Classes, enums and packages

This lecture will

- Introduce the `null` keyword and another use for the `this` keyword
- Revisit enumerations
- Examine the use of local variables within methods
- Explain how to document and write a **test harness** for a Java class
- Introduce **packages**

The `null` keyword

- So far we have combined declaration with object creation as follows:

```
Meal meal = new Meal();
Meal chips =
       new Meal("Chips",3.99,350,Diet.VEGAN);
```

- Actually, we don't have to create an object when the variable is declared:

```
Meal chips;
```

because it already has the value `null`

The `null` keyword

- We can emphasize that chips does not contain a valid reference by explicitly assigning it the special value `null`

```
Meal chips = null;
```

Testing for `null`

- The keyword null can only be used for references to objects – we cannot say

```
int i = null;
```

WRONG

- We can test for a `null` value in an expression:

```
if (  chips != null   ) {
  // do something with chips
}
```

```
if (  chips == null   ) chips = new Meal();
```

Assignments and declarations revisited

```
Meal chips = null;
```

This is both a **declaration** and an **assignment**

- We could subsequently create an object and store its reference in the variable by calling a constructor:

```
chips = new Meal();
```

This is an **assignment** but not a **declaration**

Enumerations and Classes

- Up to now we have been treating enumerations as a type but they are really a special kind of class
- An **`enum`** is a class where every instance that will ever be created is created when the class itself is created and can never be changed
- Because they are a class an **`enum`** variable can be assigned the value **`null`**

```
diet = null;
```

Enumerations and Classes

Since an **`enum`** is a class where everything is declared at the start

- You never need to use **`new`**
- The values like **`OMNIVOUROUS`**, **`VEGAN`** etc. are in capitals because they are constants, they can't be changed
- **`Diet`** should really be declared by itself in a file called **`Diet.java`** like any other class

Enumeration types in Classes

- We have a file called **`Diet.java`** in the same directory as **`Meal.java`** which contains

```
public enum Diet {
  OMNIVOUROUS, VEGAN, VEGETARIAN, UNSPECIFIED
}
```

- We don't need to compile it. If **`Meal`** uses **`Diet`** typing

```
U:\myjava>javac Meal.java
```

will automatically compile **`Diet`**

Compile time errors may not come from the file you are trying to compile

Interacting classes

- When **`Meal`** is executed any methods in **`Diet`** it calls will be executed
- Java sorts this out for you, you don't need to tell it to do anything as long as **`Diet.java`** is in the same directory as **`Meal.java`**
- This applies to any Java class, not just **`enum`**s; if you have two or more public classes in the same directory they can run each others methods without any further declaration

Enumerations and `null`

- Our default constructor was

```
public Meal() {
    this("Unknown", DEFAULT_PRICE,
        DEFAULT_CALORIES, Diet.UNSPECIFIED);
}
```

but diet is an enum – which is a class – so we don't need **`Diet.UNSPECIFIED`** because we can use **`null`**

```
public Meal() {
    this("Unknown", DEFAULT_PRICE,
        DEFAULT_CALORIES, null);
}
```

`toString` for `Meal` with `Diet`

```
public String toString() {
   String d = " which is suitable for a "+diet+" diet";
   return "Meal Name=" + name +  ", Price=" + price +
     ", Calories=" + calories + d.toLowerCase();
}
```

```
Meal pizza = new Meal("Special Pizza", 8.99, 800,
                                    Diet.OMNIVOROUS);
System.out.println(pizza);
```

```
Special Pizza, Price=8.99, Calories=800 which is
                        suitable for a omnivorous diet
```

A `toString` method for enumerations

- Classes should have a **`toString`** method
- **`enum`**s are a kind of class
- **`enum`**s can have their own **`toString`** method

A `toString` method for enumerations

You only ever need the method **`name`** inside **`toString`**. Elsewhere use **`toString`**

Note semicolon

```
public enum Diet {
   OMNIVOUROUS, VEGAN, VEGETARIAN;

   public String toString() {
       return
          "a "+
          name().toLowerCase() +
          " diet";
   }
}
```

A toString method for enumerations

```
public enum Diet {
   OMNIVOUROUS, VEGAN, VEGETARIAN;

   public String toString() {
      String indefiniteArticle = "a";
      if  (  this == OMNIVOROUS  )
          indefiniteArticle+="n";
      return
         indefiniteArticle + " " +
         name().toLowerCase() +
         " diet";
   }
}
```

A `toString` method for enumerations

```
public enum Diet {
   OMNIVOUROUS, VEGAN, VEGETARIAN;

   public String toString() {
      String indefiniteArticle;
      switch( name().charAt(0)){
         case'A': case'E': case'I':
         case'O': case'U':
            indefiniteArticle = "an";
            break;
         default : indefiniteArticle = "a";
      }
      return
         indefiniteArticle + " " +
         name().toLowerCase() + " diet";
   }
}
```

Printing enums again

Run time error if **`diet`** is set to **`null`**

```
public String toString() {
     return name + ", Price=" + price +
        ", Calories=" + calories +
         " which is suitable for "+diet;
}
```

```
Dinner pizza =
   new Dinner("Special Pizza", 8.99, 800,
                                        Diet.OMNIVOROUS);
System.out.println(pizza);
```

```
Special Pizza, Price=8.99, Calories=800 which is
                        suitable for an omnivorous diet
```

Printing enums again

```
public String toString() {
     String priceAndCalories = ", Price=" + price +
               ", Calories=" + calories;
     if  (  diet == null  )
          return name+priceAndCalories;
     else
          return name+priceAndCalories+
               " which is suitable for "+diet;
}
```

```
Meal pizza = new Meal("Special Pizza", 8.99, 800,
                                        Diet.VEGAN);
System.out.println(pizza);
```

```
Special Pizza, Price=8.99, Calories=800 which is
                           suitable for a vegan diet
```

Reading in enum values

- Remember
  - **`Integer.valueOf("12345")`** turns its String parameter into the integer 12345
  - **`Diet.valueOf("VEGAN")`** turns its String parameter into **`Diet.VEGAN`**

Reading in enum values

```
m.setDiet(Diet.valueOf(
  keyboard.readString().toUpperCase()));
```

- This works with Omnivorous, vegan and VEGETARIAN
- But not ovo-lacto vegetarian

Creating enums

```
public enum Diet {
     OMNIVOROUS, VEGAN, VEGETARIAN;
     public String toString() {…}
     public static Diet called(String s) {
       if  (  s != null  )
          switch (s.toUpperCase())  {
               case "OMNIVOROUS" : return OMNIVOROUS;
               case "VEGAN" : return VEGAN;
               case "VEGETARIAN" : return VEGETARIAN;
               default : return null;
          }
       return null;
     }
}
```

Notice multiple **`return`** statements and no **`break`** statements

`String`s can be **`null`** too

No need for the Diet prefix within the **`enum`**

```
m.setDiet(Diet.called(keyboard.readString()));
```

Methods

- Everything inside a Java program happens in one method or another
- The standard methods for any class
  - Get methods
  - Set methods
  - Constructors
  - The **`toString`** method
  - The **`main`** method

are just the beginning

Methods

- Take information in in the form of parameters
- Export information from the **`return`** statement or statements
- The **`return`** statement can return an expression not just the value of a variable
- The formal parameters are always declared as variables but the actual parameters can be expressions, it is their value that is important

Returning an expression

```
public class Circle {

  private double x,y;     // the circle centre
  private double radius;  // the radius

  public String toString() { …… }
  public Circle(double x, double y, double r) { …… }
  ……


  public double circumference() {
       return 2 * Math.PI * radius;
  }

  public double area() {
       return Math.PI * radius * radius;
  }
}
```

Parameter Passing

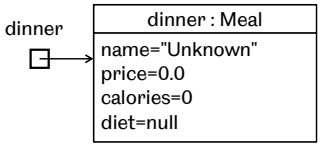
- Parameters are passed into a Java methods **by value**
- The actual parameters can be expressions which are evaluated before execution
- Even if the actual parameter is a variable only the value of the variable is assigned to the formal parameter before the method is executed
- Actual parameters are never changed

Call by value

- Consider the following

```
Meal dinner = new Meal();
```

- The data structure is

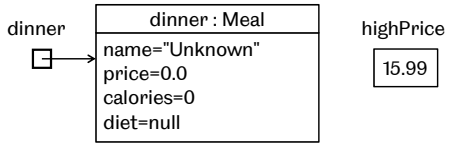


Call by value

- Consider the following

```
Meal dinner = new Meal();
double highPrice = 15.99;
```

- The data structures are

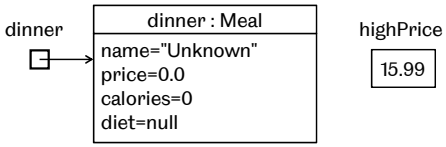


Call by value

- Consider the following

```
Meal dinner = new Meal();
double highPrice = 15.99;
dinner.setPrice(highPrice);
```

- The data structures are



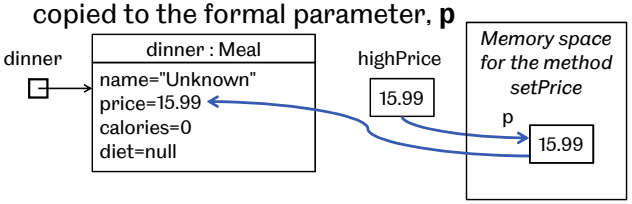

```
public void setPrice(double p){  price = p;  }
```

Call-by-value continued

```
dinner.setPrice(highPrice);
```

```
public void setPrice(double p){  price = p;  }
```

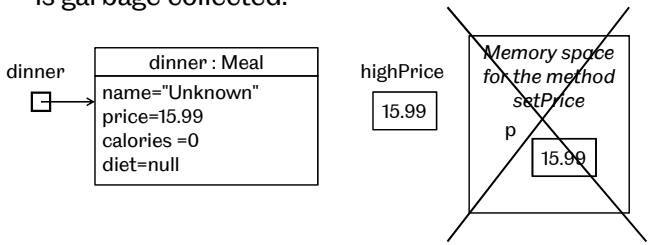
- First the value of the actual parameter is copied to the formal parameter, **p**



- Then the value of **p** is assigned to price

Call-by-value continued

- The variable **p** is only accessible within the **`setPrice`** method; it is a **local variable**
- After the **`setPrice`** method has executed, **p** is garbage collected.



Changing the formal parameter

- In call-by-value, modifying the formal parameter does not change the actual parameter
- The formal parameter **p** is local to the method and garbage collected afterwards
- So is anything declared in the method

```
public void setPrice(double p) {
  double markUp = 0.1;
  p += p*markUp;
  price = p;
}
```

Changing the formal parameter

```
dinner.setPrice(highPrice)
```

```
public void setPrice(double p) {
  double markUp = 0.1;
  p += p*markUp;
  price = p;
}
```

Local variables within methods

- The space for formal parameters is created when the method is run and garbage collected immediately afterwards
- As is the space for anything declared within a method – all variables declared within a method are **local variables** and exist only whilst the method is running
- Changing the formal parameter within a method has no effect on the actual parameter

Choosing parameter names

- Consider the **`setRadius`** method of the **`Circle`** class:

```
public class Circle {
     private double xCentre, yCentre;   // the centre
     private double radius;             // the radius

     public void setRadius(double r) {
          radius = r;
     }
```

- We might think it is more readable to call the formal parameter **`radius`**, rather than **`r`**:

```
     public void setRadius(double radius) {
          radius = radius;
     }
```

The `this` keyword

- We can solve this problem by using the keyword **`this`**:

```
   public void setRadius(double radius) {
      this.radius = radius;
   }
```

- The keyword **`this`** indicates that the instance variable is being referred to, not the formal parameter
- Hence, as in chained constructors, **`this`** means **the current instance of the class**
- Some programmers use this notation all the time

Is this a correct declaration of `setRadius`?

```
public void setRadius(double r) {
     this.radius = r;
}
```

✓1. Yes
2. No

Scope and visibility

- Every variable in Java has a **scope**, which determines where it can be accessed during compilation
- Every variable in Java has space in memory which is created when necessary at run time and destroyed once that variable goes out of scope

Scope and visibility

- Java uses **scope rules**, the most important of which are:
  - Formal parameters can be used throughout a method declaration
  - Variables declared within a method or compound statement have **local scope**. They are usable from the line they are declared until the closing bracket of the method or compound statement
  - Instance and static variables of objects have **global scope**. They are in scope throughout the class declaration

Scope and memory

- Space is created for
  - formal parameters whenever their method is executed and destroyed as soon as execution ends
  - variables with local scope when their declaration is executed and destroyed when the variable goes out of scope
  - instance variables whenever an object is created and it lasts as long as the object
  - static variables on compilation and it is not destroyed until execution is complete

Scope and visibility – example

```
public class Something {
   public void methodOne() {
      int x;
      x = 1;
   }
   public void methodTwo() {
      int y;
      y = x+1;
   }
}
```

Causes a compilation error

- We get an error when this program is compiled because the scope of **x** is limited to **methodOne**.

```
Something.java:8: cannot resolve symbol
symbol  : variable x
```

A hole in the scope of a variable

- What happens if a local variable declared in a method has the same name as an instance variable?

```
public class Something2 {
   private int x = 2;
   public void methodOne() {
      int x=1;
      System.out.println("methodOne x = " + x);
   }
   public void methodTwo() {
      System.out.println("methodTwo x = " + x);
   }
}
```

Don't do this

Exercise

```
public class Something3 {
   private int x = 2;
   private double y = 3.0;
   public void methodA(int y) {
      int x=1;
      System.out.println("methodA x and y " + x + ", " + y);
   }
   public void methodB(int x) {
      methodA(this.x);
      System.out.println("methodB x and y " + x + ", " + y);
   }
   public void methodC() {
      methodA(x);
      System.out.println("methodC x and y " + x + ", " + y);
   }
   public void methodD(int x) {
      this.x = this.x+x;
      methodA(x);
      System.out.println("methodD x and y " + x + ", " + y);
   }
```

Exercise and soluti

```
public class Something3 {
   private int x = 2;
   private double y = 3.0;
   public void methodA(int y) {
      int x=1;
      System.out.println("methodA x and y " + x + ", " + y);
   }
   public void methodB(int x) {
      methodA(this.x);
      System.out.println("methodB
   }
   public void methodC() {
      methodA(x);
      System.out.println("methodC
   }
   public void methodD(int x) {
      this.x = this.x+x;
      methodA(x);
      System.out.println("methodD x and y " + x + ", " + y);
   }
```

```
public static void
              main(String[] args) {
   Something3 s = new Something3();
   s.methodA(4);
   s.methodB(4);
   s.methodC();
   s.methodD(4);
}
```

```
methodA x and y 1, 4
methodA x and y 1, 2
methodB x and y 4, 3.0
methodA x and y 1, 2
methodC x and y 2, 3.0
methodA x and y 1, 4
methodD x and y 4, 3.0
```

The `main` Method

- When we started this course we wrote programs which were a class that contained nothing except a **`main`** method that did everything
- Any program we write will still need a **`main`** method as an entry point and won't work without it
- But now we have seen a class which uses another class and this is normally how object oriented programming works

Providing a test harness

- We can declare a **`main`** method for any class.
- This is useful for testing a class in isolation before integrating it into a larger program.
- Used in this way, the **`main`** method provides a **test harness** for a class.
- Note that the Java interpreter only runs the **`main`** method of the class that is invoked with the interpreter. Any other **`main`** methods are ignored.

Test harness for the `Circle` class

```
public class Circle {
  private double x,y;     // the circle centre
  private double radius;  // the radius

  public String toString() { …… }

  public Circle(double x, double y, double r) { …… }

  public void setRadius(double r) {   radius = r;   }

  public double circumference() { return 2.0*Math.PI*radius; }

  public double area() { return Math.PI * radius * radius; }

  public static void main (String[] args) {
        Circle c = new Circle(0.0, 0.0, 2.0);
        c.setRadius(10.0);
        System.out.println("Main method in class Circle");
        System.out.println("Circum: "+c.circumference());
        System.out.println("Area: " + c.area());
  }
}
```

The test harness

Packages and scope

- Java can find classes stored in the same directory as whatever it is compiling
- In practice Java classes can be arranged into packages, groups of related classes stored in the same directory
- For example the **`sheffield`** package; it is stored as a subdirectory so you have to tell Java to look for it

```
import sheffield.*;
```

Packages and scope

- Every class in the sheffield package has the following first line

```
package sheffield;
```

which tells the compiler that it belongs to the package

Making your own package

- To create a package, put all the source (`.java`) files in a directory with a meaningful name e.g. **`packagename`**
- Give each of them a first line which is

```
package packagename;
```

- Then from the **parent** directory of the package directory, compile all the source files:

```
U:\myjava>javac packagename\*.java
```

Using your own package

- Once you have compiled the package you can use all its public classes within any class in its parent directory which starts

```
import packagename.*;
```

- Classes don't have to be public, only the ones you want other classes to be able to use
- You can also have private classes

Access permission

- There are actually four levels of access permission:

public	(least restrictive)
protected	
default (package visibility)	↓
private	(most restrictive)

- We won't discuss the protected level yet

Access levels

- **public** – visible everywhere.
- **default** – visible within their own class and within other classes in the same package.
- **private** – only visible within their own class.
- Normally, classes and constants are public (they are intended for use by class and instance users) and attributes are private (the hidden state of an object).
- Methods can be either public or private

Documenting with `javadoc`

- Classes keep their workings private but are meant to be usable without that knowledge
- And not just by whoever wrote the class immediately after it was written
- Having written a class, we need to document it properly
- Java provides a tool called **`javadoc`**, which creates HTML documentation from special comments

Documenting with `javadoc`

- Documentation comments begin with **`/**`** and end with **`*/`**
- Formatting tags can be included within these symbols
- The most useful tags are:

```
@author
@param
@return
```

Documenting public classes

- You already know a class should be preceded by a comment that says what it does
- A documentation comment should start /** and end */
- Conventionally each line within the comment should start with an asterisk
- And it should contain `@author` followed by the author's name as well as information about what the class does

Documenting public methods

- Each public method should be directly preceded by a Javadoc comment that says what it does
- Method comments can also include **`@author`** tags if the class has multiple authors
- If it has parameters each should be described in order by a line that starts **`@param`** followed by its name and what it is there for
- If it returns anything it needs an **`@return`** to explain what it returns

Documenting `Meal`

```
/**
* Finds the price of the meal
* @return double the meal's price
*/
public double getPrice() {
   return price;
}
```

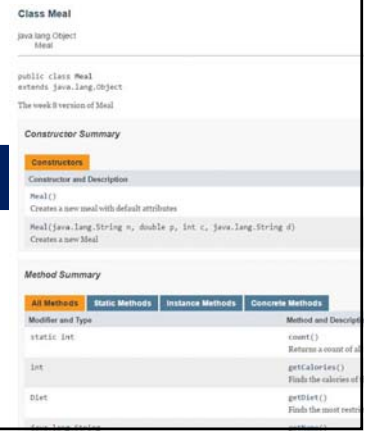
Documenting `Meal`

```
/**
* Changes the price of the meal by setting it to the
* the parameter's value.   The method fails if the
* new price is not reasonable
* @param p    the new price of the meal
*/
public void setPrice(double p) {
  if  (  validPrice(p)  )
     price = p;
  else {
     System.out.println("Bad price "+p+
                                  " in setPrice");
     System.exit(0);
  }
}
```

Generating the documentation

- To generate the documentation, use the **`javadoc`** tool:

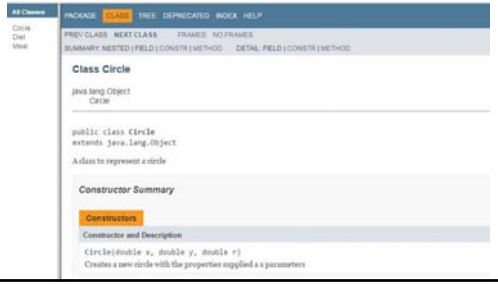
```
U:\myjava>javadoc Meal.java
```



Generating the documentation

- You can also generate documentation for every Java class in a directory

```
U:\myjava>javadoc java8\*.java
```



Documenting a package

- Generate the **`javadoc`** documentation by invoking the following from the **parent** directory:

```
U:\myjava>javadoc packagename
```

Summary of key points

- The keyword `this` refers to the current instance of the class and `null` refers to no instance of any class
- Enumerations are a special kind of class
- To run a Java program the class must have a **main method** and that is what is obeyed.
- A main method can be used as a **test harness**
- Classes and **packages** should be documented with `javadoc`