Reference, classes, enums and packages

This lecture will

- Introduce the `null` keyword and demonstrate another use for the `this` keyword
- Revisit the difference between assignment and reference and its implications for objects
- Examine the use of local variables within methods
- Revisit enumerations
- Look at values returned from methods
- Explain how to 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.NORMAL);
```

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

```
Meal chips;
```

- We can emphasize that chips does not contain a valid reference by 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:

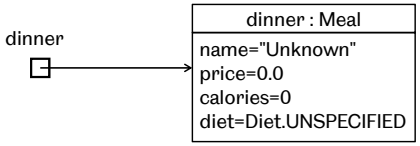
```
chips = new Meal();
```

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

Assignment and reference

- The following declaration creates a new object that is referred to by the variable **dinner**.

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

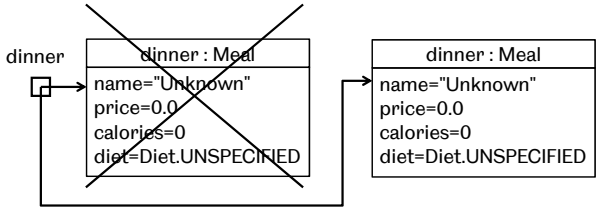


Assignment and reference

- If we now do the following assignment

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

a new object is created and the old object (which no longer has a reference) is garbage collected.

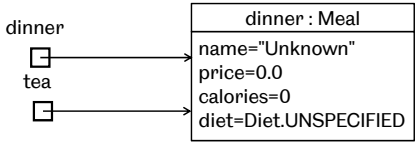


Assignment and references

- Now consider this:

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

- The second statement does not create a new object; **the reference is copied**.



Assignment and references

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

- **Dinner** and **tea** refer to the same object. We can demonstrate this as follows:

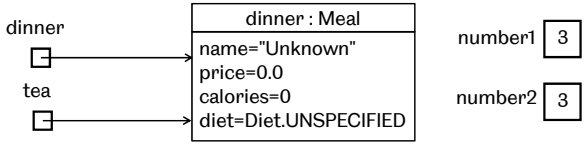
```
dinner.setPrice(30.99);
System.out.println(
   "The price of Tea = " + tea.getPrice());
```

```
The price of Tea = 30.99
```

Assignment and references

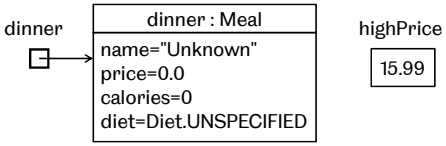
- Note that if we do the same thing with simple types then we have two independent variables that contain the same value:

```
Meal dinner = new Meal();
Meal tea = dinner;
int number1 = 3;
int number2 = number1;
```



Call-by-value

- This is an example of **call-by-value**. Consider the following situation:



- Now this statement is executed:

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

which uses this method

```
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

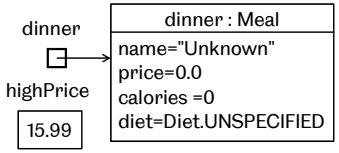
- The variable **p** is only accessible within the **`setPrice`** method; it is a **local variable**.
- After the **`setPrice`** method has executed, **p** is garbage collected.

Actual parameter is unchanged

- In call-by-value, modifying the formal parameter does not change the actual parameter:

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

- Assume that we start with the situation shown on the right:



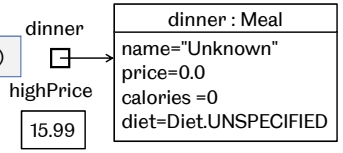
Actual parameter is unchanged

- In call-by-value, modifying the formal parameter does not change the actual parameter:

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

- Now we call

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



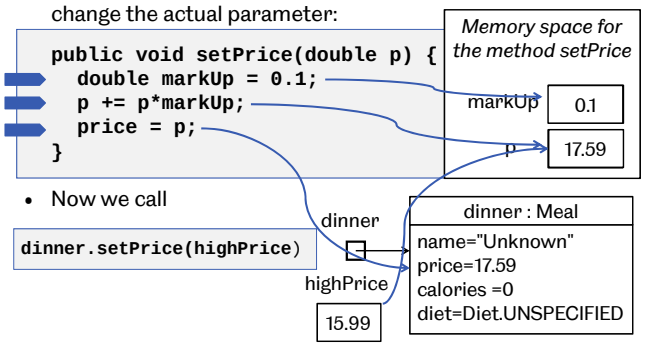
Actual parameter is unchanged

- In call-by-value, modifying the formal parameter does not change the actual parameter:

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

- Now we call

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



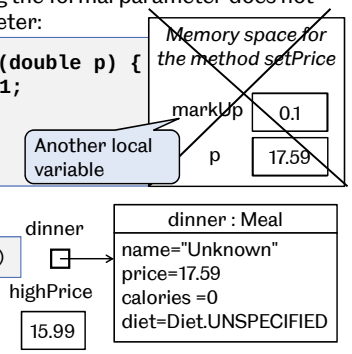
Actual parameter is unchanged

- In call-by-value, modifying the formal parameter does not change the actual parameter:

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

- Now we call

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



Local variable 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

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
- So
  - You never need to use **new**
  - The values like **NORMAL**, **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 {
  NORMAL, 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

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);
}
```

Printing enums

```
public String toString() {
     return name +  ", Price=" + price +
            ", Calories=" + calories +
            " and Diet=" + diet;
}
```

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

```
Special Pizza, Price=8.99, Calories=800 and Diet=NORMAL
```

Printing enums again

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

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

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

A `toString` method for enumerations

- **enum**s are a kind of class
- Classes can have methods
- An **enum** can have a **`toString`** method too

Note semicolon

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

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

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

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 a "+diet;
}
```

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

```
Special Pizza, Price=8.99, Calories=800 which is
                           suitable for a normal 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 a "+diet;
}
```

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

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

Reading in enum values

- If you are asking a user for a value which will be assigned to an **enum** variable you can use the **`valueOf()`** method
  - **`Integer.valueOf("12345")`** turns its String parameter into the integer 12345
  - **`Diet.valueOf("NORMAL")`** turns its String parameter into **`Diet.NORMAL`**
- This only works if the user can be trusted not to type in anything unexpected

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

works with Normal, vegan and VEGETARIAN but not ovo-lacto vegetarian

Creating enums

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

Returning computed values

- Instead of returning the value of an instance variable, an expression can be computed by a **`return`** statement.
- The type of the expression and the return type of the method must be compatible.

❓ **In general, when is it appropriate to compute a return value, as opposed to returning an attribute value?**

Example – computing return values

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

     public Circle(double x, double y, double r) {
       xCentre = x; yCentre = y;
       radius = r;
     }

     ……

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

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

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;
     }
```

- However, this simply assigns the formal parameter to itself

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.

The `main` Method

- When we started this course we wrote programs like this

```
public class Simple {
   public static void main(String[] args) {
      System.out.print("Running a Java application");
      System.out.println("...finished.");
   }
}
```

A class which contained nothing except a **`main`** method which 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

Scope and visibility

- Every variable in Java has a **scope**, which determines how long it exists in memory and when it expires.
- Once a variable goes out of scope, the memory that it occupied is marked for garbage collection and the variable cannot be referenced.
- Java uses **scope rules**, the most important of which are:
  - Variables declared within a method or compound statement have **local scope**. They exist from the line they are declared until the closing bracket of the method or code block.
  - Instance variables have **global scope**. They are in scope so long as their enclosing object is in scope.

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

- Now there is a **hole in the scope** of instance variable **x**. Its value is accessible globally except in **methodOne**, where **x** refers to local variable with the same name.

Exercise

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

```
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 solution

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

```
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("meth
   }
   public void methodC() {
      methodA(x);
      System.out.println("meth
   }
   public void methodD(int x)
      this.x = this.x+x;
      methodA(x);
      System.out.println("methodD x and y " + x + ", " + y);
   }
```

```
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
```

Packages and scope

- In practice Java classes are arranged into **packages**, which are closely linked to the underlying directory structure.
- Consider the Sheffield package. In your directory (e.g., **myjava**) you have a directory called **sheffield** which contains the byte code for the package.
- Every class in the **sheffield** package has the following first line, which tells the compiler that it belongs to the package:

```
package sheffield;
```

- Other programs in the **myjava** directory use the **sheffield** package. In these programs, we have an `import` statement:

```
import sheffield.*;
```

Finding a package

- The Java compiler looks for a package in three places:

1. From the current working directory, it will look for the directory structure specified in the **`import`** statement.
2. It will search from the directory in which the JDK was installed.
3. It will examine an environment variable called CLASSPATH, which lists all the directories in which classes may be found.

```
Under MS-DOS/Windows:
set CLASSPATH=.;c:\somedirectory;c:\adir\subdirectory

Under Unix/Mac OSX:
setenv CLASSPATH=.:/somedirectory:/adir/subdirectory
```

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
```

- Now you can use all the classes in your package within any class in the top level directory which starts

```
import packagename.*;
```

Packages and scope

- 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.
- For classes
  - **public** : the class is visible everywhere.
  - **default** : if no access permission is specified, the class is only visible to other classes in the same package.

Scope for methods, constants, variables

- **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 are either public (accessor methods) or private (if they support the implementation of public methods).

Summary of key points

- The keyword `this` refers to the current instance of the class and the keyword `null` refers to no instance of any class
- Enumerations are a special kind of class which should be in their own file and can have instance and class methods
- Methods can return expressions as well as values of existing variables
- To run a Java program the class invoked must have a main method and that method is what is obeyed.
- A main method can be used as a test harness for a class that is to be used by other classes
- Be very careful if you reuse instance variable identifiers as local variables within methods or formal parameter names
- Groups of classes can be bundled into packages and imported into other classes