Programming 1 / Object Oriented Programming

Introduction to Java
Lecture #4: "Madness in the methods" parameters, return values and constructors

The story so far

- In our classes, we've had
  - Attributes that have a **visibility** of **public** to store state and identity for objects
  - Methods for behaviour
    - so far these have simply printed things to the console (possibly after doing some calculations)

The problem with just printing a method's result

- Consider a version of Ball class
  - This version has a bounce method that determines that the height of the bounce is diameter times two:
  - result would be to print `13` to the console

```
public class Ball
{
  public double diameter;

  public void bounce()
  {
    double height = this.diameter * 2;
    System.out.println(height);
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.bounce();
  }
}
```

The problem with just printing a method's result

- What about if we wanted to *use* that result in some way?
  - What about if we wanted to calculate whether the bounced height was high enough to go over another object
    - …perhaps a goal in football, or a net in tennis?

The problem with just printing a method's result

```
public class Ball
{
  public double diameter;

  public void bounce()
  {
    double height = this.diameter * 2;
    System.out.println(height);
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.bounce();
    if (the tennisBall bounces heigher
        than a net..)
    {
       System.out.println("15-love!");
    }
    else
    {
       System.out.println("love-15!");
    }
  }
}
```

The problem with just printing a method's result

```
public class Ball
{
  public double diameter;

  public void bounce()
  {
    double height = this.diameter * 2;
    System.out.println( height );
  }
}
```

How do we get this…

…so we can use it here?

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.bounce();
    if (the tennisBall bounces heigher
        than a net..)
    {
       System.out.println("15-love!");
    }
    else
    {
       System.out.println("love-15!");
    }
  }
}
```

The concept of a return value

- A **return value** lets us send a result back from a method to whatever originally called it
- **THIS IS DIFFERENT FROM JUST PRINTING THE RESULT FROM WITHIN THE METHOD!**

The concept of a return value – a real-world illustration

- Consider if I asked you "what's two plus two"?
  - Printing a result (System.out.println)
    - is like telling someone the answer verbally
    - once the air has stopped vibrating, the answer is gone
  - Returning a result as a return value
    - is like writing down the answer and handing someone the piece of paper
    - The person receiving the piece of paper might read it out loud ("print" it!), or they might write down on another piece of paper (put it into a variable) or they might do something else with it

Using a return value with our balls

```
public class Ball
{
  public double diameter;


  public void bounce()
  {
    double height = this.diameter * 2;
    System.out.println(height);

  }
}
```

Using a return value with our balls

```
public class Ball
{
  public double diameter;

        double
  public void bounce()
  {
    double height = this.diameter * 2;
    System.out.println(height);
    return height;
  }
}
```

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

The **return type** is specified before the method name

The **return statement** is when you "transmit" the result of your method back to the calling statement

The **return value** is what gets "transmitted" back to the calling statement

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    double bounced = tennisBall.bounce();
    if (bounced > 36)
    {
       System.out.println("15-love!");
    }
    else
    {
       System.out.println("love-15!");
    }
  }
}
```

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    double bounced = tennisBall.bounce();
    if (bounced > 36)
    {
       System.out.println("15-love!");
    }
    else
    {
       System.out.println("love-15!");
    }
  }
}
```

Call our method

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height ;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    double bounced = tennisBall.bounce();
    if (bounced > 36)
    {
      System.out.println("15-love!");
    }
    else
    {
      System.out.println("love-15!");
    }
  }
}
```

Assign whatever gets **returned** to a newly declared variable, `bounced`

Call our method

Using a return value with our balls

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    double bounced = tennisBall.bounce();
    if (bounced > 36)
    {
      System.out.println("15-love!");
    }
    else
    {
      System.out.println("love-15!");
    }
  }
}
```

We can make use of `bounced` in the usual ways

Remember your grammar!

- A call to a method will evaluate to whatever its return value is…
- …so the method call will "fit" in your code anywhere where its returned data type would

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    if (tennisBall.bounce() > 36)
    {
       System.out.println("15-love!");
    }
    else
    {
       System.out.println("love-15!");
    }
  }
}
```

- This is *grammatically* no different than doing

  if (*13.0 > 36*)

Passing parameters to methods

- "Transmission" need not only be one way (i.e. from method to caller)
- You can pass **parameters** to your methods as well as get a value **returned** from them
- **THIS IS VERY DIFFERENT FROM GETTING INPUT FROM THE KEYBOARD!**

Passing parameters to methods

- Consider the earlier example when I asked someone to add two plus two
- I "called a method"… i.e. "add some numbers"
- …but I also passed two parameters to the "method"
  - what were they?

Passing parameters to methods

- Implementing a Calculator class
  - …one of the methods such a class might have is `add`
  - …so if we were to write `add` for 2 + 2 we might do the opposite:
  - But what about scenarios other than 2 + 2?

```
public class Calculator
{
  public double add()
  {
     double num1 = 2;
     double num2 = 2;
     return num1+num2;
  }
}
```

Passing parameters to methods

- We can specify **parameters** within the brackets that follow the method name (which become part of the **method signature**)

```
public class Calculator
{
  public double add(double num1, double num2)
  {
     return num1+num2;
  }
}
```

Passing parameters to methods

- We can specify **parameters** within the brackets that follow the method name (which become part of the **method signature**)

```
public class Calculator
{
  public double add( double num1 , double num2 )
  {
     return num1+num2;
  }
}
```

Parameters

Passing parameters to methods

```
public class Calculator
{
  public double add(double num1, double num2)
  {
     return num1+num2;
  }
}

public class Main
{
  public static void main(String[] args)
  {
     Calculator casio = new Calculator();
     double result1 = casio.add(2,2);
     double result2 = casio.add(6, result1);
     System.out.println("The first result is "+result1);
     System.out.println("The second result is "+result2);
  }
}
```

Passing parameters to methods

```
public class Calculator
{
  public double add(double num1 , double num2 )
  {
     return num1+num2;
  }
}

public class Main
{
  public static void main(String[] args)
  {
     Calculator casio = new Calculator();
     double result1 = casio.add( 2 , 2 ;
     double result2 = casio.add(6, result1);
     System.out.println("The first result is "+result1);
     System.out.println("The second result is "+result2);
  }
}
```

?

Things to consider/remember about method parameters

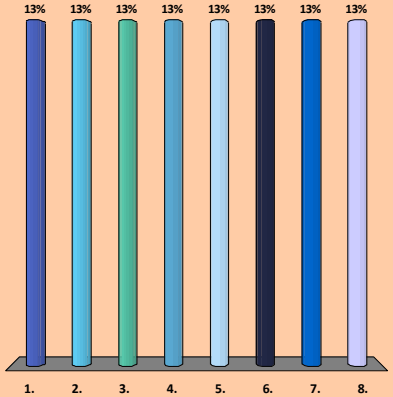
- Each and every parameter MUST have a data type, *even* if they're all the same type
  - `public double add(double num1, double num2)` ✅
    - and NOT
  - `public double add(double num1, num2)` ❌
- You can think of parameters being like variables that exist for the duration of the method, but that can have values assigned to them from the calling code

What would be the output of the following project?

1. 2
2. 3
3. 4
4. 5
5. 9
6. 10
7. 13
8. There would be an error

```
public class Wibble
{
  public int wobble;
  public int flibble;

  public void niblick(int foo)
  {
    this.wobble = foo*2;
  }

  public void bibble(int bar)
  {
    this.flibble = bar * 3;
  }

  public int flange()
  {
    return wobble + flibble;
  }
}
```

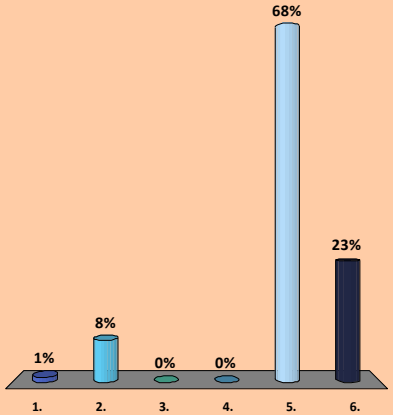
```
public class Klunge
{
  public static void main
              (String[] args)
  {
    Wibble x = new Wibble();
    x.niblick(2);
    x.bibble(3);
    int bar = x.flange();
    System.out.println(bar);
  }

}
```



What would this code print?

1. 30
2. 32
3. 2
4. 0
5. It wouldn't *print* anything!
6. It wouldn't even run; there'd be an error

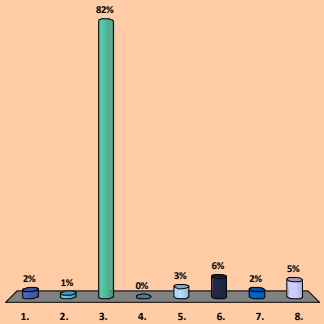
```
public class Wibble
{
  public void add(int x, int y)
  {
    return (x+1+y+1);
  }
}

public class Klunge
{
  public static void main
              (String[] args)
  {
    Wibble x = new Wibble();
    int y = x.add(10,20);
  }
}
```



OK, how would you fix it then so that it at least *runs*?

1. Remove the word **void** in line 3 of **Wibble**
2. Add the word **int** before **void** in line 3 of **Wibble**
3. Replace the word **void** with the word **int** in line 3 of **Wibble**
4. Remove the word **void** in line 3 of **Klunge**
5. Add the word **int** before **void** in line 3 of **Klunge**
6. Replace the word **void** with the word **int** in line 3 of **Klunge**
7. Add a line towards the end of **Klunge** to print out the value of **y**
8. Something else

```
public class Wibble
{
  public void add(int x, int y)
  {
    return (x+1+y+1);
  }
}

public class Klunge
{
  public static void main
                (String[] args)
  {
    Wibble x = new Wibble();
    int y = x.add(10,20);
  }
}
```



Constructors

- A constructor is a special kind of method
- It has no return type
- It is called when you create a new instance of a class
- ALL Java classes have a constructor, even if you don't declare one…
- …but you can declare constructors of your own to replace the default one

Constructors

- When we create a new instance of a class, for example

```
Ball tennisBall = new Ball();
```

- we are in fact calling the **constructor** of the class

Constructors and our balls

```
public class Ball
{
  public double diameter;

  public Ball()
  {
     System.out.println("We have a new ball");
  }

  // getters, setters, bounce and
  // roll would follow here…
}
```

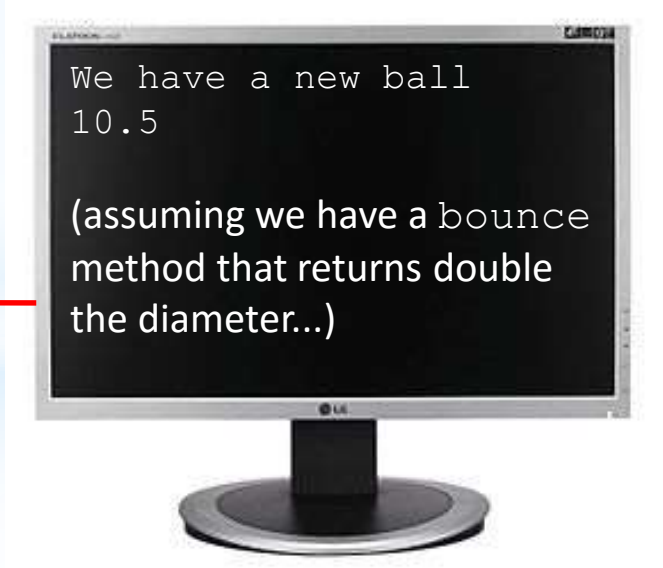
```
public class Main
{
  public static void main(String[] args)
  {
    Ball snookerBall = new Ball();
    snookerBall.setDiameter(5.25);
    System.out.println(snookerBall.bounce());
  }
}
```

What would the main method print?

Constructors and our balls

```
public class Ball
{
  public double diameter;

  public Ball()
  {
     System.out.println("We have a new ball");
  }

  // getters, setters, bounce and
  // roll would follow here…
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball snookerBall = new Ball();
    snookerBall.setDiameter(5.25);
    System.out.println(snookerBall.bounce());
  }
}
```



Constructors and our balls

```
public class Ball
{
  public double diameter;

  public Ball()
  {
     System.out.println("We have a new ball");
  }

  // getters, setters, bounce and
  // roll would follow here…
}

public class Main
{
  public static void main(String[] args)
  {
    Ball snookerBall = new Ball();
    snookerBall.setDiameter(5.25);
    System.out.println(snookerBall.bounce());
  }
}
```

Creating a new Ball calls the constructor

Using constructors to set default values

- We might use a constructor to set default values for a new object
- Say we wanted each new ball to default to
  - Colour: white
  - Diameter: 5

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball()
  {
     this.diameter = 5;
  }

  // getters, setters, bounce and
  // roll would follow here…
  ...
}
```

Using constructors to set default values

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball()
  {
     this.diameter = 5;
     this.colour = "white";
  }

  public double getDiameter()
  {
     return this.diameter;
  }

  // other getters, setters, bounce and
  // roll would follow here…
```

```
public class main
{
   public static void main (String[] a)
   {
      Ball newBall = new Ball();
      double d = newBall.getDiameter();
   }
}
```

What will go into `d`?

Using constructors to set default values

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball()
  {
     this.diameter = 5;
     this.colour = "white";
  }

  public double getDiameter()
  {
     return this.diameter;
  }

  // other getters, setters, bounce and
  // roll would follow here…
```

```
public class main
{
   public static void main (String[] a)
   {
      Ball newBall = new Ball();
      double d = newBall.getDiameter();
   }
}
```

What will go into d?

means the constructor method is run…

diameter is set to 5

what will getDiameter give us?

Parameterised constructors

- A constructor is a (special kind of) method
- Methods can be written with parameters
- Therefore constructors may have parameters
- We can use such constructors to create both a new object, and set its attributes in a single go

Parameterised constructors

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  ...
```

```
public class main
{
   public static void main (String[] a)
   {
      Ball newBall = new Ball(10,"blue");
      double d = newBall.getDiameter();
   }
}
```

...NB: not complete!!!!

....rest of class would follow,

e.g. bounce etc

Parameterised constructors

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  ...
```

...NB: this class is abridged (i.e. not complete for space reasons…!)

….rest of class would follow, e.g. getters, setters, bounce etc

```
public class main
{
   public static void main (String[] a)
   {
      Ball newBall = new Ball(10,"blue");
      double diam = newBall.getDiameter();
      String colour = newBall.getColour();
   }
}
```

What would be in `diam`?

Would would be in `colour`?

Caveat emptor (buyer, beware)

- The moment you supply your own constructor, the default one no longer exists!
- If you have a constructor with the signature
  - public Ball(double d , double c)
- you will no longer be able to create a new Ball using
  - Ball b = new Ball();

Fortunately… (multiple methods with the same name)

- You CAN have more than one method declaration with the same name in the same class
- This includes constructors (which are just a special type of method)
- This will work as long as the **method signatures** are different

Multiple constructors

```
public class Ball
{
  public double diameter;
  public String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  public Ball(double d)
  {
     this.diameter = d;
     // default value for colour
     this.colour = "white";
  }

  public Ball(String c)
  {
     this.colour = c;
     // default value for diameter
     this.diameter = 5;
  } ...
```

The method signatures for the constructors are DIFFERENT when you include the parameters as WELL as the name of the method

You can do this with ALL methods – not just constructors

This is an example of **method overloading** and can also be referred to as **static polymorphism**

Multiple methods with same name

```
public class Calculator
{
   public double add(double num1, double num2)
   {
     return num1+num2;
   }

   public double add(double num1, double num2, double num3)
   {
     return num1+num2+num3;
   }
}

public class Main
{
   public static void main(String[] args)
   {
      Calculator calc = new Calculator();
      double result = calc.add(2.2,3.1);
   }
}
```

Multiple methods with same name

```
public class Calculator
{
   public double add(double num1 , double num2 )
   {
     return num1+num2;
   }

   public double add(double num1, double num2, double num3)
   {
     return num1+num2+num3;
   }
}

public class Main
{
   public static void main(String[] args)
   {
      double result = add( 2.2 , 3.1 );
   }
}
```

The version of add that is used is determined by the method signature…

…two doubles supplied as parameters, therefore the version that runs is the one that TAKES two doubles (as opposed to the one with 3)

Summary

- Methods can be given **parameters** from and **return values** to the code that calls them
- A **constructor** is a special kind of method
- Constructors run whenever an instance of a class is created
  - (i.e. whenever `new MyClass()` is done)
- All classes have a constructor even if you don't declare one
  - If you don't, a **default constructor** is created by Java behind the scenes – it will have no parameters and does nothing*

* this is not *entirely* accurate, but is an acceptable simplification for now

Summary

- Constructors have no return type but can accept parameters
- If you declare a constructor, the default one is no longer created by Java
- If you declare a constructor with parameters, you will no longer be able to create instances of your objects with a parameterless instantiation e.g. `new MyClass()`

Summary

- Methods (including constructors) can be declared with the same name as long as the **method signatures** differ
  - i.e. they must accept a different combination of parameters
- This is called **method overloading** or **static polymorphism**
- If you create a parameterised constructor, you can use this to add a non-parameterised one
  - …which would restore the ability to do `new MyClass()`