

List of Slides

- 1 Title
- 2 **Chapter 10:** Separate classes
- 3 Chapter aims
- 5 **Section 2:** Example:Age history with Date class
- 6 Aim
- 7 Age history with Date class
- 8 Class: objects: contain a group of variables
- 9 Age history with Date class
- 10 Class: objects: are instances of a class
- 12 Variable: instance variables
- 14 Age history with Date class
- 15 Age history with Date class
- 16 Method: constructor methods
- 20 Age history with Date class
- 21 The full Date code
- 22 Age history with Date class

- 23 Age history with Date class
- 24 Class: is a type
- 25 Variable: of a class type
- 26 Age history with Date class
- 27 Variable: of a class type: stores a reference to an object
- 29 Type: primitive versus reference
- 30 Age history with Date class
- 31 Class: making instances with new
- 34 Age history with Date class
- 35 Age history with Date class
- 36 Age history with Date class
- 37 Age history with Date class
- 38 Age history with Date class
- 39 Age history with Date class
- 40 Age history with Date class
- 41 Method: accepting parameters: of a class type
- 42 Age history with Date class
- 43 Class: accessing instance variables

44	Age history with Date class
47	Age history with Date class
48	Trying it
49	Coursework: AddQuadPoly
50	Section 3: Improving the Date class: lessThan() and equals() methods
51	Aim
52	Improving the Date class: lessThan() and equals() methods
53	Method: class versus instance methods
60	Improving the Date class: lessThan() and equals() methods
61	Improving the Date class: lessThan() and equals() methods
62	Improving the Date class: lessThan() and equals() methods
63	Improving the Date class: lessThan() and equals() methods
64	Improving the Date class: lessThan() and equals() methods
66	Improving the Date class: lessThan() and equals() methods
67	Improving the Date class: lessThan() and equals() methods
68	Improving the Date class: lessThan() and equals() methods
69	Improving the Date class: lessThan() and equals() methods
70	Improving the Date class: lessThan() and equals() methods

71 Improving the Date class: lessThan() and equals() methods
72 Variable: of a class type: stores a reference to an object: avoid misur
75 Coursework: CompareQuadPoly
76 **Section 4:** Improving the Date class: toString() method
77 Aim
78 Improving the Date class: toString() method
79 Improving the Date class: toString() method
81 Method: a method may have no parameters
82 Improving the Date class: toString() method
83 Improving the Date class: toString() method
88 Coursework: AddQuadPoly and CompareQuadPoly with toString()
89 **Section 5:** Improving the Date class: addYear() method
90 Aim
91 Improving the Date class: addYear() method
92 Variable: instance variables: should be private by default
93 Improving the Date class: addYear() method
95 Method: returning a value: of a class type
98 Improving the Date class: addYear() method

99	Improving the Date class: addYear() method
100	Improving the Date class: addYear() method
101	Type: String: conversion: from object
104	Improving the Date class: addYear() method
106	Improving the Date class: addYear() method
109	Coursework: QuadPoly with an addition method
110	Section 6: Alternative style
111	Aim
112	Class: objects: this reference
113	Alternative style
115	Alternative style
117	Concepts covered in this chapter

Java Just in Time

John Latham

November 6, 2018

Chapter 10

Separate classes

Chapter aims

- All programs up to now have been in one **class**.
- Programs generally consist of more than one class
 - two reasons why discussed already.



Coffee time: Can you remember those two reasons? One of them is about size, and the other is relevant every time we use, say, System Or Math.

Chapter aims

- Look at using a class as template for **constructing objects**
 - used in another class.
- Meet Java technology for **object oriented programming**:
 - **constructor methods**,
 - **instance variables**,
 - **instance methods**.
- OOP helps reduce complexity of sophisticated programs.
 - We revisit age history example.

Section 2

Example:

Age history with Date class

Aim

AIM: To introduce the principle of using more than one **class** in a program, and in particular, the idea of using a class as a template for the **construction** of **objects**. We also introduce **instance variables**, **constructor methods**, creating **new** objects, the fact that a **class** is a **type** and the use of **references**.

Age history with Date class

- Previously used three **variables** to store each date.
- In abstract they were a single date with three parts.
- Ideally, this could be reflected in our program. . . .

Class: objects: contain a group of variables

- Group collection of **variables** into one entity
 - by creating an **object**.
- E.g. represent a point using x and y value.
 - Combine x and y variables into single `Point` object.

Age history with Date class

- Want to combine three variables
 - components of a date
 - into one **object**.
- Need to define a **class** telling Java how to make such objects.

Class: objects: are instances of a class

- Before can make **objects**, need to tell Java how they are **constructed**.
- E.g. to make a `Point` object, need to say
 - will be pair of **variables** inside it called `x` and `y`
 - what **types** they are
 - how they get their values.
- We write a **class** to act as template for creation of objects
 - contains the blue-print / instructions to Java.
- Need a class for each kind of object desired.
 - E.g. have a `Point` class saying how to make `Point` objects or `Chicken` class for making `Chickens`, etc..
 - choose any name we feel is appropriate
 - * (by convention always start with capital letter).

Class: objects: are instances of a class

- Having described template, can ask Java to make objects of that class
 - at **run time**.
- Objects are **instances** of their class.
 - E.g. particular `Point` objects all instances of our `Point` class.
 - Can create many different `Point` objects
 - * each contains own `x` and `y` variables
 - all made from the one template
 - * the `Point` class.

Variable: instance variables

- The **variables** inside **objects** are called **instance variables**
 - belong to **instances** of a **class**.
- Declare like **class variables** but without **reserved word** `static`.
- E.g. part of definition of a `Point` class
 - each `Point` object has two instance variables:

```
public class Point
{
    private double x;
    private double y;
    ...
} // class Point
```

Variable: instance variables

- Instance variables have visibility **modifier**
 - **private** means can only be accessed by code inside class which declared them.
- Class variables belong to class which declared them
 - created at **run time** in the **static context** when class is loaded.
 - Only one copy.
- Instance variables belong to objects
 - created at **run time** dynamically – **dynamic context** –
 - * when object they are part of is created.
 - As many copies as there are instances of the class
 - * each object has its own set of instance variables.

Age history with Date class

- Have Date class as template for creating Date objects.
- Source code stored in **file** Date.java
 - **compiled** separately to produce Date.class byte-code file.
- Have three **instance variables** – three components of a date
 - with **public** visibility
 - * so can be accessed by **main method** in AgeHistory2 class.

Age history with Date class

```
001: // Representation of a date.
002: public class Date
003: {
004:     // The day, month and year of the date.
005:     public int day, month, year;
```

- This tells Java that `Date` objects each have three instance variables.
 - Note lack of `static`.
- Next tell it how to **construct** `Date` objects
 - how do variables get their values?

Method: constructor methods

- A **class** to be used as **object** template should have **constructor method**
 - special kind of **method**
 - containing instructions for **constructing instances** of the class.
- Constructor method has same name as class it is defined in.
- Usually **public**, no **return type** nor **void**.
- Can have **method parameters**,
 - typically initial values for some/all **instance variables**.

Method: constructor methods

- E.g. constructor method for `Point` class with instance variables `x` and `y`:

```
public Point(double requiredX, double requiredY)
{
    x = requiredX;
    y = requiredY;
} // Point
```

- Tells Java in order to construct instance of class `Point`
 - must be given two `double` values
 - first will be placed in `x` instance variable
 - second in `y` instance variable.

Method: constructor methods

- Constructor methods called like other **methods**
 - except precede **method call** with **reserved word** `new`.
- E.g. create `Point` object with `x` and `y` being 7.4 and -19.9.

```
new Point(7.4, -19.9);
```

- Can have as many `Point` objects as we wish.
- Each has own pair of instance variables
 - possibly different values for `x` and `y`.
- E.g. four `Point` objects – coordinates of a rectangle with centre (0, 0).

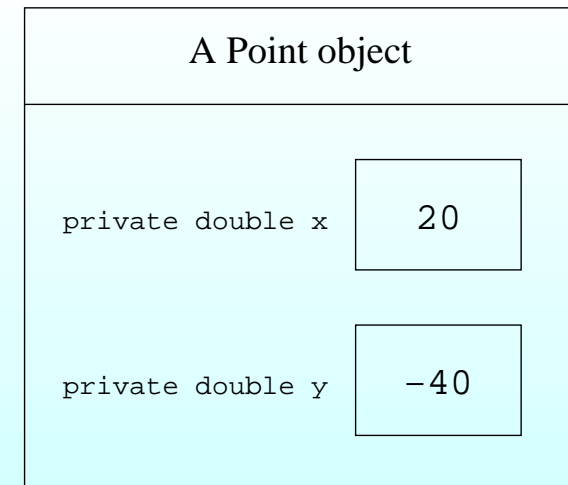
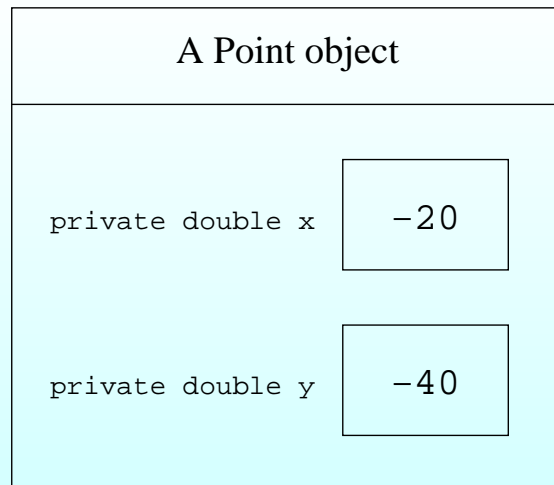
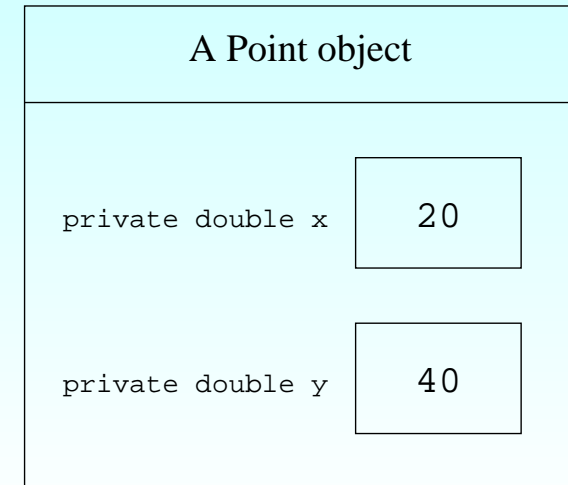
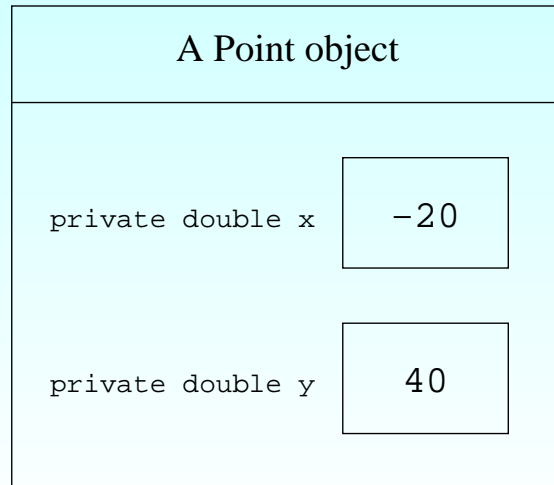
```
new Point(-20, 40);
```

```
new Point(-20, -40);
```

```
new Point(20, 40);
```

```
new Point(20, -40);
```

Method: constructor methods



Age history with Date class

- Date constructor is given three `int` values
 - sets the three instance variables.

```
008: // Construct a date -- given the required day, month and year.
009: public Date(int requiredDay, int requiredMonth, int requiredYear)
010: {
011:     day = requiredDay;
012:     month = requiredMonth;
013:     year = requiredYear;
014: } // Date
015:
016: } // class Date
```

The full Date code

```
001: // Representation of a date.
002: public class Date
003: {
004:     // The day, month and year of the date.
005:     public int day, month, year;
006:
007:
008:     // Construct a date -- given the required day, month and year.
009:     public Date(int requiredDay, int requiredMonth, int requiredYear)
010:     {
011:         day = requiredDay;
012:         month = requiredMonth;
013:         year = requiredYear;
014:     } // Date
015:
016: } // class Date
```

Age history with Date class

- Can compile Date **class**
 - but cannot run it – no **main method**.

Console Input / Output

```
$ javac Date.java
$ java Date
Exception in thread "main" java.lang.NoSuchMethodError: main
$ _
```

Run

Age history with Date class

- Rest of program lives in `AgeHistory2` **class**
 - similar to previous version
 - but uses `Date` class to store dates.
- Will have variables of type `Date`.

Class: is a type

- A **type** is a **set** of values.
 - E.g. `int` is all whole numbers representable in 32 **binary digits**.
 - E.g. `double` is all **real** numbers representable using **double precision**.
 - E.g. `boolean` contains `true` and `false`.
- A **class** can be used as template to create **objects**
 - thus has an associated type:
 - * the set of all objects that can be created – **instances** of that class.
 - E.g. `Point` type is set of all `Point` objects that can be created.

Variable: of a class type

- Can use a **class** much like built-in **types** e.g. `int`, `double` and `boolean`.
- Can declare **variable** whose type is a class.
- E.g. assuming class `Point`:

```
Point p1;
```

```
Point p2;
```

- declares two **local variables** or **method variables** of type `Point`.
- Can also have **class variables** whose type is a class.
 - **instance variables** too.

Age history with Date class

- We have class variable of type Date.

```
001: // Print out an age history of two people.
002: // Arguments: present date, first birth date, second birth date.
003: // Each date is three numbers: day month year.
004: public class AgeHistory2
005: {
006:     // The present date.
007:     private static Date presentDate;
```

- This will store a **reference** to a Date **object** containing the three date components supplied as **command line arguments**.

Variable: of a class type: stores a reference to an object

- For a **variable** of a built-in **primitive type**, Java knows how much memory will be needed for it.
 - E.g. all **int variables** need 4 bytes.
 - E.g. all **double variables** need 8 bytes.
- Java needs this to allocate memory addresses for variables.
- But is not possible to calculate how much memory is needed for **objects**
 - **instances** of different classes will have different sizes
 - sometimes instances of the same class have different sizes!
- Size of an object is only known when it is created, at **run time**.

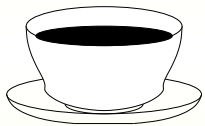
Variable: of a class type: stores a reference to an object

- So, variables of a class type do not store an object
 - instead store a **reference** to an object.
 - * essentially the memory address at which the object resides
 - * only known when object is created at run time.
 - But size of all references is the same
 - * so Java knows how much memory to allocate for a variable of class type.
- Strictly, type associated with a class is **set** of possible *references* to instances of the class.

Type: primitive versus reference

- Every **type** in Java is either **primitive type** or **reference type**.
- Values of primitive types have size known at **compile time**.
- Types for which size of individual values is only known at **run time**
 - e.g. **classes**are called reference types
 - values are always accessed via **reference**.

Age history with Date class



Coffee time: Do you think `String` is a **reference type** or a **primitive type**? Hint: how long is a string?

Class: making instances with new

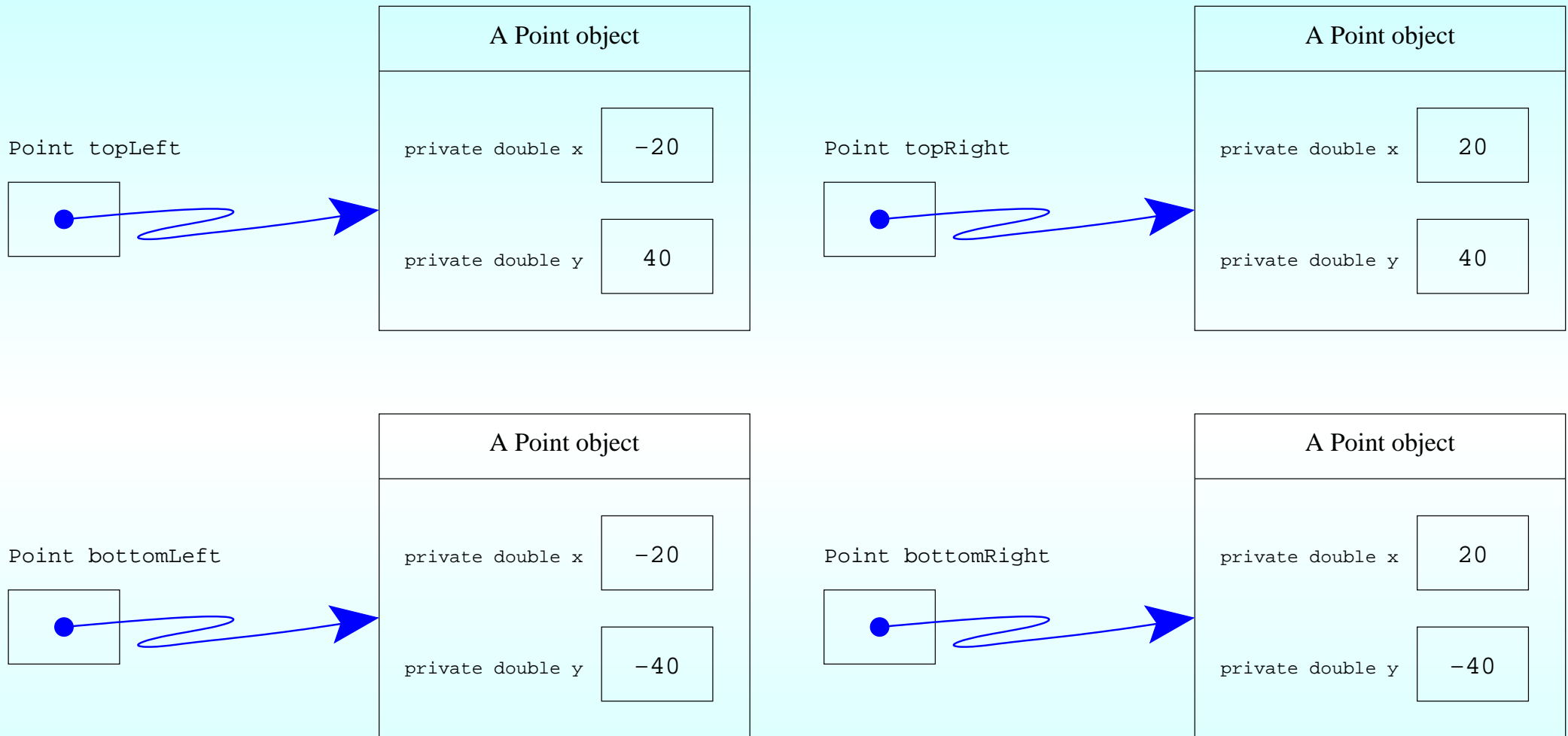
- An **instance** of a **class** created by calling class **constructor method**
 - using **reserved word** `new`
 - supplying **method arguments** for **method parameters**.
- When **executed** at **run time** JVM creates an **object**
 - helped by constructor method code.
- Constructor methods always **return** a value
 - but not stated in heading
 - value is **reference** to **newly** created object.
- Reference can be stored in a **variable**.

- E.g. assuming `Point` class:

```
Point topLeft      = new Point(-20, 40);  
Point bottomLeft  = new Point(-20, -40);  
Point topRight    = new Point(20, 40);  
Point bottomRight = new Point(20, -40);
```

- Declares four `Point` variables.
- Creates four instances of `Point`.
- Diagram...

Class: making instances with new



- Each Point object has two **instance variables**, x and y.

Age history with Date class

- Previously had three class variables
 - each held one `int` component of present date.

```
private static int presentDay
```

01

```
private static int presentMonth
```

07

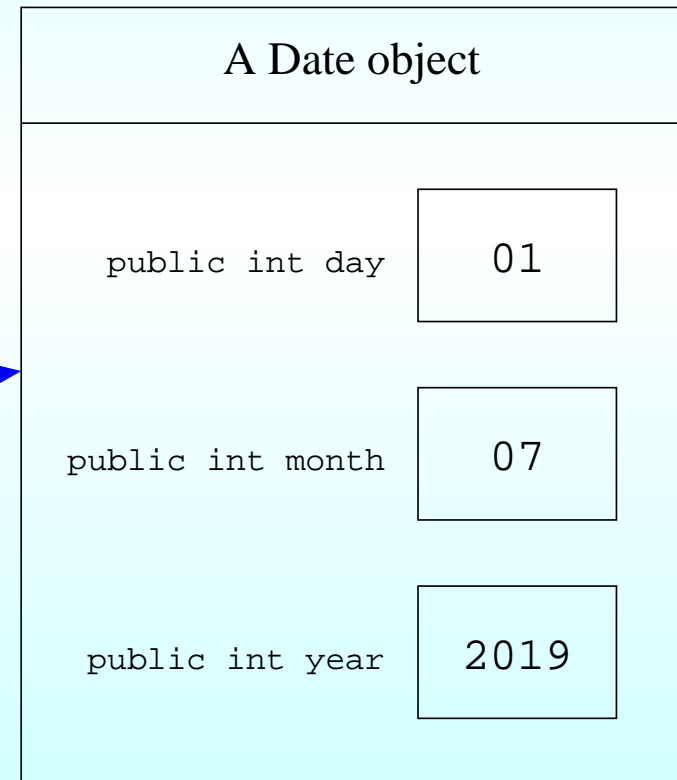
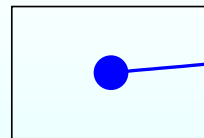
```
private static int presentYear
```

2019

Age history with Date class

- Now have one class variable
 - contains **reference** to a Date object
 - * containing three `int` **instance variables**.

```
private static Date presentDate
```



Age history with Date class

- Put **main method** first to ease discussion.
 - Get first three command line arguments
 - turn to `ints`
 - pass to Date **constructor method**
 - store reference in `presentDate`.

```
010: // The main method: get arguments and call printAgeHistory.
011: public static void main(String[] args)
012: {
013:     // The present date.
014:     presentDate = new Date(Integer.parseInt(args[0]),
015:                             Integer.parseInt(args[1]),
016:                             Integer.parseInt(args[2]));
```

Age history with Date class

- Then
 - do similar with next two argument triples
 - store resulting references in local variables
 - call `printAgeHistory()` method twice.

Age history with Date class

```
018:    // The dates of birth: these must be less than the present date.
019:    Date birthDate1 = new Date(Integer.parseInt(args[3]),
020:                               Integer.parseInt(args[4]),
021:                               Integer.parseInt(args[5]));
022:
023:    Date birthDate2 = new Date(Integer.parseInt(args[6]),
024:                               Integer.parseInt(args[7]),
025:                               Integer.parseInt(args[8]));
026:
027:    // Now print the two age histories.
028:    printAgeHistory(1, birthDate1);
029:    printAgeHistory(2, birthDate2);
030: } // main
```

Age history with Date class

- Use of Date class has improved main method
 - more succinct
 - raised level of **abstraction** w.r.t. dates.

Age history with Date class

- `printAgeHistory()` is now given a single `Date` argument
 - instead of three separate date components.

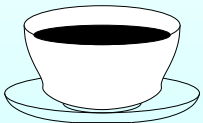
- The **method parameters** of a **method** can be any **type**
 - including **classes**.
 - * must be given **method argument** value of that type
 - * e.g. **reference** to an **object** which is **instance** of class named as parameter type.

Age history with Date class

```
033: // Print the age history of one person, identified as personNumber.  
034: // The birth date must be less than the present date.  
035: private static void printAgeHistory(int personNumber, Date birthDate)  
036: {
```

Class: accessing instance variables

- The **instance variables** of an **object** accessed by
 - taking **reference** to the object
 - appending dot (.)
 - then name of **variable**.
- E.g. if variable `p1` contains reference to a `Point` object
 - `p1.x`
 - * is instance variable `x` belonging to `Point` referred to by `p1`.



Coffee Where else have we seen a dot being used to address
time: something?

Age history with Date class

- `printAgeHistory()` accesses instance variables of `Date` objects
 - referenced by `birthDate` and `presentDate`.

```
037:    // Start by printing the event of birth.
038:    System.out.println("Pn " + personNumber + " was born on "
039:        + birthDate.day + "/" + birthDate.month
040:        + "/" + birthDate.year);
041:
042:    // Now we will go through the years since birth but before today.
043:    int someYear = birthDate.year + 1;
044:    int ageInSomeYear = 1;
```

Age history with Date class

```
045:   while (someYear < presentDate.year
046:       || someYear == presentDate.year
047:       && birthDate.month < presentDate.month
048:       || someYear == presentDate.year
049:       && birthDate.month == presentDate.month
050:       && birthDate.day < presentDate.day)
051:   {
052:       System.out.println("Pn " + personNumber + " was " + ageInSomeYear
053:           + " on " + birthDate.day + "/" + birthDate.month
054:           + "/" + someYear);
055:       someYear++;
056:       ageInSomeYear++;
057:   } // while
058:
```

Age history with Date class

```
059:    // At this point birthDate.day/birthDate.month/someYear
060:    // will be the next birthday, aged ageInSomeYear.
061:    // This will be greater than or equal to the present date.
062:    // If the person has not yet had their birthday this year
063:    // someYear equals presentDate.year,
064:    // otherwise someYear equals presentDate.year + 1.
065:
066:    if (birthDate.month == presentDate.month
067:        && birthDate.day == presentDate.day)
068:        // then someYear must equal presentDate.year.
069:        System.out.println("Pn " + personNumber + " is "
070:                            + ageInSomeYear + " today!");
071:    else
072:        System.out.println("Pn " + personNumber + " will be "
073:                            + ageInSomeYear + " on " + birthDate.day + "/"
074:                            + birthDate.month + "/" + someYear);
075:    } // printAgeHistory
076:
077: } // class AgeHistory2
```

Age history with Date class



*Coffee
time:*

The introduction of a separate `Date` class helped improve part of the program, but has so far made other parts of the program worse than it was when we stored a date using three separate variables! Identify which parts were made better, and which worse.

Trying it

- Get same results as did from previous version.

(Summary only)

Write a **class** to store quadratic polynomials, and a program that adds together two quadratic polynomials to form a third.

Section 3

Improving the Date class: `lessThan()` and `equals()` methods

Aim

AIM: To introduce the concept of **instance methods**. We also look at common misunderstandings about **variables** and **references**.

Improving the Date class: `lessThan()` and `equals()` methods

- The **while loop** and **if else statement conditions** still rather complex
 - comparing two dates using each of the three components.
- Simplify this
 - move date comparison logic to `Date` **class**.

Method: class versus instance methods

- Up to now all **methods** have had `static` in heading
 - can be **executed** in **static context**
 - i.e. used as soon as **class** is loaded.
- Known as **class methods**
 - they belong to the class.
- If omit `static` **modifier** we have an **instance method**
 - can only be run in a **dynamic context**
 - attached to a particular **instance** of the class.

Method: class versus instance methods

- Compare with distinction between **class variables** and **instance variables**
 - one copy of a class variable
 - * created when class is loaded
 - one copy of an instance variable for every instance
 - * created when instance is created.

Method: class versus instance methods

- Class methods belong to the class they are defined in
 - one copy of their code at **run time**
 - * ready for use immediately.
- Instance methods belong to an instance
 - are as many copies of the code at run time as there are instances
 - * ready to run when instance is created.
 - Not really copied, but behaves as though is:
 - * runs in context of the instance it belongs to.

Method: class versus instance methods

- E.g. assume `Point` with instance variables `x` and `y`.
- Might have instance method
 - takes no **method parameters**
 - **returns** distance of a point from origin.
 - * Pythagoras: $\sqrt{x^2 + y^2}$.

```
public double distanceFromOrigin()  
{  
    return Math.sqrt(x * x + y * y);  
} // distanceFromOrigin
```

Method: class versus instance methods

- Class methods accessed by name of class, dot (.) and then name of method.
 - E.g. `Math.sqrt`
- Instance methods belonging to an **object** accessed by
 - take **reference** to the *object*
 - append a dot (.)
 - then name of method.
- E.g. assume `p1` contains reference to a `Point`
 - `p1.distanceFromOrigin()`
 - * invokes instance method `distanceFromOrigin()` belonging to the `Point` referred to by `p1`.

Method: class versus instance methods

- E.g. following produces 5 and 75.

```
Point p1 = new Point(3, 4);
```

```
Point p2 = new Point(45, 60);
```

```
System.out.println(p1.distanceFromOrigin());
```

```
System.out.println(p2.distanceFromOrigin());
```

- First method call uses instance variables of object referred to by p1
 - i.e. values 3 and 4 for x and y.
- Second method call uses instance variables of object referred to by p2
 - i.e. values 45 and 60 for x and y.

Method: class versus instance methods

- E.g. method to determine distance between a point and given other point.

```
public double distanceFromPoint(Point otherPoint)
{
    double xDistance = x - otherPoint.x;
    double yDistance = y - otherPoint.y;

    return Math.sqrt(xDistance * xDistance + yDistance * yDistance);
} // distanceFromPoint
```

- This would print 70.0, twice.

```
System.out.println(p1.distanceFromPoint(p2));
System.out.println(p2.distanceFromPoint(p1));
```


Improving the Date class: `lessThan()` and `equals()` methods

- First part of improved `Date` is the same.

```
001: // Representation of a date.
002: public class Date
003: {
004:     // The day, month and year of the date.
005:     public int day, month, year;
006:
007:
008:     // Construct a date -- given the required day, month and year.
009:     public Date(int requiredDay, int requiredMonth, int requiredYear)
010:     {
011:         day = requiredDay;
012:         month = requiredMonth;
013:         year = requiredYear;
014:     } // Date
```

Improving the Date class: lessThan() and equals() methods

- Instance method:
 - compare this Date **object** with given other one
 - * **return true** iff they are **equivalent**.

```
017: // Compare this date with a given other one, for equality.
018: public boolean equals(Date other)
019: {
020:     return day == other.day && month == other.month && year == other.year;
021: } // equals
```

Improving the Date class: lessThan() and equals() methods

- Instance method:
 - compare this Date **object** with given other one
 - * **return true** iff this one is earlier.

```
024: // Compare this date with a given other one, for less than.
025: public boolean lessThan(Date other)
026: {
027:     return year < other.year
028:         || year == other.year
029:         && (month < other.month
030:             || month == other.month && day < other.day);
031: } // lessThan
032:
033: } // class Date
```

Improving the Date class: lessThan() and equals() methods

Console Input / Output

```
$ javac Date.java  
$ _
```

Run

*Coffee
time:*

What do you think of the following version of lessThan()?

```
public boolean lessThan(Date other)  
{  
    return year < other.year  
        || year == other.year && month < other.month  
        || year == other.year && month == other.month  
        && day < other.day;  
} // lessThan
```

Does it have the same effect as the one in our example?
If so, in what ways is it better or worse?



Improving the Date class: lessThan() and equals() methods

- First part of AgeHistory2 is the same.

```
001: // Print out an age history of two people.
002: // Arguments: present date, first birth date, second birth date.
003: // Each date is three numbers: day month year.
004: public class AgeHistory2
005: {
006:     // The present date.
007:     private static Date presentDate;
008:
009:
010:     // The main method: get arguments and call printAgeHistory.
011:     public static void main(String[] args)
012:     {
013:         // The present date.
014:         presentDate = new Date(Integer.parseInt(args[0]),
015:                                 Integer.parseInt(args[1]),
016:                                 Integer.parseInt(args[2]));
```

Improving the Date class: lessThan() and equals() methods

```
017:
018:     // The dates of birth: these must be less than the present date.
019:     Date birthDate1 = new Date(Integer.parseInt(args[3]),
020:                               Integer.parseInt(args[4]),
021:                               Integer.parseInt(args[5]));
022:
023:     Date birthDate2 = new Date(Integer.parseInt(args[6]),
024:                               Integer.parseInt(args[7]),
025:                               Integer.parseInt(args[8]));
026:
027:     // Now print the two age histories.
028:     printAgeHistory(1, birthDate1);
029:     printAgeHistory(2, birthDate2);
030: } // main
```

Improving the Date class: lessThan() and equals() methods

```
033: // Print the age history of one person, identified as personNumber.
034: // The birth date must be less than the present date.
035: private static void printAgeHistory(int personNumber, Date birthDate)
036: {
037:     // Start by printing the event of birth.
038:     System.out.println("Pn " + personNumber + " was born on "
039:         + birthDate.day + "/" + birthDate.month
040:         + "/" + birthDate.year);
```

Improving the Date class: lessThan() and equals() methods

- Replace **int variable** someYear
 - with Date **variable**, someBirthday.
- And ageInSomeYear renamed ageOnSomeBirthday.

```
042:    // Now we will go through the years since birth but before today.
043:    // We keep track of the birthday we are considering.
044:    Date someBirthday
045:        = new Date(birthDate.day, birthDate.month, birthDate.year + 1);
046:    int ageOnSomeBirthday = 1;
```


Improving the Date class: lessThan() and equals() methods

- While loop **condition** *much* simpler than previously.

```
047:  while (someBirthday.lessThan(presentDate))
048:  {
049:      System.out.println("Pn " + personNumber + " was " + ageOnSomeBirthday
050:                          + " on " + someBirthday.day + "/" + someBirthday.month
051:                          + "/" + someBirthday.year);
052:      someBirthday = new Date(someBirthday.day, someBirthday.month,
053:                             someBirthday.year + 1);
054:      ageOnSomeBirthday++;
055:  } // while
```

Improving the Date class: `lessThan()` and `equals()` methods

- Each time round **loop, reference** in `someBirthday` is changed
 - refers to a **new** `Date` representing next birthday.

Improving the Date class: lessThan() and equals() methods

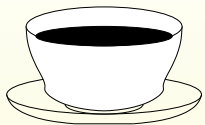
- If else statement **condition** also much simpler than previously.

```
057:    // Now deal with the next birthday.
058:    if (someBirthday.equals(presentDate))
059:        System.out.println("Pn " + personNumber + " is "
060:                               + ageOnSomeBirthday + " today!");
061:    else
062:        System.out.println("Pn " + personNumber + " will be "
063:                               + ageOnSomeBirthday + " on " + someBirthday.day
064:                               + "/" + someBirthday.month + "/" + someBirthday.year);
065:    } // printAgeHistory
066:
067: } // class AgeHistory2
```

Improving the Date class: `lessThan()` and `equals()` methods



Coffee time: Java would have permitted us to write the condition of the if else statement as `someBirthday == presentDate`. Why would this not work?



Coffee time: The introduction of instance methods has helped improve more of the program. What aspects still have room for improvement?

Variable: of a class type: stores a reference to an object: avoid misunderstanding

- Two common mistakes:
 1. Misconception: A **variable** is an **object**.
 2. Misconception: A variable contains an object.
- Neither: variables of a **class type** can contain a *reference* to an object.
- Common question – “why do we have to write `Date` twice in following?”.

```
Date someBirthday  
= new Date(birthDate.day, birthDate.month, birthDate.year + 1);
```

- Because doing three things:
 1. Declaring a variable,
 2. **constructing** an object,
 3. storing reference to object in the variable.

Variable: of a class type: stores a reference to an object: avoid misunderstanding

- Can have variable without object.

```
Date someBirthday;
```

- Can have object without variable (useful?)

```
new Date(birthDate.day, birthDate.month, birthDate.year + 1);
```

```
System.out.println(new Point(3, 4).distanceFromPoint(new Point(45, 60)));
```

- Can have two variables referring to same object.

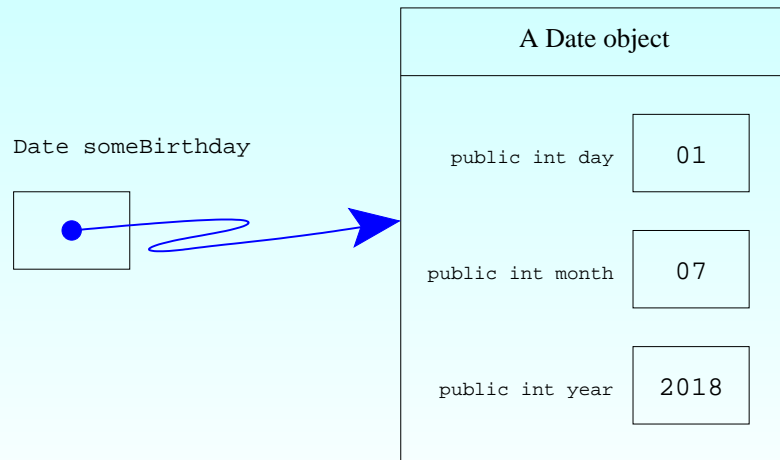
```
Date theSameBirthday = someBirthday;
```

- Can change value of variable – make it refer to different object.

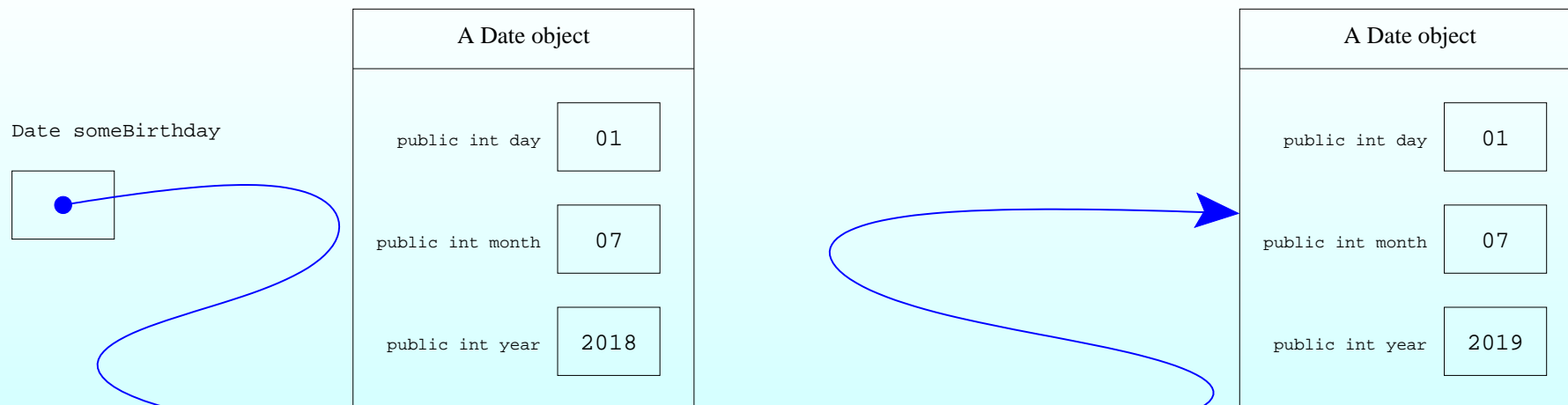
```
someBirthday = new Date(someBirthday.day, someBirthday.month,  
                        someBirthday.year + 1);
```

Diagram...

Variable: of a class type: stores a reference to an object: object: avoid misunderstanding



```
someBirthday = new Date(someBirthday.day, someBirthday.month, someBirthday.year + 1);
```



(Summary only)

Extend a **class** that stores quadratic polynomials, and write a program that compares the 'size' of two quadratic polynomials.

Section 4

Improving the Date class: toString() method

Aim

AIM: To reinforce the concept of **instance methods**. We also note that a **method** might have no **method parameters**.

- Continue to improve AgeHistory2
 - several places where we print a date
 - put logic for producing a date string in Date **class**
 - * as **instance method**.
 - Use in AgeHistory2 class.
- First part of Date is the same.

Improving the Date class: toString() method

```
001: // Representation of a date.
002: public class Date
003: {
004:     // The day, month and year of the date.
005:     public int day, month, year;
006:
007:
008:     // Construct a date -- given the required day, month and year.
009:     public Date(int requiredDay, int requiredMonth, int requiredYear)
010:     {
011:         day = requiredDay;
012:         month = requiredMonth;
013:         year = requiredYear;
014:     } // Date
015:
016:
```

Improving the Date class: toString() method

```
017: // Compare this date with a given other one, for equality.
018: public boolean equals(Date other)
019: {
020:     return day == other.day && month == other.month && year == other.year;
021: } // equals
022:
023:
024: // Compare this date with a given other one, for less than.
025: public boolean lessThan(Date other)
026: {
027:     return year < other.year
028:         || year == other.year
029:         && (month < other.month
030:             || month == other.month && day < other.day);
031: } // lessThan
```

Method: a method may have no parameters

- List of **method parameters** may be empty.
 - E.g. if method always has same effect / **returns** same result.
 - E.g. if result depends solely on values of **instance variables**
 - * rather than values in context of method call.

Improving the Date class: toString() method

```
034: // Return the day/month/year representation of the date.
035: public String toString()
036: {
037:     return day + "/" + month + "/" + year;
038: } // toString
039:
040: } // class Date
```

Console Input / Output

```
$ javac Date.java
$_
```

Run

Improving the Date class: toString() method

- Use toString() **instance method** when need to print a Date.

```
001: // Print out an age history of two people.
002: // Arguments: present date, first birth date, second birth date.
003: // Each date is three numbers: day month year.
004: public class AgeHistory2
005: {
006:     // The present date.
007:     private static Date presentDate;
008:
009:
010:     // The main method: get arguments and call printAgeHistory.
011:     public static void main(String[] args)
012:     {
013:         // The present date.
014:         presentDate = new Date(Integer.parseInt(args[0]),
015:                                 Integer.parseInt(args[1]),
016:                                 Integer.parseInt(args[2]));
```


Improving the Date class: toString() method

```
017:
018:     // The dates of birth: these must be less than the present date.
019:     Date birthDate1 = new Date(Integer.parseInt(args[3]),
020:                               Integer.parseInt(args[4]),
021:                               Integer.parseInt(args[5]));
022:
023:     Date birthDate2 = new Date(Integer.parseInt(args[6]),
024:                               Integer.parseInt(args[7]),
025:                               Integer.parseInt(args[8]));
026:
027:     // Now print the two age histories.
028:     printAgeHistory(1, birthDate1);
029:     printAgeHistory(2, birthDate2);
030: } // main
031:
032:
```

Improving the Date class: toString() method


```
033: // Print the age history of one person, identified as personNumber.
034: // The birth date must be less than the present date.
035: private static void printAgeHistory(int personNumber, Date birthDate)
036: {
037:     // Start by printing the event of birth.
038:     System.out.println("Pn " + personNumber + " was born on "
039:         + birthDate.toString());
040:
```

Improving the Date class: toString() method

```
041:    // Now we will go through the years since birth but before today.
042:    // We keep track of the birthday we are considering.
043:    Date someBirthday
044:        = new Date(birthDate.day, birthDate.month, birthDate.year + 1);
045:    int ageOnSomeBirthday = 1;
046:    while (someBirthday.lessThan(presentDate))
047:    {
048:        System.out.println("Pn " + personNumber + " was " + ageOnSomeBirthday
049:            + " on " + someBirthday.toString());
050:        someBirthday = new Date(someBirthday.day, someBirthday.month,
051:            someBirthday.year + 1);
052:        ageOnSomeBirthday++;
053:    } // while
054:
```

Improving the Date class: toString() method

```
055:    // Now deal with the next birthday.
056:    if (someBirthday.equals(presentDate))
057:        System.out.println("Pn " + personNumber + " is "
058:                            + ageOnSomeBirthday + " today!");
059:    else
060:        System.out.println("Pn " + personNumber + " will be "
061:                            + ageOnSomeBirthday + " on "
062:                            + someBirthday.toString());
063:    } // printAgeHistory
064:
065: } // class AgeHistory2
```



Coursework: AddQuadPoly and CompareQuadPoly with toString()

(Summary only)

Extend a **class** that stores quadratic polynomials, and modify programs that add together, and compare the 'size' of, two quadratic polynomials.

Section 5

Improving the Date class: addYear () method

Aim

AIM: To further reinforce **instance methods**, meet Java's `toString()` convention and focus on the visibility of **instance variables**. We also see a **return type** which is a **class**.

Improving the Date class: addYear() method

- Three more things to do.
 - Another **instance method** for Date to create **new** Date one year later.
 - * When use in AgeHistory2 there will be no direct accesses to Date **instance variables** left outside Date.
 - Make Date instance variables **private**.
 - Use Java's implicit toString() mechanism.

Variable: instance variables: should be private by default

- Generally best to make **instance variables private**
 - permits us to change way **class** works without affecting code in other classes.
- E.g. Point class:
 - *might* decide to reimplement – have instance variables for polar coordinate radius and angle instead of x and y .
 - Safe because x and y instance variables were private
 - * no other code could possibly be using them.
 - Still have same public interface
 - * **constructor method** would convert given x and y to polar
 - * `toString()` would convert them back.



Coffee How might we store dates differently, instead of using day,
time: month and year?

Improving the Date class: addYear() method

```
001: // Representation of a date.
002: public class Date
003: {
004:     // The day, month and year of the date.
005:     private int day, month, year;
```

- Next part is the same.

```
008:     // Construct a date -- given the required day, month and year.
009:     public Date(int requiredDay, int requiredMonth, int requiredYear)
010:     {
011:         day = requiredDay;
012:         month = requiredMonth;
013:         year = requiredYear;
014:     } // Date
015:
016:
```

Improving the Date class: addYear() method

```
017: // Compare this date with a given other one, for equality.
018: public boolean equals(Date other)
019: {
020:     return day == other.day && month == other.month && year == other.year;
021: } // equals
022:
023:
024: // Compare this date with a given other one, for less than.
025: public boolean lessThan(Date other)
026: {
027:     return year < other.year
028:         || year == other.year
029:         && (month < other.month
030:             || month == other.month && day < other.day);
031: } // lessThan
032:
033:
034: // Return the day/month/year representation of the date.
035: public String toString()
036: {
037:     return day + "/" + month + "/" + year;
038: } // toString
```

Method: returning a value: of a class type

- A **method** may have a **class** as its **return type**
 - so it typically **returns** a **reference** to an **instance** of that class.

Method: returning a value: of a class type

- E.g. assuming a Point class:

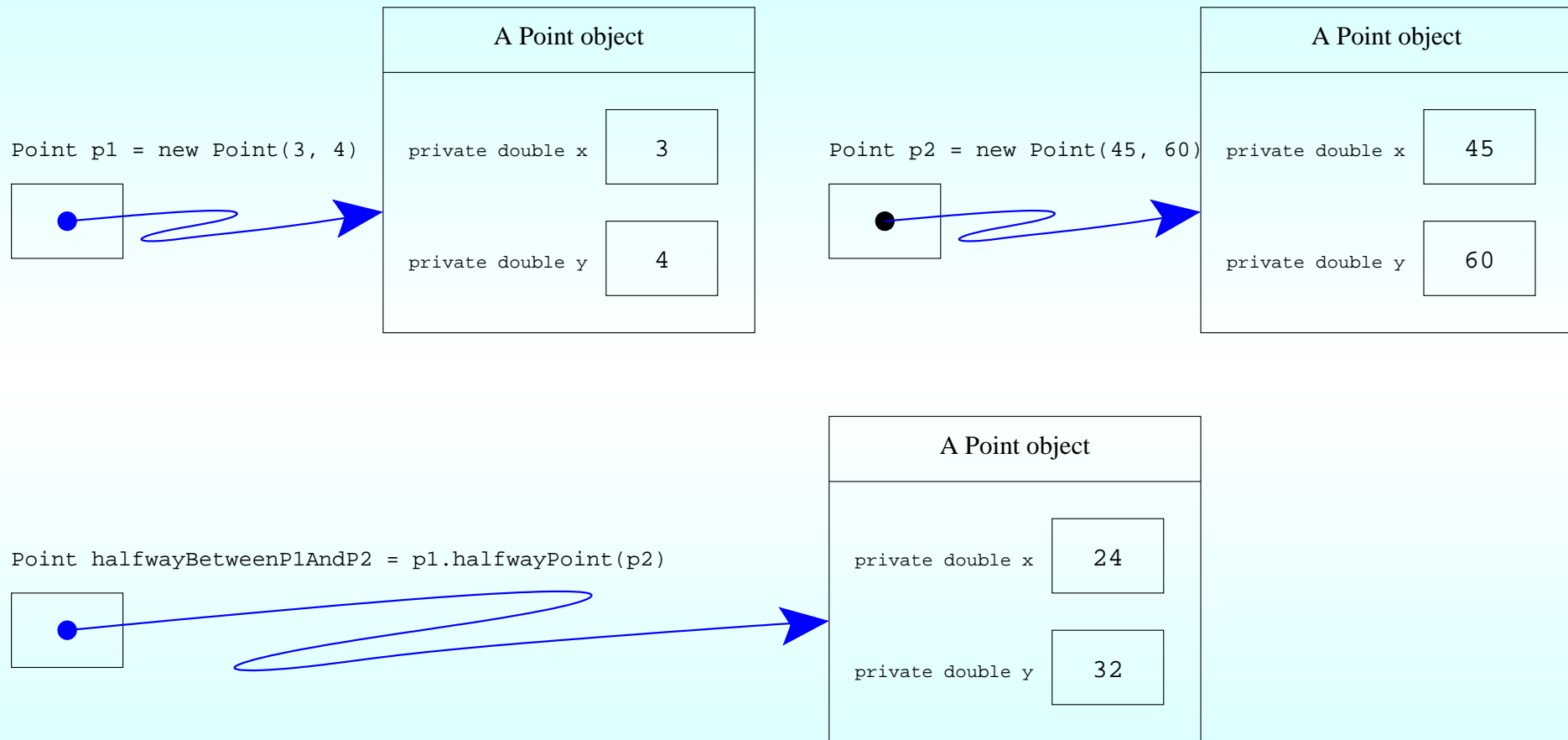
```
public Point halfWayPoint(Point otherPoint)
{
    double newX = (x + otherPoint.x) / 2;
    double newY = (y + otherPoint.y) / 2;
    return new Point(newX, newY);
} // halfWayPoint
```

- Perhaps used as:

```
Point p1 = new Point(3, 4);
Point p2 = new Point(45, 60);

Point halfWayBetweenP1AndP2 = p1.halfWayPoint(p2);
```

Method: returning a value: of a class type



Improving the Date class: addYear () method

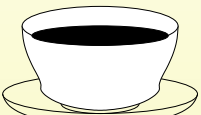
*Coffee
time:*

Suppose the `Point` class in the concept above had an `equals ()` instance method, defined as follows.

```
public boolean equals(Point other)
{
    return x == other.x && y == other.y;
} // equals
```

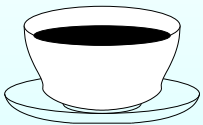
Predict the values of `equalByReference` and `equalByMethod`.

```
Point halfWayBetweenP1AndP2 = p1.halfWayPoint(p2);
Point halfWayBetweenP2AndP1 = p2.halfWayPoint(p1);
boolean equalByReference
    = halfWayBetweenP1AndP2 == halfWayBetweenP2AndP1;
boolean equalByMethod
    = halfWayBetweenP1AndP2.equals(halfWayBetweenP2AndP1);
```



Coffee What are the values of the same **variables** after the following *alternative* code is run?

```
Point halfWayBetweenP1AndP2 = p1.halfWayPoint(p2);  
Point halfWayBetweenP2AndP1 = halfWayBetweenP1AndP2;  
boolean equalByReference  
    = halfWayBetweenP1AndP2 == halfWayBetweenP2AndP1;  
boolean equalByMethod  
    = halfWayBetweenP1AndP2.equals(halfWayBetweenP2AndP1);
```



Improving the Date class: addYear() method

```
041: // Return a new Date which is one year later than this one.
042: public Date addYear()
043: {
044:     return new Date(day, month, year + 1);
045: } // addYear
046:
047: } // class Date
```

Console Input / Output

```
$ javac Date.java
$ _
```

Run

Type: String: conversion: from object

- Commonly wish to have an **instance method** to produce a `String` representation of an **object**.
- By convention always called `toString`.
- E.g. for `Point` class:

```
public String toString()  
{  
    return "(" + x + "," + y + ")";  
} // toString
```

Type: string: conversion: from object

- When **compiler** finds **object reference** as **concatenation operand**
 - it assumes we wish `toString()` to be called.

- E.g.

```
Point p1 = new Point(10, 40);  
System.out.println("The point is " + p1.toString());
```

could/should be written:

```
Point p1 = new Point(10, 40);  
System.out.println("The point is " + p1);
```

Type: String: conversion: from object

- Also, for convenience, separate version of `System.out.println()`:

```
System.out.println(p1);
```

same effect as:

```
System.out.println("" + p1);
```

Improving the Date class: addYear() method

```
001: // Print out an age history of two people.
002: // Arguments: present date, first birth date, second birth date.
003: // Each date is three numbers: day month year.
004: public class AgeHistory2
005: {
006:     // The present date.
007:     private static Date presentDate;
008:
009:
010:     // The main method: get arguments and call printAgeHistory.
011:     public static void main(String[] args)
012:     {
013:         // The present date.
014:         presentDate = new Date(Integer.parseInt(args[0]),
015:                                 Integer.parseInt(args[1]),
016:                                 Integer.parseInt(args[2]));
017:
```

Improving the Date class: addYear() method

```
018:    // The dates of birth: these must be less than the present date.
019:    Date birthDate1 = new Date(Integer.parseInt(args[3]),
020:                               Integer.parseInt(args[4]),
021:                               Integer.parseInt(args[5]));
022:
023:    Date birthDate2 = new Date(Integer.parseInt(args[6]),
024:                               Integer.parseInt(args[7]),
025:                               Integer.parseInt(args[8]));
026:
027:    // Now print the two age histories.
028:    printAgeHistory(1, birthDate1);
029:    printAgeHistory(2, birthDate2);
030: } // main
```

Improving the Date class: addYear() method

- Implicit use of toString().
- Explicit use of addYear().

```
033: // Print the age history of one person, identified as personNumber.
034: // The birth date must be less than the present date.
035: private static void printAgeHistory(int personNumber, Date birthDate)
036: {
037:     // Start by printing the event of birth.
038:     System.out.println("Pn " + personNumber + " was born on " + birthDate);
039:
```

Improving the Date class: addYear() method

```
040:    // Now we will go through the years since birth but before today.
041:    // We keep track of the birthday we are considering.
042:    Date someBirthday = birthDate.addYear();
043:    int ageOnSomeBirthday = 1;
044:    while (someBirthday.lessThan(presentDate))
045:    {
046:        System.out.println("Pn " + personNumber + " was " + ageOnSomeBirthday
047:            + " on " + someBirthday);
048:        someBirthday = someBirthday.addYear();
049:        ageOnSomeBirthday++;
050:    } // while
051:
```


Improving the Date class: addYear() method

```
052:    // Now deal with the next birthday.
053:    if (someBirthday.equals(presentDate))
054:        System.out.println("Pn " + personNumber + " is "
055:            + ageOnSomeBirthday + " today!");
056:    else
057:        System.out.println("Pn " + personNumber + " will be "
058:            + ageOnSomeBirthday + " on " + someBirthday);
059:    } // printAgeHistory
060:
061: } // class AgeHistory2
```

- Compare this version with the original year-only version.

Coffee time: What do you think would happen if we place, as an **operand** of **concatenation**, a **reference** to an object belonging to a class in which we have not defined a `toString()` instance method?



(Summary only)

Further extend a **class** that stores quadratic polynomials, and modify a program that adds together two quadratic polynomials.

Section 6

Alternative style

Aim

AIM: To show an alternative way of talking about **instance variables** and **instance methods** from within the same **class**, using `this`.

Class: objects: this reference

- In **constructor methods** or **instance methods** sometimes wish to talk about the **object** being created or to which the instance method belongs.
- The **reserved word** `this` in an **expression** is such a **this reference**.
- E.g.

```
public Point halfThisPoint()  
{  
    return halfWayPoint(new Point(0, 0));  
} // halfThisPoint
```

- Alternatively:

```
public Point halfThisPoint()  
{  
    return new Point(0, 0).halfWayPoint(this);  
} // halfThisPoint
```

- This author prefers use of **this reference** only when needed.
- Some like following style:

```
001: // Representation of a date.  
002: public class Date  
003: {  
004:     // The day, month and year of the date.  
005:     private int day, month, year;  
006:  
007:
```

Alternative style

```
008: // Construct a date -- given the required day, month and year.
009: public Date(int day, int month, int year)
010: {
011:     this.day = day;
012:     this.month = month;
013:     this.year = year;
014: } // Date
```

- Note same names for **method parameters** and instance variables.

Alternative style

```
017: // Compare this date with a given other one, for equality.
018: public boolean equals(Date other)
019: {
020:     return this.day == other.day && this.month == other.month
021:         && this.year == other.year;
022: } // equals
023:
024:
025: // Compare this date with a given other one, for less than.
026: public boolean lessThan(Date other)
027: {
028:     return this.year < other.year
029:         || this.year == other.year
030:         && (this.month < other.month
031:             || this.month == other.month && this.day < other.day);
032: } // lessThan
033:
```


Alternative style

```
034:
035:  // Return the day/month/year representation of the date.
036:  public String toString()
037:  {
038:      return this.day + "/" + this.month + "/" + this.year;
039:  } // toString
040:
041:
042:  // Return a new Date which is one year later than this one.
043:  public Date addYear()
044:  {
045:      return new Date(this.day, this.month, this.year + 1);
046:  } // addYear
047:
048: } // class Date
```

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
 - a self-test question,
 - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.