Object composition

This lecture will

- Look at **copies of objects**
- Teach you about using the **current date**
- Look at **object composition**
- Demonstrate how **arrays can be objects** as well as contain pointers to objects
- Explain how arrays can be returned by methods and passed as parameters to methods

Classes as Toolkits

- Classes are designed to be used by other classes
- They can be used simply to wrap a lot of methods together as a toolkit.
- Consider the Math class – a static class that consists of a lot of useful static methods and two static constants; Math.E and Math.PI
- You have been using it in many other classes

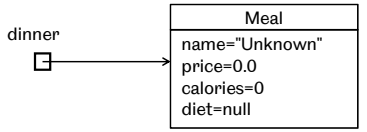
Immutable classes

- You have been using the String class too
- String is not static, you can create instances, but it is immutable – objects cannot be changed
- This is sometimes a sensible approach – if you have a circle object maybe you don't need to be able to change its radius or centre
- Think about how your objects are to be used by other classes

Assignment and reference

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

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

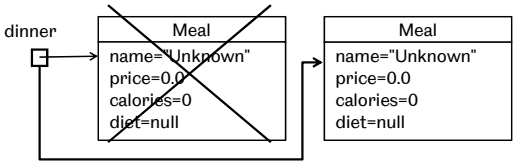


Assignment and reference

- 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.

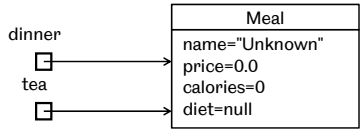


Assignment and references

- 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

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

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

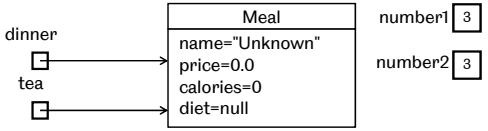
```
dinner.setPrice(30.99);
System.out.println(
       "The price of Tea = " + tea.getPrice());
```

```
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;
```



Copying an object

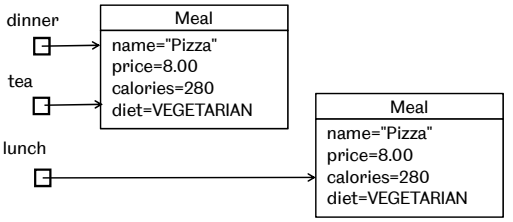
- Suppose we wanted a copy of the object **`dinner`** not just a reference to the same object
- We can give **`Meal`** a **`copy`** method

```
public Meal copy()  {
  return new Meal(name, price, calories, diet);
}
```

```
Meal lunch = dinner.copy();
```

Copying a reference v. an object

```
Meal dinner = new Meal("Pizza", 8.00, 280,
                       Diet.VEGETARIAN);
Meal tea = dinner;
Meal lunch = dinner.copy();
```



Object composition

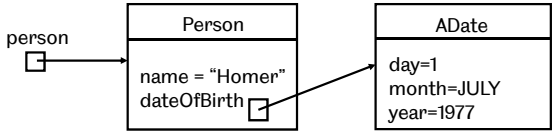
- Objects can use other objects
- Their instance variables can be references to other objects
- This is know as **object composition**
- Object composition arises naturally in the real world

Object composition

- Objects can use other objects
- **`Meal`**'s instance variables are three basic types and a reference to an **`enum Diet`**
- Copying a basic type creates a new thing of the same type
- Copying a reference to an object just creates a new pointer
- If the object is an **`enum`** it is no problem but with other objects it can cause problems

Composition in the real world

- Consider a person class with attributes **`Name`** and **`dateOfBirth`**
- A date is an obvious candidate for an independent class because it is likely to be reused



The `ADate` class

```
public class ADate {
    private int day;
    private Month month;
    private int year;

    public ADate(int day, Month m, int year) {…}
    public ADate(int day, int month, int year) {…}
    public ADate(int day, String month, int year) {…}

    public int getDay() { return day;  }
    public Month getMonth() { return month; }
    public int getYear() { return year;  }

    public String toString() {
       if  (  month == null ) return day+"/???/"+year;
       else  return day+"/"+month.toNumber()+"/"+year;
    }
}
```

Don't use '/'

The `Month` class

```
public enum Month {
    JANUARY,   FEBRUARY,   MARCH,   APRIL,   MAY,
    JUNE,   JULY,   AUGUST,   SEPTEMBER,  OCTOBER,
    NOVEMBER,   DECEMBER;

    public int toNumber() { return ordinal()+1; }

    public String toString() {…}

    public static Month valueOf(int m) {
       switch(m) {
          case 1 : return JANUARY;
          case 2 : return FEBRUARY;
          …
       }
       return null;
    }
}
```

Another method available for any **`enum`**. Counts the constants from zero

The `Person` class

```
public class Person {
   private static final String NO_NAME = "NONAME";

   private String name;
   private ADate dateOfBirth;

   public Person(String n, ADate d) {
      name = n;
      dateOfBirth = d;
   }
   public Person() { this ( NO_NAME, null );   }

   public void setName(String n) { name = n; }
   public String getName() { return name; }
   public void setDateOfBirth(ADate d) { …… }
   public ADate getDateOfBirth() {return dateOfBirth;}

   public String toString () {
      return name + "(" + dateOfBirth + ")";
   }
}
```

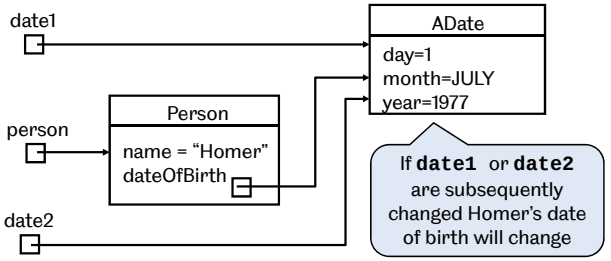
Testing Objects within objects

```
public static void main (String[] args) {
   ADate date1 = new ADate(1, Month.JULY, 1964);
   Person person = new Person("Homer", date1);
   System.out.println("person: " + person);
   ADate date2 = person.getDateOfBirth();
   System.out.println("date2: " + date2);
}
```

```
person: Homer (1/7/1977)
date2: 1/7/1977
```

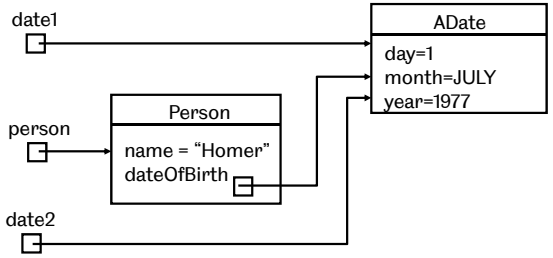
Memory during the main method

```
ADate date1 = new ADate(1, Month.July, 1964);
Person person = new Person("Homer", date1);
ADate date2 = person.getDateOfBirth();
```



Memory during the main method

- This situation arises because the constructor copies a reference to the date and the accessor returns one – we should be using copies



Using copies

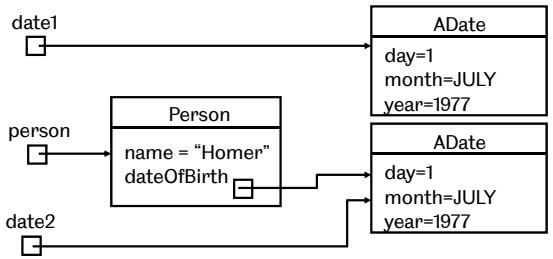
- We add a method called copy to the ADate class:

```
public class ADate {
   private int day;
   private Month month;
   private int year;

   public ADate(int d, Month m, int y) {
      day = d;   month = m;  year = y;
   }
……
   public ADate copy() {
      return new ADate(day, month, year);
   }
}
```

Using copy in a constructor

```
public Person(String n, ADate d) {
    name = n;
    dateOfBirth = d.copy();
}
```



Using copy everywhere

```
public Person(String n, ADate d) {
   name = n;
   dateOfBirth = d.copy();
}
public void setDateOfBirth(ADate d) {
   dateOfBirth = d.copy();
}
public ADate getDateOfBirth() {
   return dateOfBirth.copy();
}
```

Always a good idea

Always a good idea

Be careful in accessors; it can cause new problems

Information hiding

- We could hide use of the **`ADate`** class within the **`Person`** class so that the users of the class had no access to it

`Person` class with concealed `ADate`

```
private String name;
private ADate dateOfBirth;

public Person(String n, int d, Month m, int y) {
    name = n; dateOfBirth = new ADate(d,m,y);
}
public void setDateOfBirth(int d, Month m, int y) {
    dateOfBirth = new ADate(d,m,y);
}
public int getDayDateOfBirth() {
    return dateOfBirth.getDay();
}
public Month getMonthDateOfBirth() {
    return dateOfBirth.getMonth();
}
public int getYearDateOfBirth() {
    return dateOfBirth.getYear();
}
```

Using a concealed class

Advantage:

- The class user is unaware that object composition is used because the **`ADate`** class does not appear in the signature of any methods of the **`Person`** class.

Disadvantage:

- Overhead on method calls when using the class

Ages

- The **`Person`** class should have a **`getAge()`** method but for that we need today's date
- Java has a class called **`LocalDate`** which will help, it has static methods like
  - **`now()`** to find the current date
  - **`of()`** to create a specific date specified by a year, month and day in that order
- It has instance methods **`isBefore()`** and **`isAfter()`** to compare dates

Ages

- The **`LocalDate`** class is only accessible if you start your program with
  `import java.time.*;`
  If you also use
  `import sheffield.*;`
  the order of **`import`** statements is irrelevant

Turning a date into an age

```
import java.time.*;
```

The date at which the program is run

```
public int getAge() {
   int currentYear = LocalDate.now().getYear();
   int age = currentYear - dateOfBirth.getYear();

   if  (  LocalDate.now().isBefore(
          LocalDate.of(currentYear,
                dateOfBirth.getMonth().toNumber(),
                dateOfBirth.getDay()))  )
      age--;
   return age;
}
```

The Person's birthday this year

The comparison

Using the Age

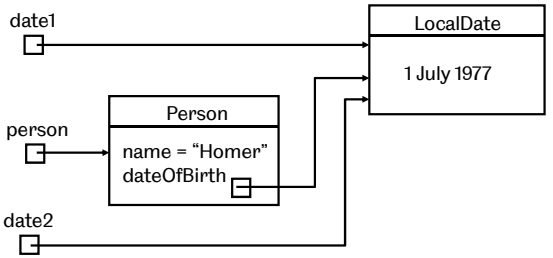
```
Homer (40) whose birthday is 1st July
```

```
public String toString () {
   if  (  dateOfBirth == null  )
      return name + " whose date of birth is unknown";
   else
      return name + " ("+getAge()+
            ") whose birthday is "+dateOfBirth.asDay();
}
```

```
public String asDay() {
   switch(day) {
      case 1:    case 21:   case 31:
         return day+"st "+month;
      case 2:    case 22:
         return day+"nd "+month;
      case 3:    case 23:
         return day+"rd "+month;
      default:
         return day+"th "+month;
   }
}
```

Memory during the main method

- If we had use LocalDate in Person



- This would not be a problem because LocalDate is immutable

Arrays as Objects

- You create objects with the key word **new**

```
Person homer = new Person();
```

- You create arrays with the key word **new**

```
int [] myArray = new int[5];
```

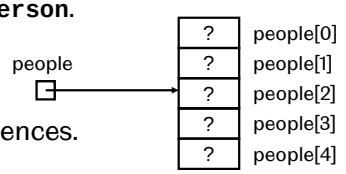
- Because arrays are a special kind of object
- Objects can contain arrays and arrays can contain objects

Arrays of objects

- We can declare an array of 5 people as

```
Person[] people = new Person[5];
```

- This states that a variable of type **`Person[]`** (an array of **`Person`**) refers to a block of 5 elements of type **`Person`**.
- Initially **`people`** contains an array of **`null`** references.

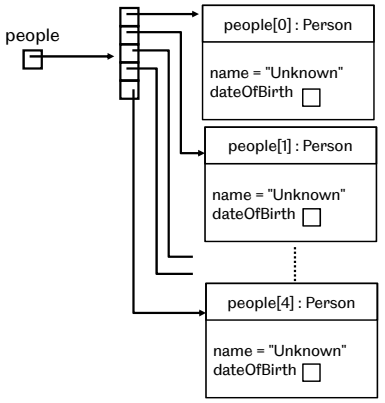


Initialising arrays of objects

- Before the array of objects can be used, each element must be initialised:

```
for (int i=0; i<people.length; i++)
   people[i] = new Person();
```

An initialised array of objects



Manipulating an array of objects

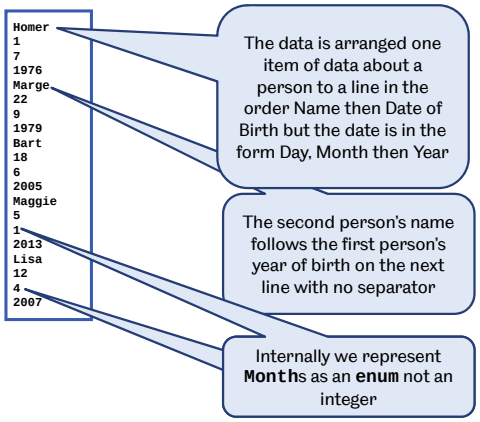
- We can manipulate individual attributes of array elements (**Person** objects) by using an array subscript and the methods of the **Person** class:

```
people[0].setName("Homer");
people[0].setDateOfBirth(
            new ADate(1, Month.JULY, 1977));
System.out.println(people[0]);
```

Reading an array of objects from a file

- We can create a simple database in the form of a text file, then read these values into an array of objects for processing (e.g., searching and sorting).
- First, we declare an array that is large enough to store a typical file of data.
- A variable is used to record the number of items **actually** read from the file.
- We recognise the end of the file using the **eof()** method of **EasyReader**

Contents of the file `simpsons.txt`



Reading an array of objects from file

```
import sheffield.*;
public class TestReadPersons {
   public static final int MAX_PERSONS = 20;
   public static void main (String[] args) {
      EasyReader file =
            new EasyReader("simpsons.txt");
      Person [] personTable = new Person[MAX_PERSONS];
      // read each person from the file ....
      // display the contents of the array .....
   }
}
```

Reading in a Person

```
// read each person from the file
int numPersons = 0;
while (!file.eof()&&(numPersons<MAX_PERSONS)) {
   String name = file.readString();
   int day = file.readInt();
   int monthNo = file.readInt
   Month month = Month.valueOf(monthNo);
   int year = file.readInt();
   personTable[numPersons]= new Person(name, day,
           month, year);
   numPersons++;
}
```

```
Homer
1
7
1976
Marge
22
9
1979
Bart
18
6
2005
Maggie
5
1
2013
Lisa
12
4
2007
```

Reading in a person – alternative version

```
// read each person from the file
int numPersons = 0;
while (!file.eof()&&(numPersons<MAX_PERSONS)) {
   personTable[numPersons]= new Person(
        file.readString(),                //Name
        file.readInt(),                   //Day of mont
        Month.valueOf(file.readInt()),    //Month numbe
        file.readInt()                    //Year
   );
   numPersons++;
}
```

```
Homer
1
7
1976
Marge
22
9
1979
Bart
18
6
2005
Maggie
5
1
2013
Lisa
12
4
2007
```

Reading an array of objects from file – the end

```
// display the contents of the array
for (int i=0; i<numPersons; i++)
    System.out.println(personTable[i]);
```

Note we can't use an enhanced **for** loop here because we are not going through every element of the array

Array Copy

- Creating an array larger than we think we need and partially filling it is a common situation
- Once we have added all the elements we need we can tidy things up by creating a new array of exactly the right size and transferring the data using **arraycopy**

Array Copy

- Tidying things up by creating a new array of exactly the right size and transferring the data using **arraycopy**

```
//After reading in the data
Person [] people = new Person[numPersons];
System.arraycopy(personTable,0,people,0,numPersons);
```

Number of elements to copy

Array to copy from

Index of first value to copy

Array to copy to

Index of first position to copy to

Array Copy

```
//After reading in the data
Person [] people = new Person[numPersons];
System.arraycopy(personTable,0,people,0,numPersons);
// display the contents of the array
for (Person p : people) System.out.println(p);
```

- Best not done in the **main** method. **Why not?**

Arrays as objects – returned by a method

```
public class TestReadPersons {
   public static final int MAX_PERSONS = 20;
   public static Person[] peopleFromFile(String fName){
      EasyReader file = new EasyReader(fName);
      Person [] personTable = new Person[MAX_PERSONS];
      int numPersons = 0;
      // read each person from the file ....
      Person [] result = new Person[numPersons];
      System.arraycopy(personTable,0,result,0,
                                        numPersons);
      return result;
   }
   public static void main (String[] args) {
      Person [] people = peopleFromFile("simpsons.txt");
      // display the contents of the array ....
    }
}
```

Must be static

Arrays as objects – returned by a method

```
public static Person[] peopleFromFile (String fName)
   EasyReader file = new EasyReader(fName);
   Person [] personTable = new Person[MAX_PERSONS];

   // read each person from the file ....
   int numPersons = 0;
   ……

   Person [] result = new Person[numPersons];
   System.arraycopy(personTable,0,result,0,numPersons);
   return result;
}
public static void main (String[] args) {
     Person [] people = peopleFromFile("simpsons.txt");
     // display the contents of the array ....
}
```

Nothing points to this when the method finishes

Something points to this

Linear search in an array of objects

- To search the array for a person with a particular name, we provide methods for the **`Person`** class to do the matching of the **`name`** attribute and supplied name:

```
public boolean matchName(String n) {
   return name.equals(n);
}
public boolean matchNameIgnoreCase(String n) {
   return name.equalsIgnoreCase(n);
}
```

Linear searching

- Find all **`Person`** objects in the array with the same name as a given name:

```
String search =
     keyboard.readString("Enter name: ");

for (Person p : people)
   if (  p.matchNameIgnoreCase(search)  )
      System.out.println(p);
```

Arrays as arguments

- Recall that **arrays are objects**; so when we pass an array as a parameter to a method, we pass a pointer to it

```
public static void displayTable(Person[] table){
   for (Person p : table)
      System.out.println(p);
}
```

- If you want to know how big the table is use **`table.length`**

Passing arrays by reference

```
import sheffield.*;
public class ReadPerson2{
   public static final int MAX_PERSONS = 20;

   public static Person[]peopleFromFile(String fName){..}
   public static void displayTable(Person[] table){
       for (Person p : table) System.out.println(p);
   }
   public static void main (String[] args) {
      Person[] peopleTable = peopleFromFile("simpsons.txt");
      System.out.println("There are "+peopleTable.length+
                                                 " people";
      displayTable(peopleTable);
   }

}
```

Note how the use of methods improves readability

Summary of key points

- Objects may contain references to other objects
- Objects can be hidden within other objects
- Java knows the current date and the `LocalDate` class can be used to access it
- Arrays are objects and we can have arrays of objects
- We can copy arrays with `System.arraycopy`