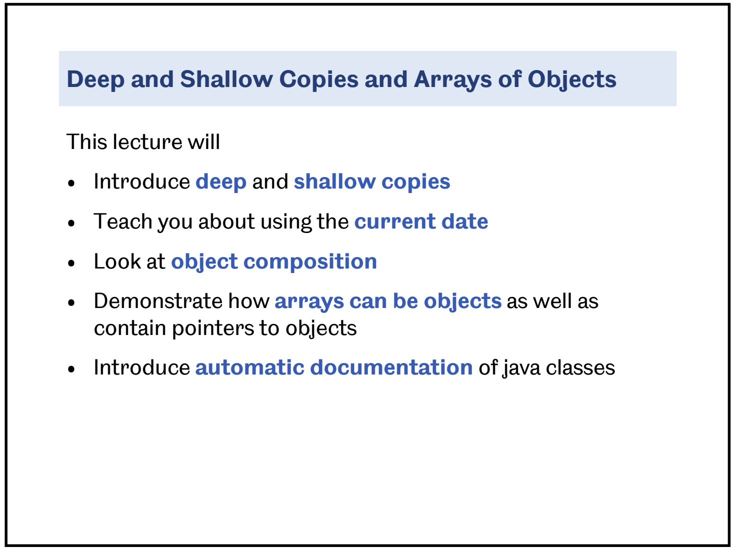
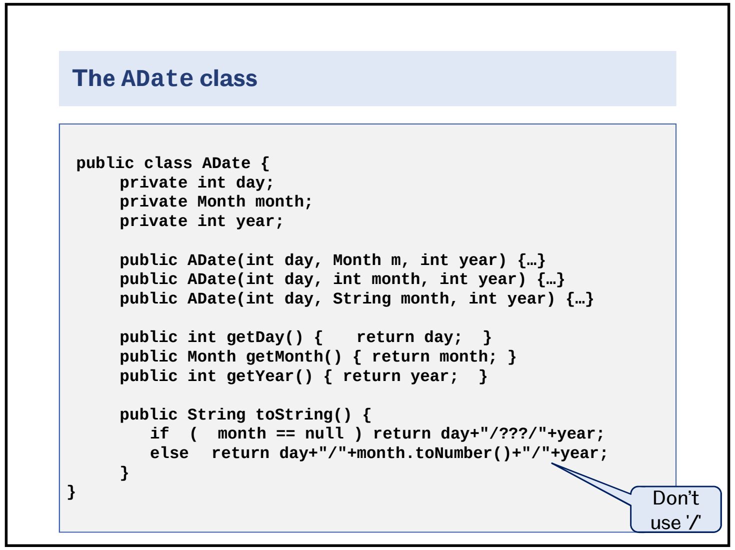


Deep and Shallow Copies and Arrays of Objects

This lecture will

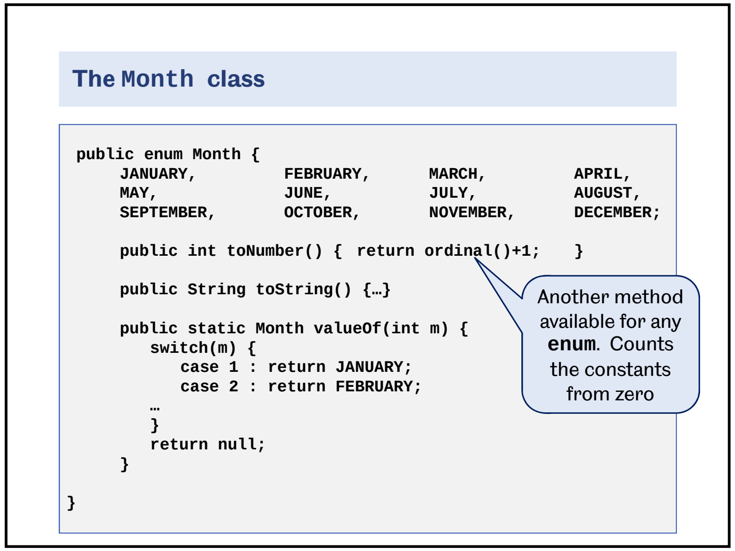
- Introduce **deep** and **shallow copies**
- Teach you about using the **current date**
- Look at **object composition**
- Demonstrate how **arrays can be objects** as well as contain pointers to objects
- Introduce **automatic documentation** of java classes



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;
    }
}
```

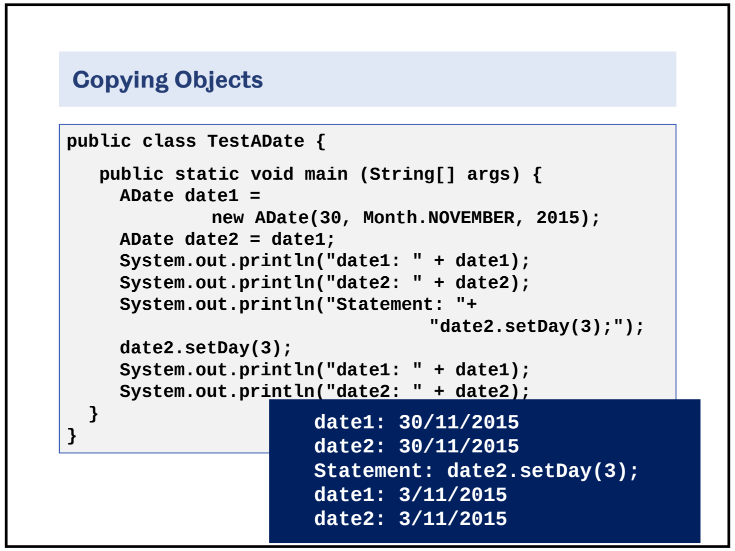
Don't use '/'



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



Copying Objects

```
public class TestADate {

   public static void main (String[] args) {
     ADate date1 =
             new ADate(30, Month.NOVEMBER, 2015);
     ADate date2 = date1;
     System.out.println("date1:  " + date1);
     System.out.println("date2:  " + date2);
     System.out.println("Statement:  "+
                              "date2.setDay(3);");
     date2.setDay(3);
     System.out.println("date1:  " + date1);
     System.out.println("date2:  " + date2);
  }
}
```

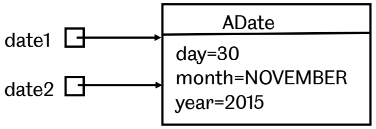
```
date1:  30/11/2015
date2:  30/11/2015
Statement:  date2.setDay(3);
date1:  3/11/2015
date2:  3/11/2015
```

Shallow copies

- The statement

```
ADate date2 = date1;
```

only copies a **reference**, giving multiple references to the same object.



- Hence, when **`date2`** is changed, so is **`date1`**. We say that **`date2`** is a **shallow copy** of **`date1`**.

Deep copies

- To make a **deep copy**, a new object is created with the same contents as the original.
- 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);
  }
}
```

Copying Objects

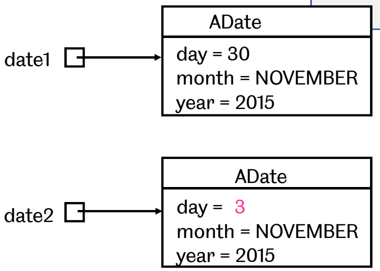
```
public class TestADate2 {
  public static void main (String[] args) {
     ADate date1 =
              new ADate(30, Month.NOVEMBER, 2015);
     ADate date2 = date1.copy();
     System.out.println("date1: " + date1);
     System.out.println("date2: " + date2);
     System.out.println("Statement: "+
                 "date2.setDay(3);");
     date2.setDay(3);
     System.out.println("date1: " + date1);
     System.out.println("date2: " + date2);
  }
}
```

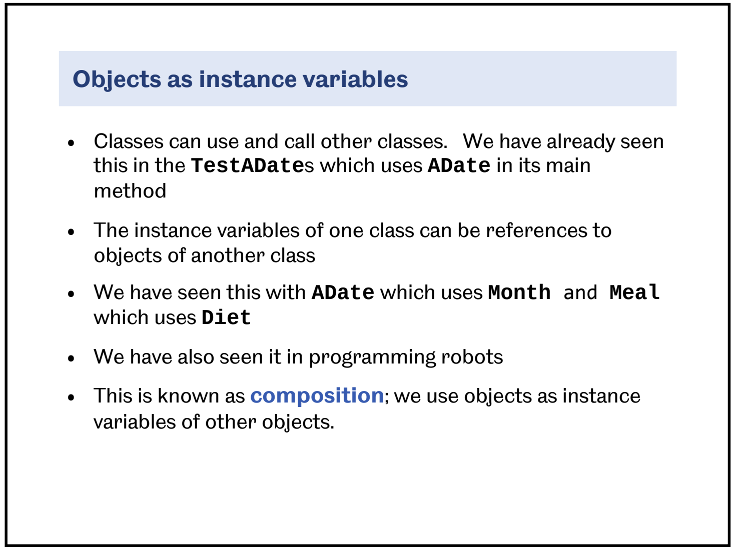
```
date1: 30/11/2015
date2: 30/11/2015
Statement: date2.setDay(3);
date1: 30/11/2015
date2: 3/11/2015
```

Memory during `TestADate3`

```
   ADate date1 = new ADate(30, Month.NOVEMBER, 2015);
   ADate date2 = date1.copy();
   ……
   date2.setDay(3);
```

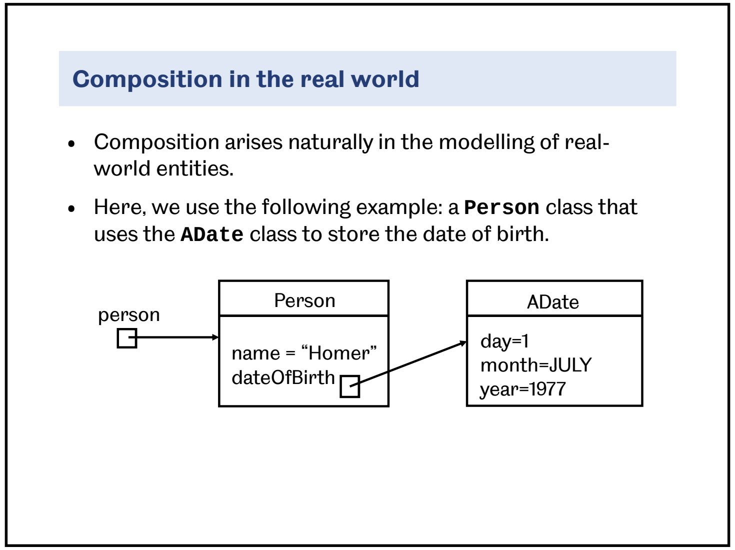
```
public ADate copy() {
     return new ADate(
            day, month, year);
```





Objects as instance variables

- Classes can use and call other classes. We have already seen this in the **`TestADate`**s which uses **`ADate`** in its main method
- The instance variables of one class can be references to objects of another class
- We have seen this with **`ADate`** which uses **`Month and Meal`** which uses **`Diet`**
- We have also seen it in programming robots
- This is known as **composition**; we use objects as instance variables of other objects.



Composition in the real world

- Composition arises naturally in the modelling of real-world entities.
- Here, we use the following example: a **`Person`** class that uses the **`ADate`** class to store the date of birth.

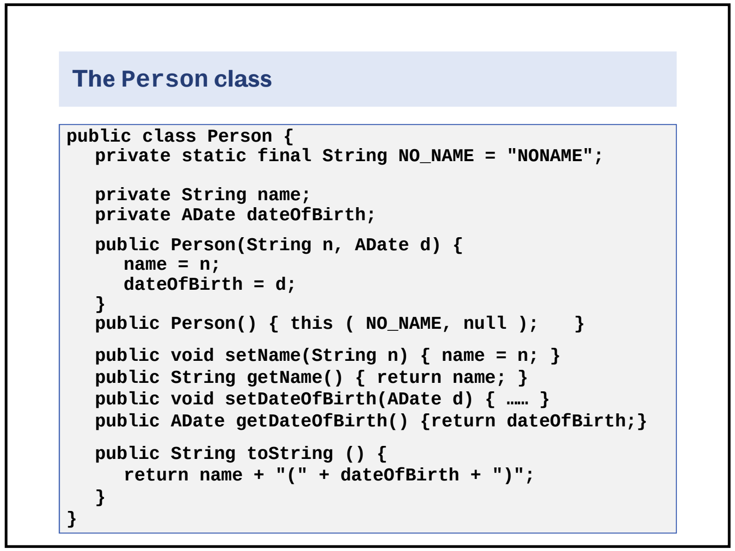
person

Person

name = “Homer”
dateOfBirth

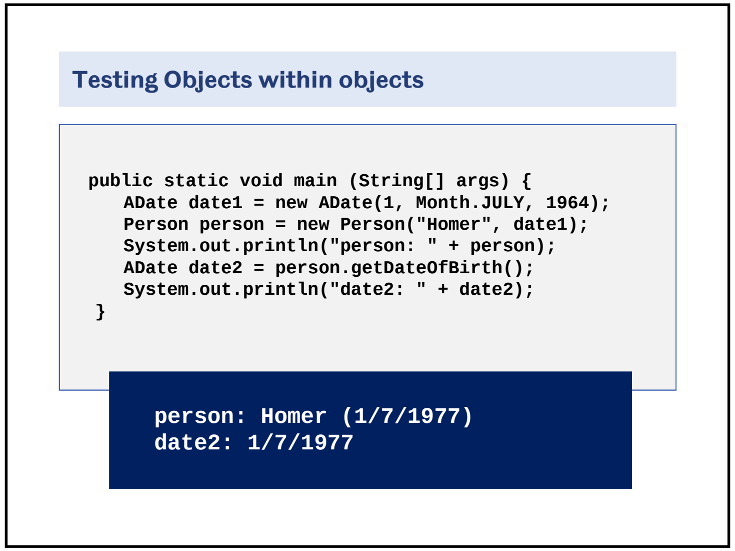
ADate

day=1
month=JULY
year=1977



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 + ")";
   }
}
```



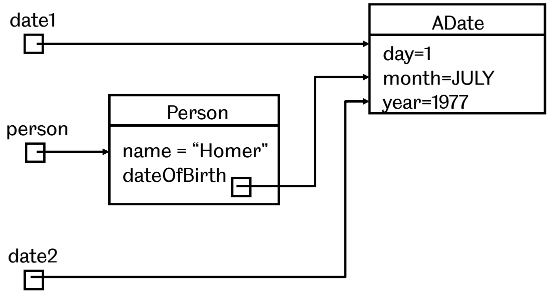
Testing Objects within objects

```
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();
```



date1

ADate

day=1

month=JULY

year=1977

person

Person

name = “Homer”

dateOfBirth

date2

Deep copies in the Person class

- The **Person** constructor makes a **shallow** copy of the **ADate** instance that is passed to it:

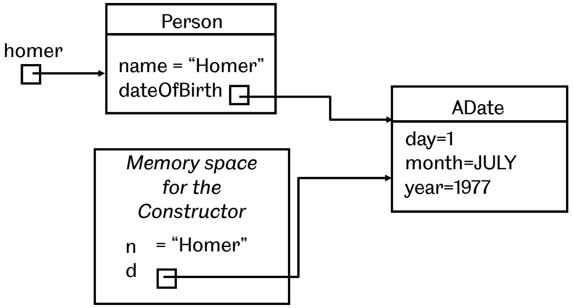
```
public Person(String n, ADate d) {
    name = n;
    dateOfBirth = d;
}
```

This approach is prone to error.

- We were left with three references to the same object; changes to **date1** and **date2** will affect Homer’s **dateOfBirth.**
- We can reduce this problem by using an **anonymous reference**

Anonymous References

```
Person homer = new Person("Homer",
                  new ADate(1,Month.JULY,1977));
```



homer

Person

name = “Homer”

dateOfBirth

ADate

day=1

month=JULY

year=1977

Memory space for the Constructor

n = “Homer”

d

Program defensively with deep copies

- Better still, we could use **deep** copies in the relevant methods:

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

Be careful in accessors; it can cause new problems

Always a good idea

Always a good idea

- **Program defensively**; in general, deep copies are preferred over shallow copies.

Information hiding

- We could hide the **`ADate`** class within the **`Person`** class by providing methods in **`Person`** that take parameters for both classes.

`Person` class with concealed `ADate`

```
private String name;
private int age;
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();
}
```

Better information hiding

- We could also hide the **`ADate`** class within the **`Person`** class by providing methods in **`Person`** that take parameters for both classes.

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

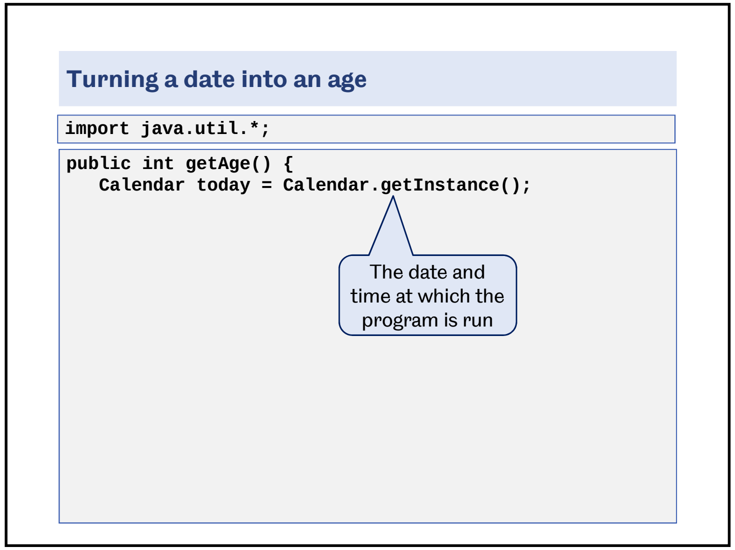
- It might be useful if the **`Person`** class had a **`getAge()`** method but for that we need today's date
- Java has a class called **`Calendar`** which has some similarities to our date and has a static method **`getInstance()`** that can be used to create an object which represents the current date when the program is run
- The **`Calendar`** class is only accessible if you start your program with

  ```
      import java.util.*;
  ```

  If you also use

  ```
      import sheffield.*;
  ```

  the order of **`import`** statements is irrelevant

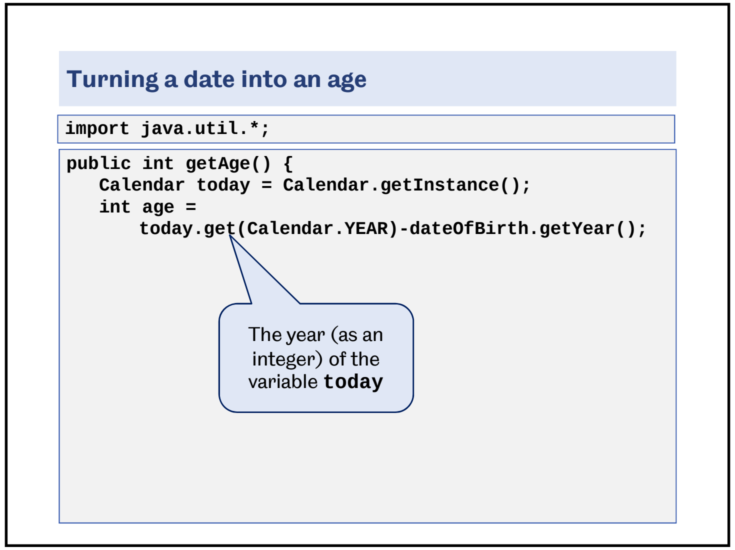


Turning a date into an age

```
import java.util.*;
```

```
public int getAge() {
   Calendar today = Calendar.getInstance();
```

The date and time at which the program is run

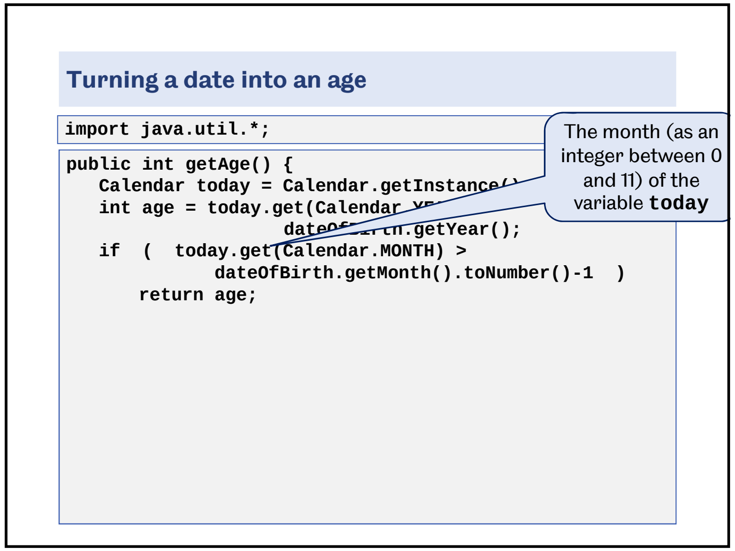


Turning a date into an age

```
import java.util.*;
```

```
public int getAge() {
   Calendar today = Calendar.getInstance();
   int age =
       today.get(Calendar.YEAR)-dateOfBirth.getYear();
```

The year (as an integer) of the variable **today**

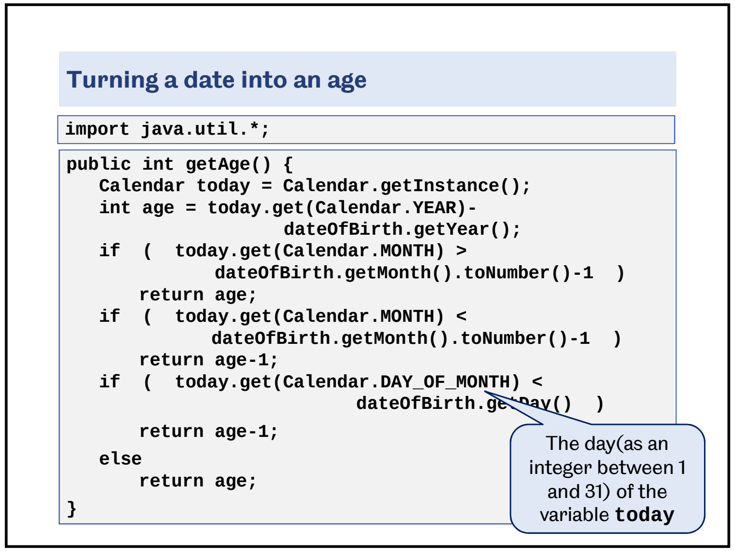


Turning a date into an age

```
import java.util.*;
```

```
public int getAge() {
   Calendar today = Calendar.getInstance();
   int age = today.get(Calendar.YEAR)-
                    dateOfBirth.getYear();
   if  (  today.get(Calendar.MONTH) >
              dateOfBirth.getMonth().toNumber()-1   )
       return age;
```

The month (as an integer between 0 and 11) of the variable **today**

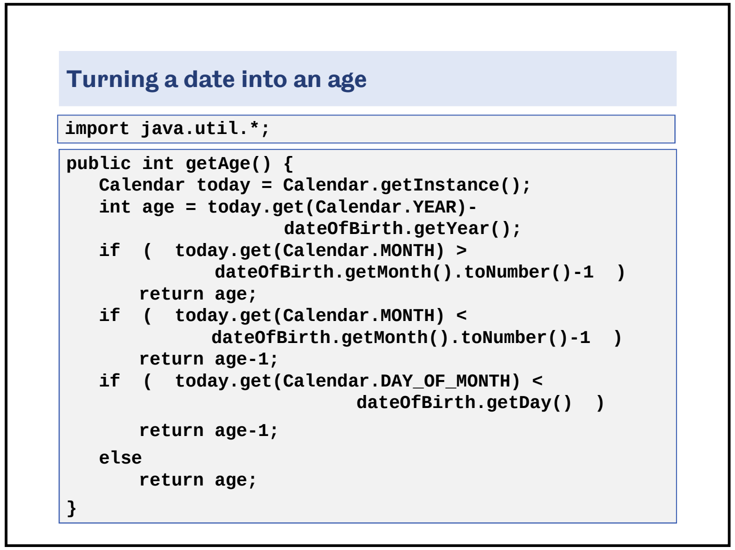


Turning a date into an age

```
import java.util.*;
```

```
public int getAge() {
   Calendar today = Calendar.getInstance();
   int age = today.get(Calendar.YEAR)-
                    dateOfBirth.getYear();
   if  (  today.get(Calendar.MONTH) >
              dateOfBirth.getMonth().toNumber()-1   )
       return age;
   if  (  today.get(Calendar.MONTH) <
              dateOfBirth.getMonth().toNumber()-1   )
       return age-1;
   if  (  today.get(Calendar.DAY_OF_MONTH) <
                       dateOfBirth.getDay()   )
       return age-1;
   else
       return age;
}
```

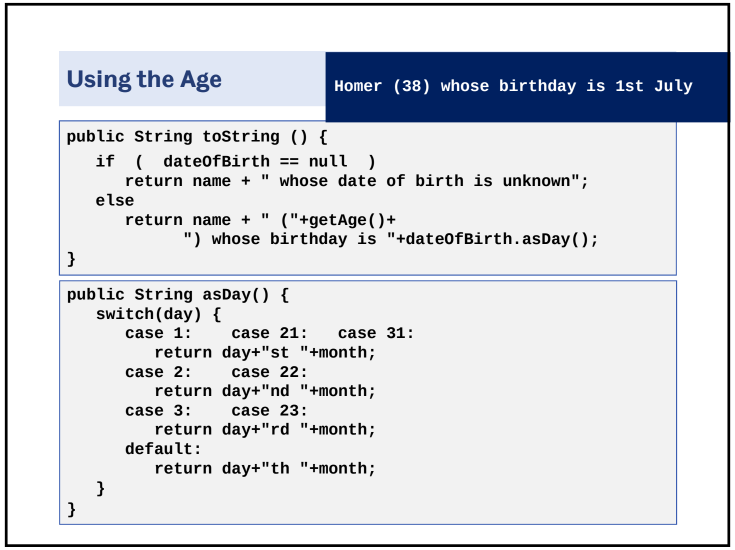
The day(as an integer between 1 and 31) of the variable **today**



Turning a date into an age

```
import java.util.*;
```

```
public int getAge() {
   Calendar today = Calendar.getInstance();
   int age = today.get(Calendar.YEAR)-
                    dateOfBirth.getYear();
   if  (  today.get(Calendar.MONTH) >
              dateOfBirth.getMonth().toNumber()-1  )
       return age;
   if  (  today.get(Calendar.MONTH) <
              dateOfBirth.getMonth().toNumber()-1  )
       return age-1;
   if  (  today.get(Calendar.DAY_OF_MONTH) <
                           dateOfBirth.getDay()  )
       return age-1;
   else
       return age;
}
```

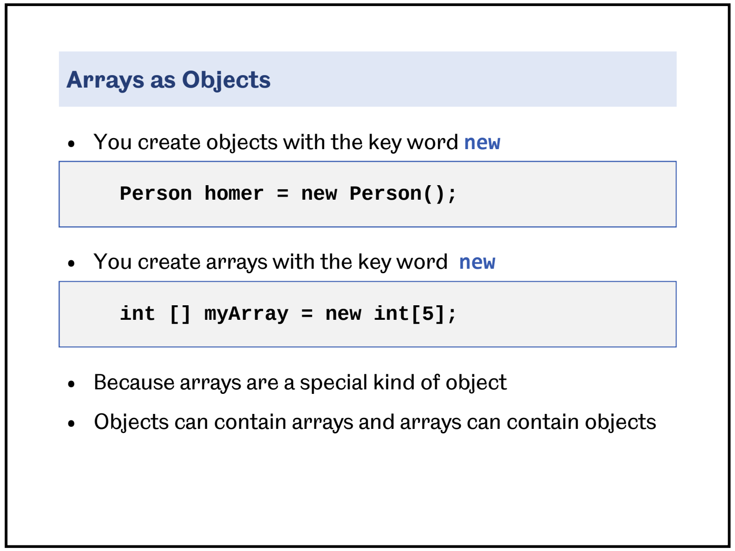


Using the Age

```
Homer (38) 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;
   }
}
```



Arrays as Objects

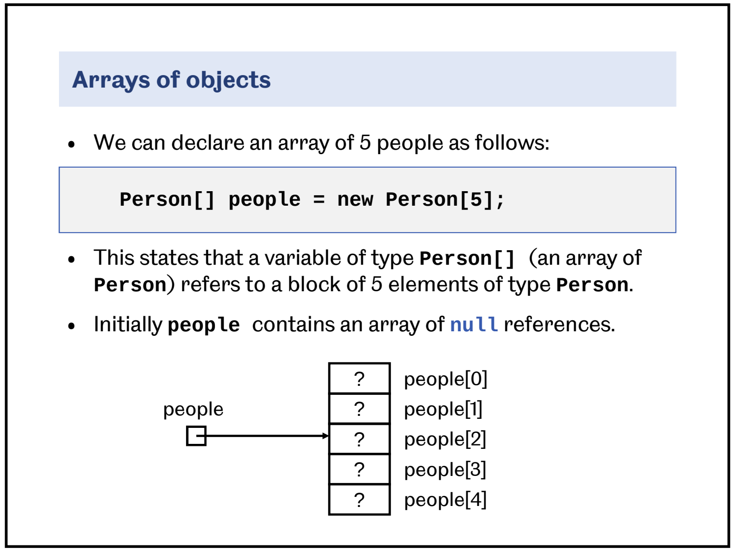
- You create objects with the key word **new**

```
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 follows:

```
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

- The declaration of **people**

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

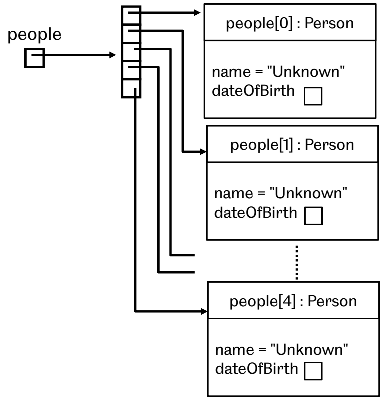
invokes the constructor for the array, but doesn't create any **Person** instances (since no constructor for **Person** has been called)

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

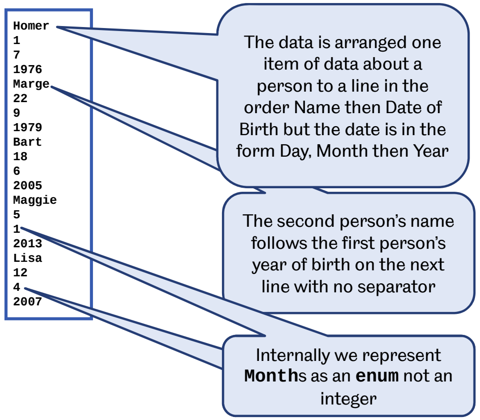
- We can manipulate individual attributes of array elements (**Person** objects) by using an array subscript and the methods of the **Person** class:

```
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`



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

The data is arranged one item of data about a person to a line in the order Name then Date of Birth but the date is in the form Day, Month then Year

The second person's name follows the first person's year of birth on the next line with no separator

Internally we represent `Month`s as an `enum` not an integer

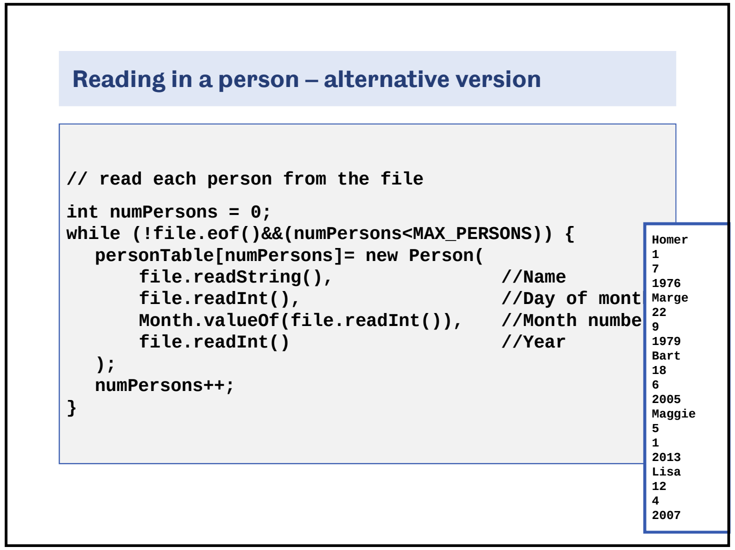
Reading an array of objects from file

```
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++;
   }
```

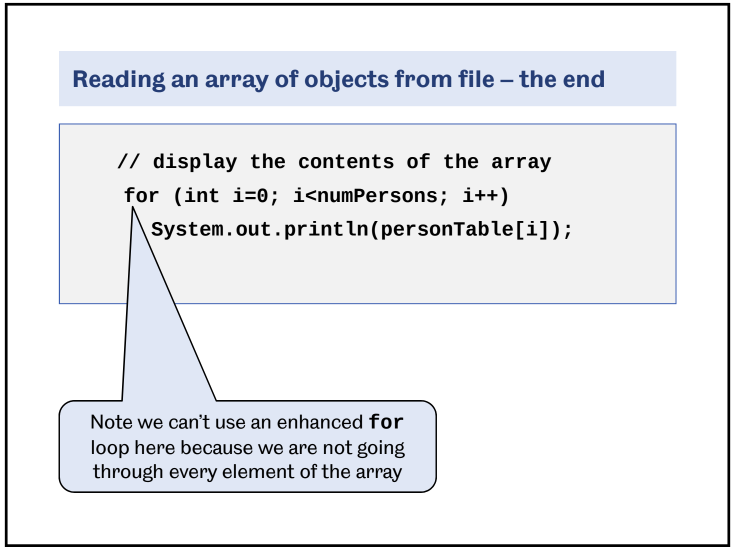
```
Homer
1
7
1976
Marge
22
9
1979
Bart
18
6
2005
Maggie
5
1
2013
Lisa
12
4
2007
```



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++;
}
```

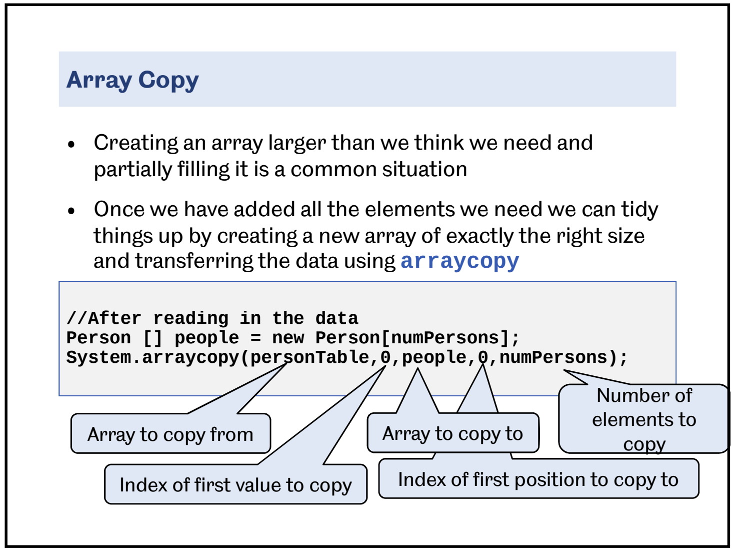
```
Homer
1
7
1976
Marge
22
9
1979
Bart
18
6
2005
Maggie
5
1
2013
Lisa
12
4
2007
```



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

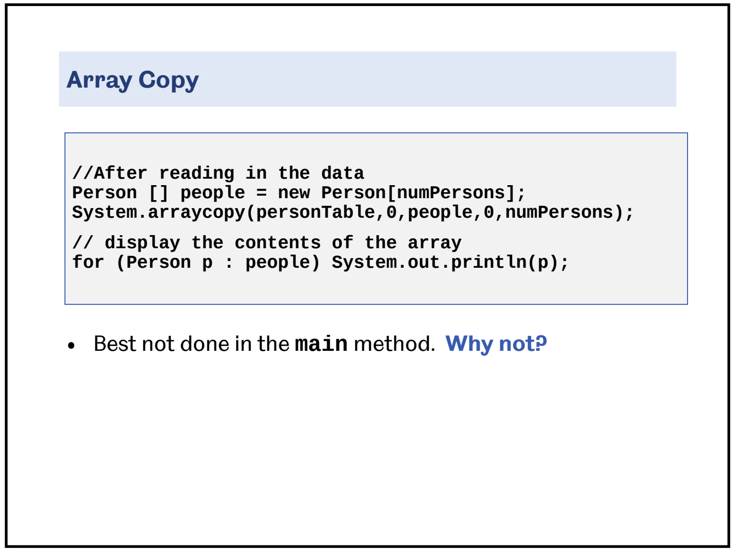


Array Copy

- 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**

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

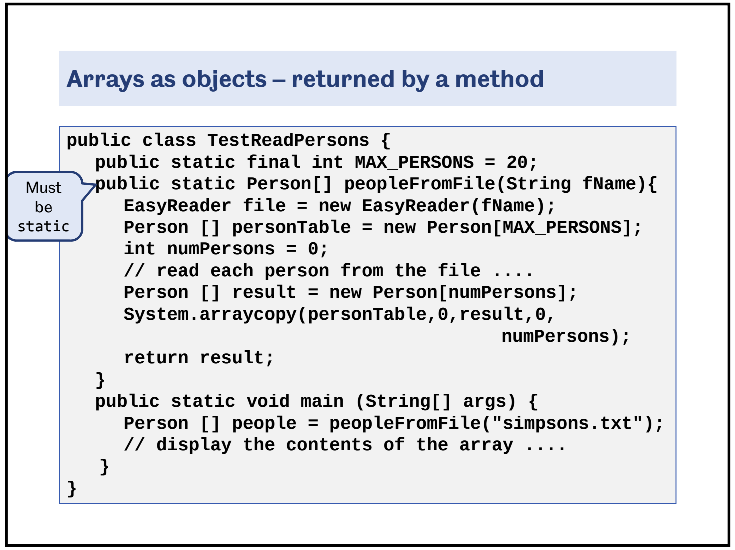
Array to copy from
Index of first value to copy
Array to copy to
Index of first position to copy to
Number of elements to copy



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);
```

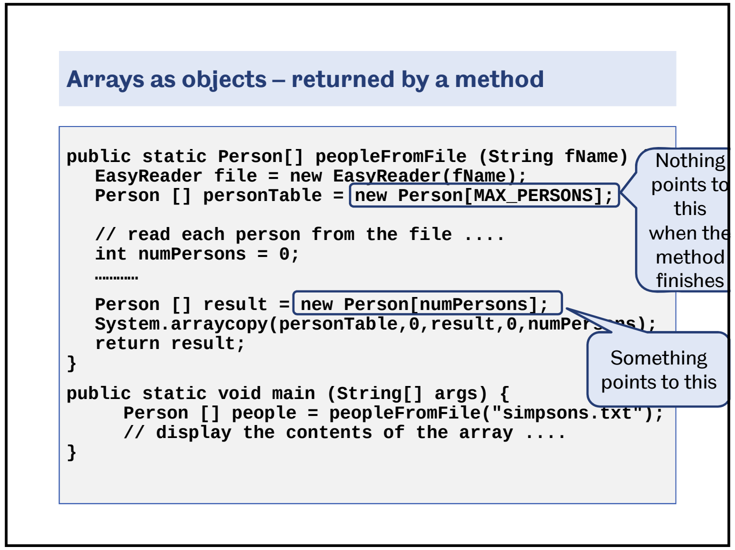
- Best not done in the **main** method. **Why not?**



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

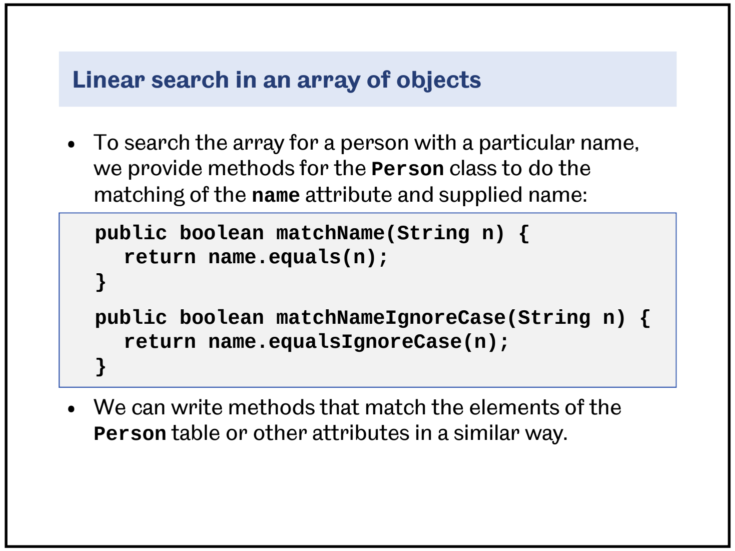


Arrays as objects – returned by a method

```
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

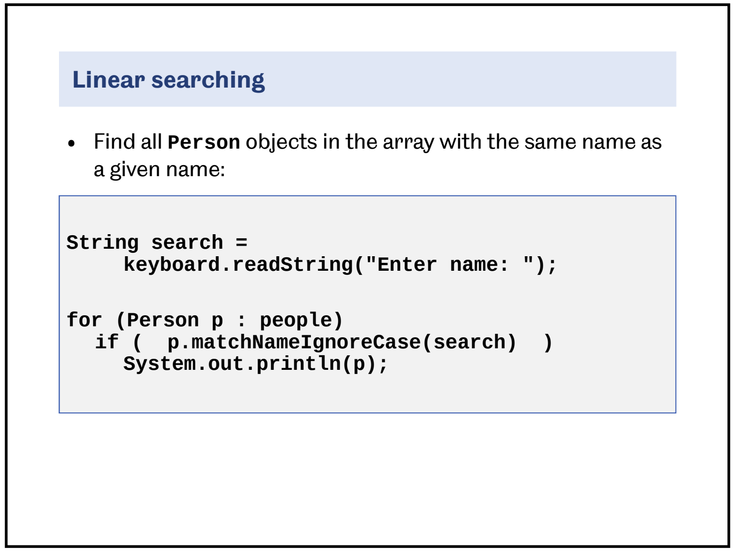


Linear search in an array of objects

- 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);
   }
```

- We can write methods that match the elements of the **Person** table or other attributes in a similar way.



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 by reference.

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

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");
      displayTable(peopleTable);
   }

}
```

Note how the use of methods improves readability

Documenting with `javadoc`

- Classes keep their workings private but they are meant to be usable without knowing how they work.
- Having written a class, we need to document it properly.
- Java provides a tool called **`javadoc`**, which creates HTML documentation from comments.
- Documentation comments begin with `/**` and end with `*/`. Within these symbols, formatting tags can be placed.
- The most useful tags are:

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

Documenting with `javadoc`

The tag

- `@author` is followed by the author's name obviously. It can be used either for a class or a method
- `@param` is used to document a public method and is followed by a parameter's name, then its type and finally what it is there for. There should be one of these for each parameter arranged in the same order as the formal parameters are declared
- `@return` is again used to document public methods. It is only used for methods with a return type which is not void and the tag is followed by the type of whatever is returned and then an explanation of what it represents

Documenting Person

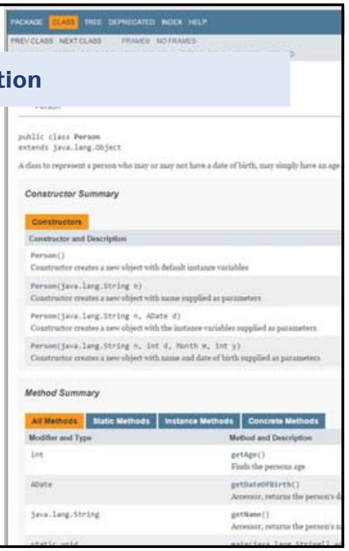
```
/**
* Mutator, changes the person's name
* @param   n      String     The new name
*/
public void setName(String n) { name = n; }
```

```
/**
* Accessor, returns the person's name
* @return    String      The name
*/
public String getName() { return name; }
```

Generating the documentation

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

```
H:\MyJava>javadoc Person.java
```



```
public class Person
extends java.lang.Object
```

A class to represent a person who may or may not have a date of birth, may simply have an age

Constructor Summary

Constructor and Description
`Person()` Constructor creates a new object with default instance variables
`Person(java.lang.String n)` Constructor creates a new object with name supplied as parameters
`Person(java.lang.String n, ADate d)` Constructor creates a new object with the instance variables supplied as parameters
`Person(java.lang.String n, int d, Month m, int y)` Constructor creates a new object with name and date of birth supplied as parameters

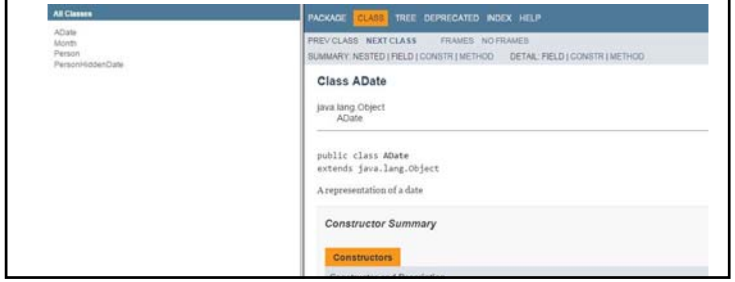
Method Summary

Modifier and Type	Method and Description
`int`	`getAge()` Finds the persons age
`ADate`	`getDateOfBirth()` Accessor, returns the person's d
`java.lang.String`	`getName()` Accessor, returns the person's n

Generating the documentation

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

```
H:\MyJava>javadoc java9\*.java
```



Class ADate

java.lang.Object
ADate

```
public class ADate
extends java.lang.Object
```

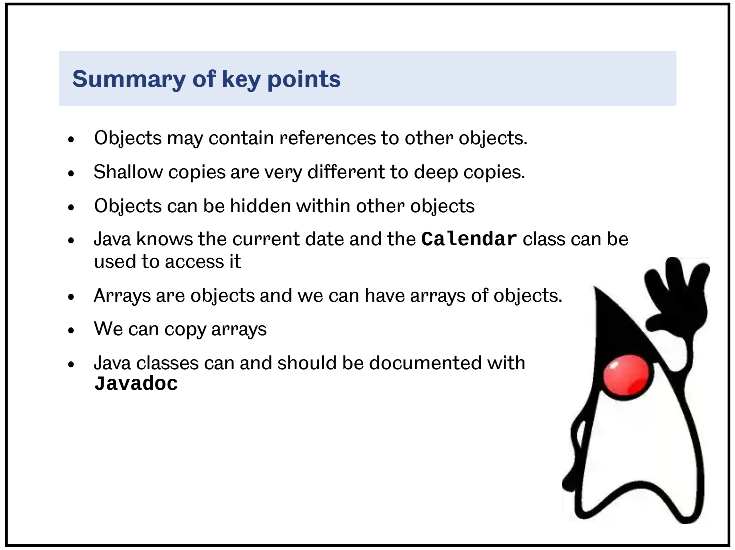
A representation of a date

Constructor Summary

Documenting a package

- You can generate **javadoc** documentation by invoking the following from the **parent** directory:

```
H:\myjava>javadoc packagename
```



Summary of key points

- Objects may contain references to other objects.
- Shallow copies are very different to deep copies.
- Objects can be hidden within other objects
- Java knows the current date and the `Calendar` class can be used to access it
- Arrays are objects and we can have arrays of objects.
- We can copy arrays
- Java classes can and should be documented with `Javadoc`