, Tree r) {
    this.l = l;
    this.r = r;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

#1) we ask a tree to accept our visitor, dynamic dispatch on the tree sub-type

#2) a specific tree accepts by forwarding (dynamic dispatch) to the specific visitor

#3) visitor handles the visits

#4) we access the result in the attribute total of the visitor object

Different Visitors – No Change to the Tree Classes ...

Visitor.java

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

SumVisitor.java

```
class SumVisitor extends Visitor {
  int total;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  } // forward to sub-trees

  @Override
  void visit(Leaf leaf) {
    total += leaf.value;
} }
```

ProdVisitor.java

```
class ProdVisitor extends Visitor {
  int total = 1;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  } // forward to sub-trees

  @Override
  void visit(Leaf leaf) {
    total *= leaf.value;
} }
```

VisitorWorld.java

```
class VisitorWorld {

  public static void main (String[] args) {
    Tree l1 = new Leaf(1);
    Tree l2 = new Leaf(2);
    Tree l3 = new Leaf(3);
    Tree n12 = new Fork(l1, l2);
    Tree tree = new Fork(n12, l3);
    SumVisitor sumV = new SumVisitor();
    ProdVisitor prodV = new ProdVisitor();
    tree.accept(sumV);
    tree.accept(prodV);
    System.out.println( "sum: " +
                        sumV.total +
                        " prod: " +
                        prodV.total );
} }
```

FLEXIBLE: we can define numerous different specialisations of Visitor, all providing different operations (e.g. sums, products..) on our Tree object structure WITHOUT changing the Tree class or any of its sub-classes

NEAT: calling <u>any</u> operation on a Tree object can be achieved by letting a Tree object accept a Visitor object that implements the operation

Leaf.java

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

no change!

Fork.java

```
class Fork extends Tree {
  Tree l;
  Tree r;

  Fork(Tree l, Tree r) {
    this.l = l;
    this.r = r;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

no change!

Tree.java

```
abstract class Tree {
  abstract void accept(Visitor v);
}
```

no change!

code extended from N. Wu

Decoupling of Operations and Data Structures!

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

Visitor.java

```
class SumVisitor extends Visitor {
  int total;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  } // forward to sub-trees

  @Override
  void visit(
```

Operations

```
class ProdVisitor extends Visitor {
  int total = 1;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  } // forward to sub-trees

  @Override
  void visit(Leaf leaf) {
    total *= leaf.value;
} }
```

ProdVisitor.java

```
class VisitorWorld {

  public static void main (String[] args) {
    Tree l1 = new Leaf(1);
    Tree l2 = new Leaf(2);
    Tree l3 = new Leaf(3);
    Tree n12 = new Fork(l1, l2);
    Tree tree = new Fork(n12, l3);
    SumVisitor sumV = new SumVisitor();
    ProdVisitor prodV = new ProdVisitor();
    tree.accept(sumV);
    tree.accept(prodV);
    System.out.println( “sum: " +
                        sumV.total +
                        " prod: “ +
                        prodV.total );
} }
```

Operations do not need to know how the data structure is actually implemented, or how it is traversed; they only need to know how to visit its components.

Data Structures do not need to know anything about the operations that are performed over them. They could be developed independently.

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

Leaf.java

```
class Fork extends Tree {

    v.visi
} }
```

Data Structures

```
abstract class Tree {
abstract void accept(Visitor v);
}
```

Tree.java

code extended from N. Wu

The Visitor Pattern

‘Visitor Pattern’ Emerges

Visitor.java

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

VisitorWorld.java

```
class VisitorWorld {
  ...
} }
```

Tree.java

```
abstract class Tree { // the client
  abstract void accept(Visitor v);
}
```

‘user client’ knows both the tree and the visitor

SumVisitor.java

```
class SumVisitor extends Visitor {
  ...
  @Override
  void visit(Fork fork) {
    ...
  }

  @Override
  void visit(Leaf leaf) {
    ...
} }
```

The Visitor Pattern represents an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Leaf.java

```
class Leaf extends Tree {
  ...
  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

Fork.java

```
class Fork extends Tree {
  ...
  @Override
  void accept(Visitor v) {
    v.visit(this);
} }
```

ProdVisitor.java

```
class ProdVisitor extends Visitor {
  ...
  @Override
  void visit(Fork fork) {
    ...
  }

  @Override
  void visit(Leaf leaf) {
    ...
} }
```

various different non-abstract visitors possible, all inherit from Visitor

various different non-abstract types of Tree may exist, all of which must implement the accept method

Useful Java Features

Variadic Arguments

- in Java, methods can have a **variable number of arguments** (thus, the method has indefinite arity)
- these **variadic** methods can be made to accept zero or more arguments of a given type using the **...** notation
- the arguments are provided to the methods as an **array**
- this is very useful for passing dynamically structured data into methods

```java
class Robot {
  ...
  void talk(String phrase) {
    if (powerLevel >= 1.0f) {
      System.out.println(name + " says " + phrase);
      powerLevel -= 1.0f;
    } else {
      System.out.println(name + " is too weak to talk.");
  } }

  void talk(String first, String... strings) {
    this.talk(first);
    for(String string : strings) {
      this.talk(string);
  } }

  void charge(float amount) {
    System.out.println(name + " charges.");
    powerLevel = powerLevel + amount;
} }
```

method takes a single argument of type String

overloaded method takes variable number of arguments of type String

```java
class VariadicRobotWorld {

  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    c3po.charge(10);
    c3po.talk("'A single hello, Java!'");
    c3po.talk("'Hello again, Java.'",
              "'Hey again!'",
              "'Still talking!'");
} }
```

method 'talk' can now be called with one or any number of String arguments

Enumeration Classes

- if you define a class using **enum** instead of **class**, the first statement must be a fixed list of **constants** of that class
- constants are the <u>only</u> objects (and can be used via **Side.NOUGHT** etc), but are guaranteed never to be duplicated, so you can use **==** for direct comparison
- constants are handled as auto-instantiated objects, you can reference them, even use a constructor for their initialisation

```
public enum Side {
  NOUGHT("O"), CROSS("X");

  String symbol;

  Side(String symbol) {
    this.symbol = symbol;
  }

  public String symbol() {
    return symbol;
  }

  public Side other() {
    return this == NOUGHT ? CROSS : NOUGHT;
} }
```

enumeration of the constants with calls to the constructor

comparison using == is possible

```
class SideWorld {

  public static void main (String[] args) {
    Side sideA = Side.NOUGHT;
    Side sideB = Side.CROSS;
    Side sideC = Side.CROSS;
    System.out.println(sideA==sideB); //false
    System.out.println(sideC==sideB); //true
    System.out.println(sideA.symbol()); //O
    System.out.println(sideB.symbol()); //X
    System.out.println(sideA.other().symbol()); //X
} }
```

enums are just objects

To Do


- recap content and check out the unit website
- write, compile, run and understand all the tiny programs from the lectures so far

Use the forum, we are there for you!