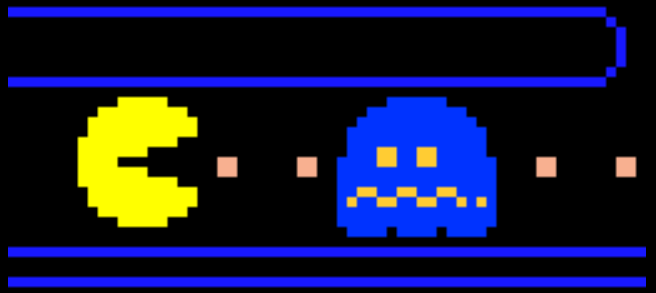
Department of Computer Science
University of Bristol

COMSM0086 – Object-Oriented Programming

# POLYMORPHISM AND DOUBLE DISPATCH

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

# double Dispatch




READY!
BROTHER
BRAIN

# What is double and / or multiple dispatch?

Dispatch based on two or more types

- Single dispatch:
  - `mammal.makeNoise();`
    - Resolves mammal to its underlying type and calls its makeNoise method rather than mammal's
- Double dispatch:
  - `(mammal1 & mammal2).makeNoise();`
    - Depends on which mammals are interacting!
- Why might you do this?

# What is multiple dispatch?

Dispatch based on two or more types

- Single dispatch:
  - `mammal.makeNoise();`
    - Resolves mammal to its underlying type and calls its makeNoise function rather than mammal's
- Double dispatch:
  - `(mammal1 & mammal2).makeNoise();`
    - Depends on which mammals are interacting!
- Why might you do this?
  - **Collision detection in games (for example)**

# Multiple Dispatch – the Big Switch

```
7  #define rock 1
8  #define paper 2
9  #define scissors 3
10
11 typedef struct entity {
12     unsigned int ID;
13     void * data;
14 } Entity;
15
16 void collision(Entity * e1, Entity * e2)
17 {
18     switch (e1->ID) {
19     case rock:
20         switch (e2->ID) {
21         case rock: /*DRAW*/
22         case paper: /*ENTITY 2 WINS*/
23         case scissors: /*ENTITY 1 WINS*/}
24     case paper:
25         switch (e2->ID) {
26         case rock: /*ENTITY 1 WINS*/
27         case paper: /*DRAW*/
28         case scissors: /*ENTITY 2 WINS*/}
29     case scissors:
30         switch (e2->ID) {
31         case rock: /*ENTITY 2 WINS*/
32         case paper: /*ENTITY 1 WINS*/
33         case scissors: /*DRAW*/}}
34 }
```

# Criticisms of the Big Switch

- Good
  - Code is relatively easy to step through
- Bad
  - Adding/removing types is a big job which will not be verified by a compiler
  - Need a type field for all types and have to keep track of all types
  - Switch statements tend to grow and show up as code replication throughout the project

# Multiple Dispatch – Function pointer / dispatch table

```c
#define rock 0
#define paper 1
#define scissors 2

typedef struct entity {
    unsigned int ID;
    void * data;
} Entity;

typedef void(*collReso)(Entity * e1, Entity * e2);

void rock_rock(Entity * e1, Entity * e2);
void rock_paper(Entity * e1, Entity * e2);
void rock_scissors(Entity * e1, Entity * e2);
void paper_rock(Entity * e1, Entity * e2);
void paper_paper(Entity * e1, Entity * e2);
void paper_scissors(Entity * e1, Entity * e2);
void scissors_rock(Entity * e1, Entity * e2);
void scissors_paper(Entity * e1, Entity * e2);
void scissors_scissors(Entity * e1, Entity * e2);

void collision(Entity * e1,Entity * e2,collReso colTbl[][3])
{
    collReso cr = colTbl[e1->ID][e2->ID];
    cr(e1, e2);
}
```

```c
int main()
{
    Entity e1, e2;
    collReso colTbl[3][3];

    colTbl[0][0] = rock_rock;
    colTbl[0][1] = rock_paper;
    colTbl[0][2] = rock_scissors;
    colTbl[1][0] = paper_rock;
    colTbl[1][1] = paper_paper;
    colTbl[1][2] = paper_scissors;
    colTbl[2][0] = scissors_rock;
    colTbl[2][1] = scissors_paper;
    colTbl[2][2] = scissors_scissors;

    e1.ID = 0;
    e2.ID = 2;

    collision(&e1, &e2, colTbl);

    return 0;
}
```

# Criticisms of Function table

- Good
  - Tables are disentangled from code.
  - Table modification has less side-effects with regards to the code which uses them (in contrast to the switch statement approach).
- Bad
  - Debugging / stepping through is awful
  - Still require type field for all types

# How can we do this in Java?

You might be tempted to do something like this (**it won't work**):

```
public class Runner {
    public static void main (String [] args){
        Mammal mDolphin = new Dolphin();
        Mammal mLion = new Lion();

        mDolphin.makeNoise(mLion);
    }
}
```

And then update the mammal class and its children to provide overloaded functions for each scenario …

# How can we do this in Java?

```
public abstract class Mammal {

    public void stateAttributes(){
        System.out.println("Warm blood, 3 inner
    }
    public abstract void makeNoise();

    public abstract void makeNoise(Dog d);
    public abstract void makeNoise(Lion l);
    public abstract void makeNoise(Dolphin d);
}

public class Dolphin extends Mammal{
    @Override
    public void makeNoise() {
        System.out.println("squeek click");
    }
    @Override
    public void makeNoise(Dog d) {
        System.out.println("Dolphin interacting with dog");
    }
    @Override
    public void makeNoise(Dolphin d) {
        System.out.println("Dolphin interacting with dolphin");
    }
    @Override
    public void makeNoise(Lion l) {
        System.out.println("Dolphin interacting with lion");
    }
}
```

But it wont work because each function is expecting a particular type of mammal: **call parameters, even if they are references, are treated as static …**

# Double dispatch in Java

Mammals and their children (done)

Rock paper scissors (done)

# Criticisms of OO Approach

- Good
  - Many errors caught at compile time
  - Avoids the issues inherent to switch-based solutions
- Bad
  - Hard to understand - especially for those unfamiliar with Visitor design pattern
  - It is an object oriented solution which violates many object oriented ideas
    - breaks encapsulation - function call interacts with more than one type

# Multiple Dispatch (Robot example)

- 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 **recursively**
- 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

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);
} }
```

different options are selected during runtime

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