Inf1-OP

Creating Classes

Volker Seeker, adapting earlier version by Perdita Stevens and Ewan Klein

School of Informatics

February 6, 2019

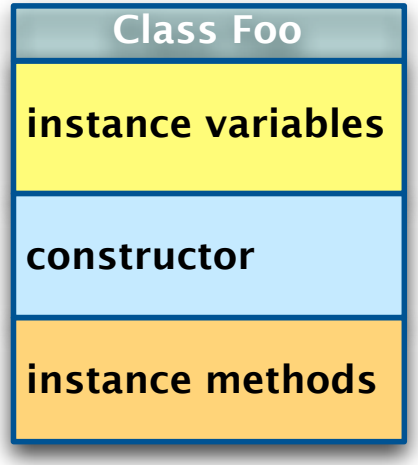
Creating classes

Last time we saw how to use a class:

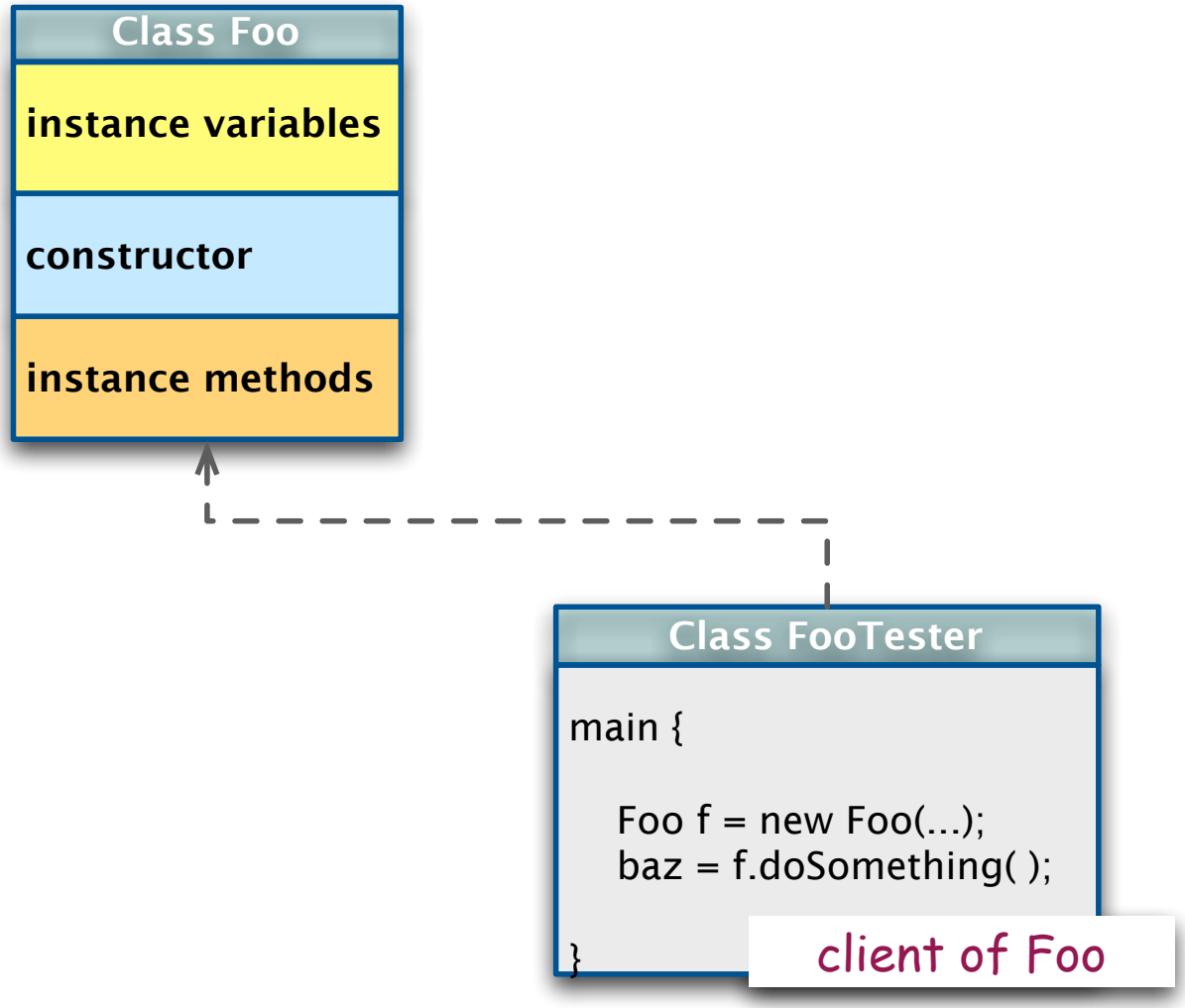
- create a new object, using `new`;
- send the object messages from its interface, to invoke its behaviour;
- we understood that the object might change its state;
- and that state and behaviour interdepend;
- but we did not expect to have access to the state, and we did not know or need to care exactly how the behaviour was implemented.

This time we will see how to define a class, including its state and behaviour, and how new objects should be created.

Classes and Clients



Classes and Clients



Classes and Clients

Client code:

- In general, a client program calls a method of some class `C`.
- Example: class `FooTester` is a client of `Foo` because it calls the `doSomething()` instance method on `Foo` objects.

Classes and Clients

Client code:

- In general, a client program calls a method of some class `C`.
- Example: class `FooTester` is a client of `Foo` because it calls the `doSomething()` instance method on `Foo` objects.

Test-first design methodology:

1. Think about the methods a client would call on instances of class `C`.
2. Design the API for class `C`.
3. Implement a client `CTester` for `C` which tests the desired behaviour.
4. Implement `C` so that it satisfies `CTester`.

CircleTester

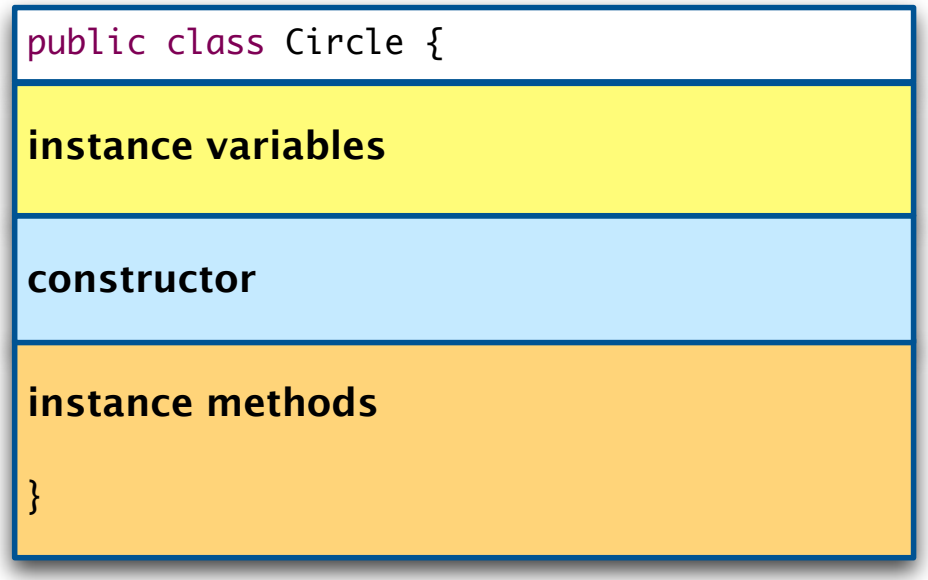
- Create a `Circle` object `c1`.
- Call a method to get the area of that object: `c1.getArea()`

```
public class CircleTester {

    public static void main(String[] args) {
        Circle c1 = new Circle();
        double area1 = c1.getArea();
        System.out.printf("Area of circle c1 is %5.2f\n", area1);

        Circle c2 = new Circle(5.0);
        double area2 = c2.getArea();
        System.out.printf("Area of circle c2 is %5.2f\n", area2);
    }
}
```

Expected Output

```
% java CircleTester
Area of circle c1 is  3.14
Area of circle c2 is 78.54
```

The `Circle` Class



The `Circle` Class: Instance Methods

```
public class Circle {
```

instance variables

constructor

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

The `Circle` Class: Instance Methods

```
public class Circle {
```

instance variables

constructor

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

- `getArea()` is an instance method of the class `Circle`.
- How does it know about `radius`?

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

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

- `radius` is an instance variable of the class `Circle`.

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

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

- `radius` is an instance variable of the class `Circle`.
- Instance variables are declared outside methods and have scope over the whole class.

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

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

- `radius` is an instance variable of the class `Circle`.
- Instance variables are declared outside methods and have scope over the whole class.
- An instance method of a class can use any instance variable of that class.

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

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

- `radius` is an instance variable of the class `Circle`.
- Instance variables are declared outside methods and have scope over the whole class.
- An instance method of a class can use any instance variable of that class.
- Instance variables do not have to be initialised; they get default values (e.g., 0 for int, `false` for boolean, `null` for all reference types).

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

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

- `radius` is an instance variable of the class `Circle`.
- Instance variables are declared outside methods and have scope over the whole class.
- An instance method of a class can use any instance variable of that class.
- Instance variables do not have to be initialised; they get default values (e.g., 0 for int, `false` for boolean, `null` for all reference types).
- How does a `Circle` object's radius get set?

The `Circle` Class: Constructors

```
public class Circle {

    private double radius;

    public Circle(double newRadius){
        radius = newRadius;
    }

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

Constructor

- has same name as the class;

The `Circle` Class: Constructors

```
public class Circle {

    private double radius;

    public Circle(double newRadius){
        radius = newRadius;
    }

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

Constructor

- has same name as the class;
- used to initialise an object that has been created: `new Circle(5.0);`

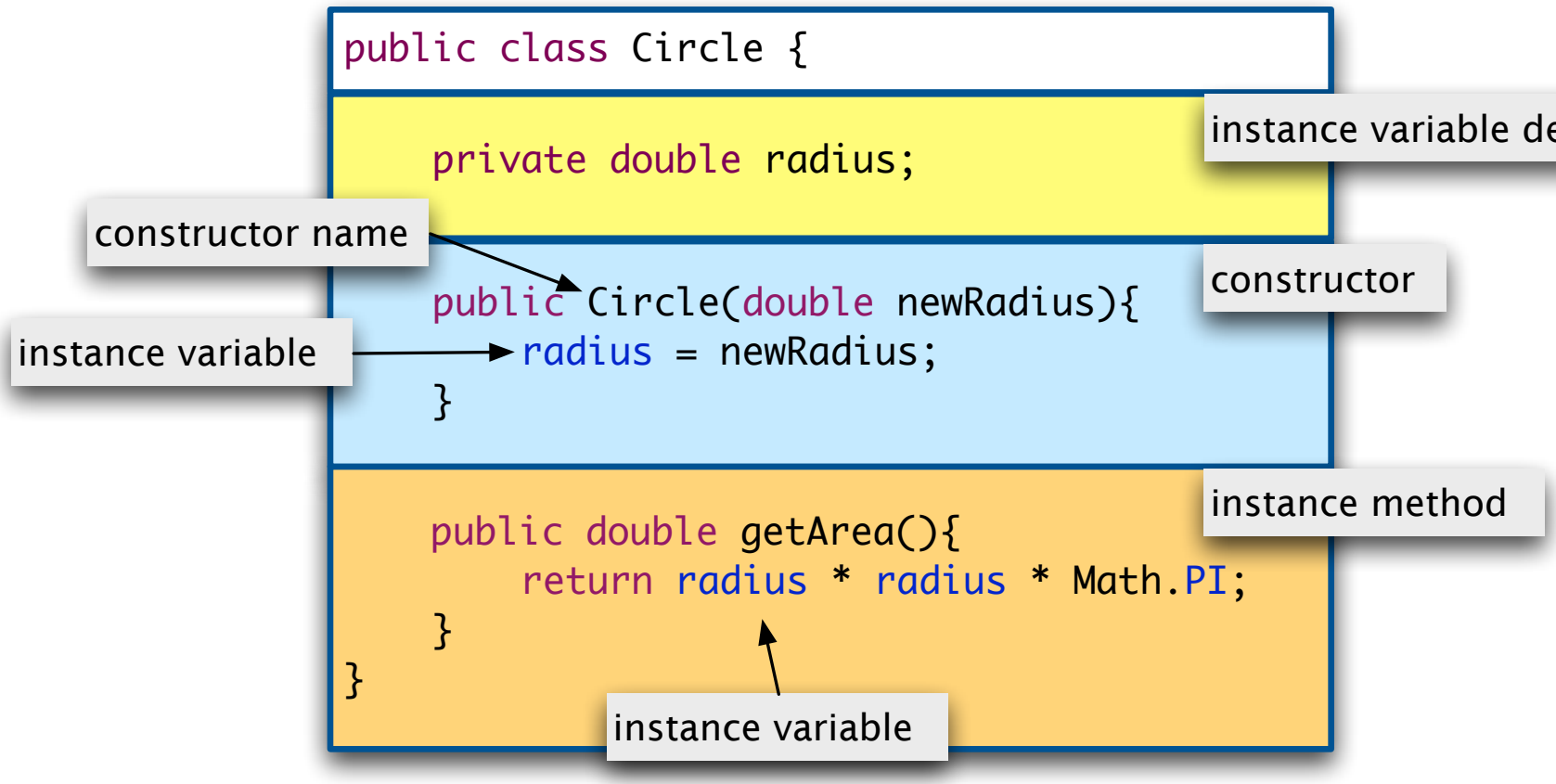
The `Circle` Class: Constructors

```
public class Circle {

    private double radius;

    public Circle(double newRadius){
        radius = newRadius;
    }

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

Constructor

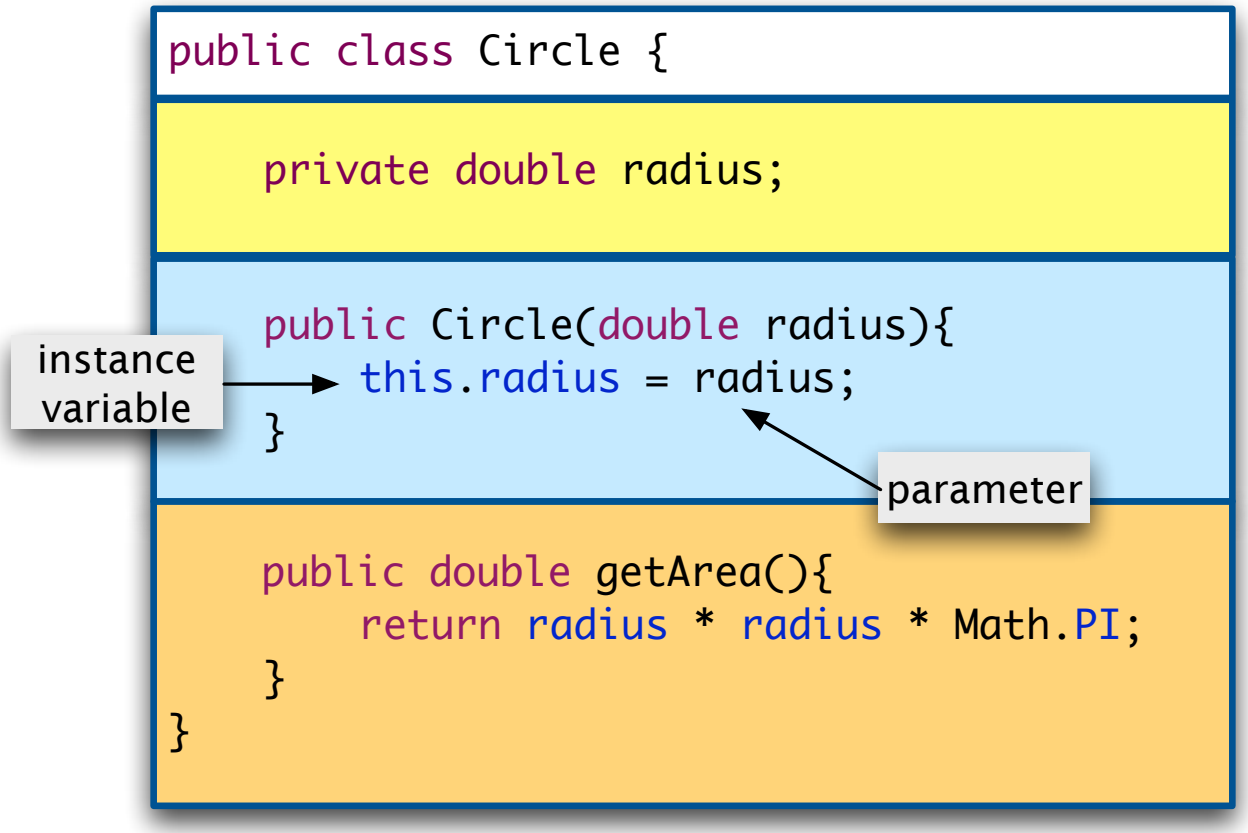
- has same name as the class;
- used to initialise an object that has been created: `new Circle(5.0);`
- must not have a return type (not even `void`).

The `Circle` Class: Anatomy



The `Circle` Class: Constructors

Alternative notation:



The Circle Class: Client

```java
public class Circle {

    private double radius;

    public Circle(double radius){
        this.radius = radius;
    }

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

Class CircleTester

```java
public static void main(String[] args) {
    Circle c1 = new Circle(1.0);
    double area1 = c1.getArea();
    System.out.printf("Area of circle c1 is %5.2f\n", area1);

    Circle c2 = new Circle(5.0);
    double area2 = c2.getArea();
    System.out.printf("Area of circle c2 is %5.2f\n", area2);
}
```

client of Circle

Interim Summary

We looked at:

- using client programs to motivate our classes, and to test them
- instance variables:
  - represent data that is particular to an object (i.e., an instance!);
  - have scope over the whole class;
  - can hold mutable state;
  - can be manipulated by any instance method in the class.
- instance methods:
  - like static methods, but can only be called on some object `o`;
  - have access to the data that is specific to `o`.
- constructors:
  - we create a new object of class `Foo` with the keyword `new`;
  - we initialise an object of type `Foo` by calling the constructor for that type;
  - the constructor is used to store data values in the object's instance variables.

Let's practice that!

What does it print?

```
1  class Number {
2      public int x;
3      public Number() { }
4  }
5
6  public class Main {
7      public static void main(String[] args) {
8          Number a = new Number();
9          System.out.println(a.x);
10         a.x=4;
11         System.out.println(a.x);
12         Number b = a;
13         b.x=5;
14         System.out.println(b.x);
15     }
16 }
```

What does it print?

```
1  class Number {
2      public int x;
3      public Number() { }
4  }
5
6  public class Main {
7      public static void main(String[] args) {
8          Number a = new Number();
9          System.out.println(a.x);
10         a.x=4;
11         System.out.println(a.x);
12         Number b = a;
13         b.x=5;
14         System.out.println(b.x);
15     }
16 }
```

Prints **0 4 5** because default initialisation of int and copying reference rather than object.

What does it print?

```
1  class Operation{
2   private int data;
3   public Operation(int d) {
4     data = d;
5   }
6   public void change(int d){
7     data = d + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation op = new Operation(50);
14    System.out.println("before change "+op.data);
15    op.change(500);
16    System.out.println("after change "+op.data);
17  }
18 }
```

What does it print?

```
1  class Operation{
2   private int data;
3   public Operation(int d) {
4     data = d;
5   }
6   public void change(int d){
7     data = d + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation op = new Operation(50);
14    System.out.println("before change "+op.data);
15    op.change(500);
16    System.out.println("after change "+op.data);
17  }
18 }
```

Prints **before change 50 - after change 600** because old data value is replaced.

What does it print?

```
1   public class Person {
2       public String name;
3       public Person() { }
4       public void assignName(String n) {
5           if (name.length() == 0) name = n;
6       }
7   }
8
9   public class Main {
10      public static void main(String[] args) {
11          Person p = new Person();
12          p.assignName("Lee");
13          System.out.println(p.name);
14      }
15  }
```

What does it print?

```
1  public class Person {
2      public String name;
3      public Person() { }
4      public void assignName(String n) {
5          if (name.length() == 0) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

Runtime error NullPointerException because default value of name is null.

What does it print?

```
1  public class Person {
2      public String name = "";
3      public Person() { }
4      public void assignName(String n) {
5          if (name.length() == 0) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

What does it print?

```
1  public class Person {
2      public String name = "";
3      public Person() { }
4      public void assignName(String n) {
5          if (name.length() == 0) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

Prints **Lee** because initialised to empty String with declaration and then set in method.

What does it print?

```
1   public class Person {
2       public String name;
3       public Person() { }
4       public void assignName(String n) {
5           if (name.equals(null)) name = n;
6       }
7   }
8
9   public class Main {
10      public static void main(String[] args) {
11          Person p = new Person();
12          p.assignName("Lee");
13          System.out.println(p.name);
14      }
15  }
```

What does it print?

```
1  public class Person {
2      public String name;
3      public Person() { }
4      public void assignName(String n) {
5          if (name.equals(null)) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

Runtime error NullPointerException. Not even **.equals** can be called on null.

What does it print?

```
1  public class Person {
2      public String name;
3      public Person() { }
4      public void assignName(String n) {
5          if (name == null) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

What does it print?

```
1  public class Person {
2      public String name;
3      public Person() { }
4      public void assignName(String n) {
5          if (name == null) name = n;
6      }
7  }
8
9  public class Main {
10     public static void main(String[] args) {
11         Person p = new Person();
12         p.assignName("Lee");
13         System.out.println(p.name);
14     }
15 }
```

Prints empty string and a new line because == comparison works.

What does it print?

```
1  public class Person {
2      public String name = "John Doe";
3
4      public Person(String n) {
5          System.out.println(name);
6          name = n;
7      }
8  }
9
10 public class Main {
11     public static void main(String[] args) {
12         Person p = new Person("Lee");
13         System.out.println(p.name);
14     }
15 }
```

What does it print?

```
1  public class Person {
2      public String name = "John Doe";
3
4      public Person(String n) {
5          System.out.println(name);
6          name = n;
7      }
8  }
9
10 public class Main {
11     public static void main(String[] args) {
12         Person p = new Person("Lee");
13         System.out.println(p.name);
14     }
15 }
```

Prints **John Doe - Lee**. Initialisation with declaration is executed before the constructor body.

Let's look at a longer example.

Hotel Reservation System

Goal: create a data type to manage hotel bookings

- Each hotel room has a number and a room rate.
- Each hotel room is associated with a representation of the days of a single month, indicating which days the room has already been booked for.

Hotel Reservation System: Client

```
public class HotelRoomReserver {

    public static void main(String[] args) {
        int startDate = Integer.parseInt(args[0]);
        int duration = Integer.parseInt(args[1]);

        HotelRoom rm1 = new HotelRoom(1, 65);
        HotelRoom rm2 = new HotelRoom(2, 65);
        HotelRoom rm3 = new HotelRoom(3, 75);
        HotelRoom[] rooms = { rm1, rm2, rm3 };

        for (int i = 0; i < rooms.length; i++) {
            HotelRoom r = rooms[i];
            if (r.isAvailable(startDate, duration)) {
                r.printBookings();
             }
        }
    }
}
```

Hotel Reservation System: Client

```
public class HotelRoomReserver {

    public static void main(String[] args) {
        int startDate = Integer.parseInt(args[0]);
        int duration = Integer.parseInt(args[1]);

        HotelRoom rm1 = new HotelRoom(1, 65);
        HotelRoom rm2 = new HotelRoom(2, 65);
        HotelRoom rm3 = new HotelRoom(3, 75);
        HotelRoom[] rooms = { rm1, rm2, rm3 };

        for (int i = 0; i < rooms.length; i++) {
            HotelRoom r = rooms[i];
            if (r.isAvailable(startDate, duration)) {
                r.printBookings();
             }
        }
    }
}
```

create and initialize objects

invoke constructor

object name

invoke method on r

Hotel Room Data Type

Goal: create a data type to manage hotel bookings

Set of values:

type	value	remarks
`int`	room number	
`int`	room rate	expressed in £
`boolean[]`	booked dates	`true` at index `i` iff room is booked for day `i`

Hotel Room Data Type

Goal: create a data type to manage hotel bookings
API:

`public class HotelRoom`

	`HotelRoom(int num, int rate)`	
`boolean`	`isAvailable(int sd, int d)`	*available from day `sd` until day `sd + d`?*
`void`	`printBookings()`	*show bookings for whole month*
`String`	`toString()`	*string representation*

Assumptions:

- Simplify by only considering a single month;
- skip index `0` in the `bookings` so that indexes and days of month line up;
- if someone is booked from day `i` to day `j`, they depart from hotel on the morning of `j`, so room only has to be free on days `i` — `(j-1)`.

Arrays of Objects

Array of `HotelRoom` objects

```
HotelRoom rm1 = new HotelRoom(1, 65);
HotelRoom rm2 = new HotelRoom(2, 65);
HotelRoom rm3 = new HotelRoom(3, 75);
HotelRoom[] rooms = { rm1, rm2, rm3 };
```

Array of `HotelRoom` objects: alternative

```
HotelRoom[] rooms = new HotelRoom[3];
rooms[0] = new HotelRoom(1, 65);
rooms[1] = new HotelRoom(2, 65);
rooms[2] = new HotelRoom(3, 75);
```

- Allocate memory for the array with `new`.
- Allocate memory for each object with `new`.

HotelRoom Class, version 1

```
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;

    public HotelRoom(int num, int rate){
        roomNumber = num;
        roomRate = rate;
    }

    public boolean isAvailable(int startDate, int duration){
        return true;
    }
}
```

HotelRoom Class, version 1

```
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;

    public HotelRoom(int num, int rate){
        roomNumber = num;
        roomRate = rate;
    }

    public boolean isAvailable(int startDate, int duration){
        return true;
    }
}
```

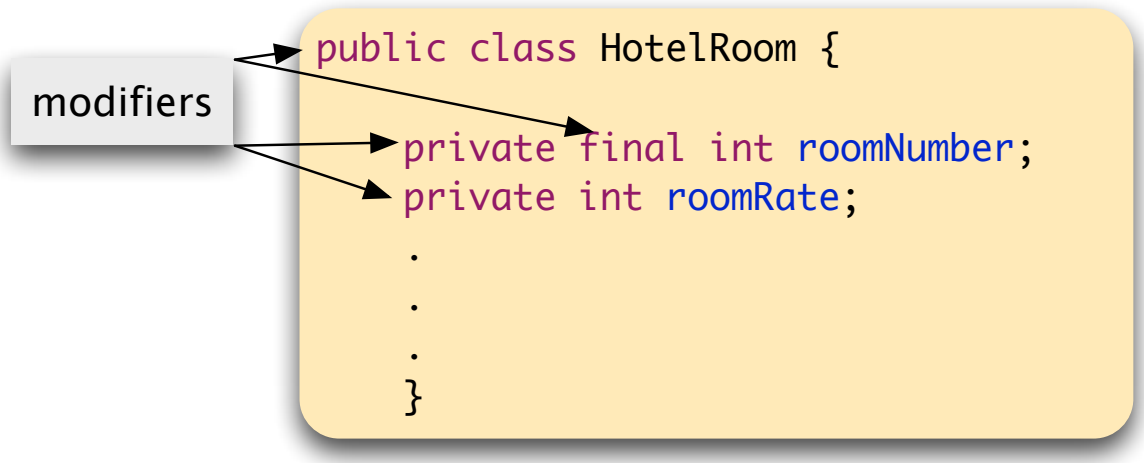
instance variables

constructor

instance method

More on Instance Variables

- Always use access modifier `private` (more on this later)
- Use modifier `final` for instance variables that never change after initial assignment.



Hotel Reservation System

Version 1

```
% java HotelReserver 12 3
Rooms available from 12 to 15
============================
HotelRoom@5f893efe
HotelRoom@2b86c6b2
HotelRoom@1d5ee671
```

Hotel Reservation System

Version 1

```
% java HotelReserver 12 3
Rooms available from 12 to 15
============================
HotelRoom@5f893efe
HotelRoom@2b86c6b2
HotelRoom@1d5ee671
```

How do we get a more informative output string when we call `System.out.println()` on a `HotelRoom` object?

HotelRoom Class, version 2

```
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;

    public HotelRoom(int num, int rate){
        roomNumber = num;
        roomRate = rate;
    }

    public boolean isAvailable(int startDate, int duration){
        return true;
    }

    public String toString(){
        return String.format("Room Number:\t%s\nRoom Rate:\t£%s.00\n",
                                        roomNumber, roomRate);
    }
}
```

Hotel Reservation System

Version 2

```
% java HotelReserver 12 3
Rooms available from 12 to 15
==============================

Room Number:    1
Room Rate:      65.00

Room Number:    2
Room Rate:      65.00

Room Number:    3
Room Rate:      75.00
```

HotelRoom Class, version 3

```
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;
    private boolean[] booked;

    public HotelRoom(int num, int rate){
        roomNumber = num;
        roomRate = rate;
        booked = HotelUtils.occupy();
    }

    public boolean isAvailable(int startDate, int duration){
        boolean available = true;
        for (int i = startDate; i < startDate + duration; i++) {
            available = available && !booked[i];
        }
        return available;
    }

    public String toString(){
        return String.format("\nRoom Number:\t%s\nRoom Rate:\t£%s.00",
                                        roomNumber, roomRate);
    }
}
```

call an external utility method which randomly flips false to true.

HotelRoom Class, version 4

```
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;
    private boolean[] booked;

    public HotelRoom(int num, int rate){
        roomNumber = num;
        roomRate = rate;
        booked = HotelUtils.occupy();
    }

    public boolean isAvailable(int startDate, int duration){
        boolean available = true;
        for (int i = startDate; i < startDate + duration; i++) {
            available = available && !booked[i];
        }
        return available;
    }

    public void printBookings(){
        HotelUtils.displayBookings(booked);
    }

    public String toString(){
        return String.format("\nRoom Number:\t%s\nRoom Rate:\t£%s.00",
                                        roomNumber, roomRate);
    }
}
```

another external utility method

Version 4

Version 4

```
% Rooms available from 12 to 15
================================

Room Number:     2
Room Rate:       65.00
 1: [ ][X][ ][X][X][X][ ]
 8: [ ][ ][X][ ][ ][ ][ ]
15: [X][ ][ ][X][ ][ ][ ]
22: [X][X][X][ ][ ][ ][X]
29: [X][X]
```

Recall that guests will leave on morning of 15^{th}, so room doesn't have to be free on day 15.

Interim Summary

Some new features:

- We implemented a `toString()` method for `HotelRoom`:
  - Java always implicitly calls this method whenever it executes commands like `System.out.println()`.
  - Every class gets a default version of `toString()`, but it's often useful to give our own classes a more specific implementation which gets used instead of the default.
- We created and used an array of type `HotelRoom[]`; i.e. `HotelRoom[] rooms = { rm1, rm2, rm3 };`

More on Constructors

Circle1: Omitting the constructor

```
public class Circle1 {
    private double radius;
    public double getArea(){
         return radius * radius * Math.PI;
     }
}
```

More on Constructors

Circle1: Omitting the constructor

```
public class Circle1 {
   private double radius;
   public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

- `Circle1 c = new Circle1(1.0)` — causes compile-time error.
- `Circle1 c = new Circle1()` — does work
  - though `c.getArea()` returns `0.00`!
- If you don't explicitly add a constructor, Java will automatically add a no-argument constructor for you.

More on Constructors

Circle again

```
public class Circle {
   private double radius;
    public Circle(double newRadius){
        radius = newRadius;
    }
   public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

- What happens if we call `Circle c = new Circle()`?
- This also causes a compile-time error — we only get the no-arg default constructor if there's no explicit constructor already defined.

More on Constructors

Generally considered good programming style to provide a no-arg constructor for your classes but not always practical.

No-arg Constructor: Version 1

```
public class Circle3 {
   private double radius;
    public Circle3(double newRadius){
        radius = newRadius;
    }
    public Circle3(){
        radius = 1.0;
    }
   public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

More on Constructors

No-arg Constructor: Version 2

```
public class Circle4 {
   private double radius;
    public Circle4(double newRadius){
        radius = newRadius;
    }
    public Circle4(){
        this(1.0);
    }
   public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

- `this(1.0);` — call another constructor of this class, and supply the value `1.0`.
- Must be the first line of the constructor.

Let's practice some more!

What does it print?

```
1  class Operation{
2   public int data;
3   public Operation(int d) {
4     data = d;
5   }
6   public void change(int data){
7     data = data + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation op = new Operation(50);
14    System.out.println("before change "+op.data);
15    op.change(500);
16    System.out.println("after change "+op.data);
17  }
18 }
```

What does it print?

```
1  class Operation{
2   public int data;
3   public Operation(int d) {
4     data = d;
5   }
6   public void change(int data){
7     data = data + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation op = new Operation(50);
14    System.out.println("before change "+op.data);
15    op.change(500);
16    System.out.println("after change "+op.data);
17  }
18 }
```

Prints **before change 50 - after change 50** because change method modifies local field.

What does it print?

```
1  class Operation{
2   public int data;
3   public Operation(int d) {
4     data = d;
5   }
6   void change(int data){
7     this.data = data + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation op = new Operation(50);
14    System.out.println("before change "+op.data);
15    op.change(500);
16    System.out.println("after change "+op.data);
17  }
18 }
```

Prints **before change 50 - after change 50** because change method modifies local field. Can be fixed with **this**.

What does it print?

```
1  class Operation2{
2   public int data;
3   public Operation2(int d) {
4     data = d;
5   }
6   public void change(Operation2 op){
7     op.data = op.data + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation2 op = new Operation2();
14    System.out.println("before change "+op.data);
15    op.change(op);
16    System.out.println("after change "+op.data);
17  }
18 }
```

What does it print?

```
1  class Operation2{
2   public int data;
3   public Operation2(int d) {
4     data = d;
5   }
6   public void change(Operation2 op){
7     op.data = op.data + 100;
8   }
9  }
10
11 class Main {
12  public static void main(String[] args){
13    Operation2 op = new Operation2();
14    System.out.println("before change "+op.data);
15    op.change(op);
16    System.out.println("after change "+op.data);
17  }
18 }
```

Prints **before change 50 - after change 150** operates on reference to itself.

Encapsulation

...or, why do instance variables have to be `private`?

Dalek Encapsulation: Unprotected Dalek

```
public class Dalek {
    public double batteryCharge = 5;
    public void batteryReCharge(double c) {...}
    public void move(int distance) {...}
}
```

Disabling the Dalek:

```
Dalek d = new Dalek(); // start off with a
                       // well-charged battery
d.batteryCharge = Double.NEGATIVE_INFINITY;
d.batteryReCharge(1000); // battery charge still -Infinity!
```

Dalek Encapsulation: Protected Dalek!

```
public class Dalek {
    private double batteryCharge = 5;
    public void batteryReCharge(double c) {...}
    public void move(int distance) {...}
}
```

Disabling the Dalek:

```
Dalek d = new Dalek(); // start off with a
                       // well-charged battery
d.batteryCharge = Double.NEGATIVE_INFINITY;
//Triggers compile-time Error
```

```
Exception ...: Unresolved compilation problem:
  The field Dalek.batteryCharge is not visible
```

Changing Internal Representation

Encapsulation:

- Keep data representation hidden with `private` access modifier.
- Expose API to clients using `public` access modifier.

Advantage: can switch internal representations without changing client.

Encapsulated data types:

- Don't touch data to do whatever you want.
- Instead, ask object to manipulate its data.

Access Modifiers Summary

Modifier	Class	Package	Global
Public	Yes	Yes	Yes
Default	Yes	Yes	No
Private	Yes	No	No

Access Modifiers Summary

Modifier	Class	Package	Global
Public	Yes	Yes	Yes
Default	Yes	Yes	No
Private	Yes	No	No

There is a fourth modifier which you will get to know later.

Immutability

Immutable data type: object's internal state cannot change once constructed.

mutable	*immutable*
`Picture`	
`Dalek`	`String`
Java arrays	primitive types

Immutability: Advantages and Disadvantages

Immutable data type: object's value cannot change once constructed.

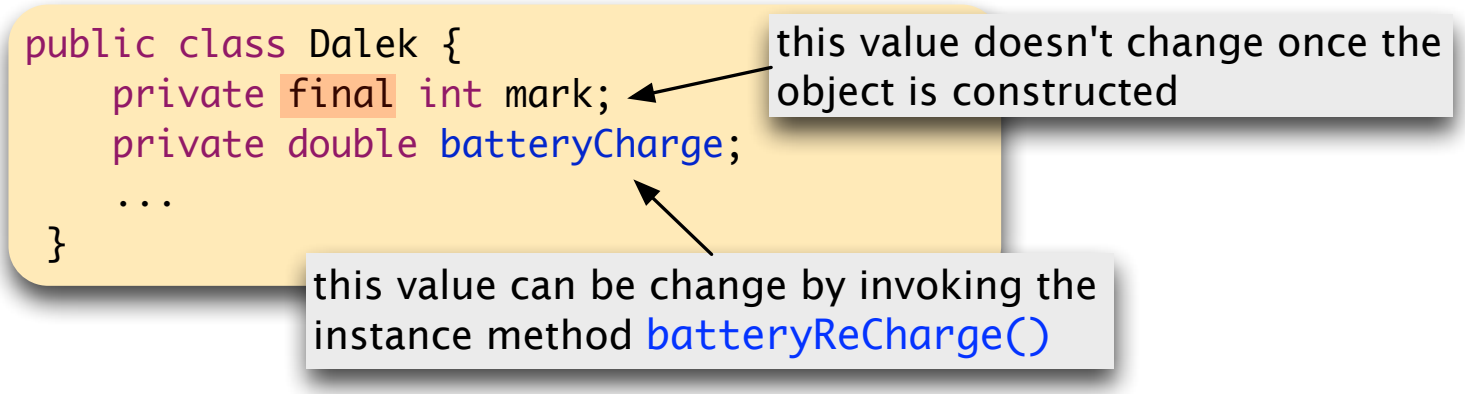
Advantages:

- Makes programs easier to debug (sometimes)
- Limits scope of code that can change values
- Pass objects around without worrying about modification
- Better for concurrent programming.

Disadvantages: New object must be created for every value.

The `final` Modifier

Final: declaring a variable to by `final` means that you can assign it a value only once, in initializer or constructor. E.g., Daleks come in three versions, Mark I, Mark II and Mark III.



Advantages:

- Helps enforce immutability.
- Prevents accidental changes.
- Makes program easier to debug.
- Documents the fact that value cannot change.

Getters and Setters

Encapsulation: instance variables should be private

```
public class Student {

    private String firstName;
    private String lastName;
    private String matric;

    public Student(String fn, String ln, String m) {
        firstName = fn;
        lastName = ln;
        matric = m;
    }
 }
```

Getters and Setters

Encapsulation: instance variables should be private

```
public class StudentTester {

   public static void main(String[] args) {
        Student student = new Student("Fiona", "McCleod", "s01234567");
        System.out.println(student.firstName);
        student.matric = "s13141516";
    }
}
```

we cannot assign to this variable!

we cannot read this variable!

Getters and Setters

Encapsulation: instance variables should be private

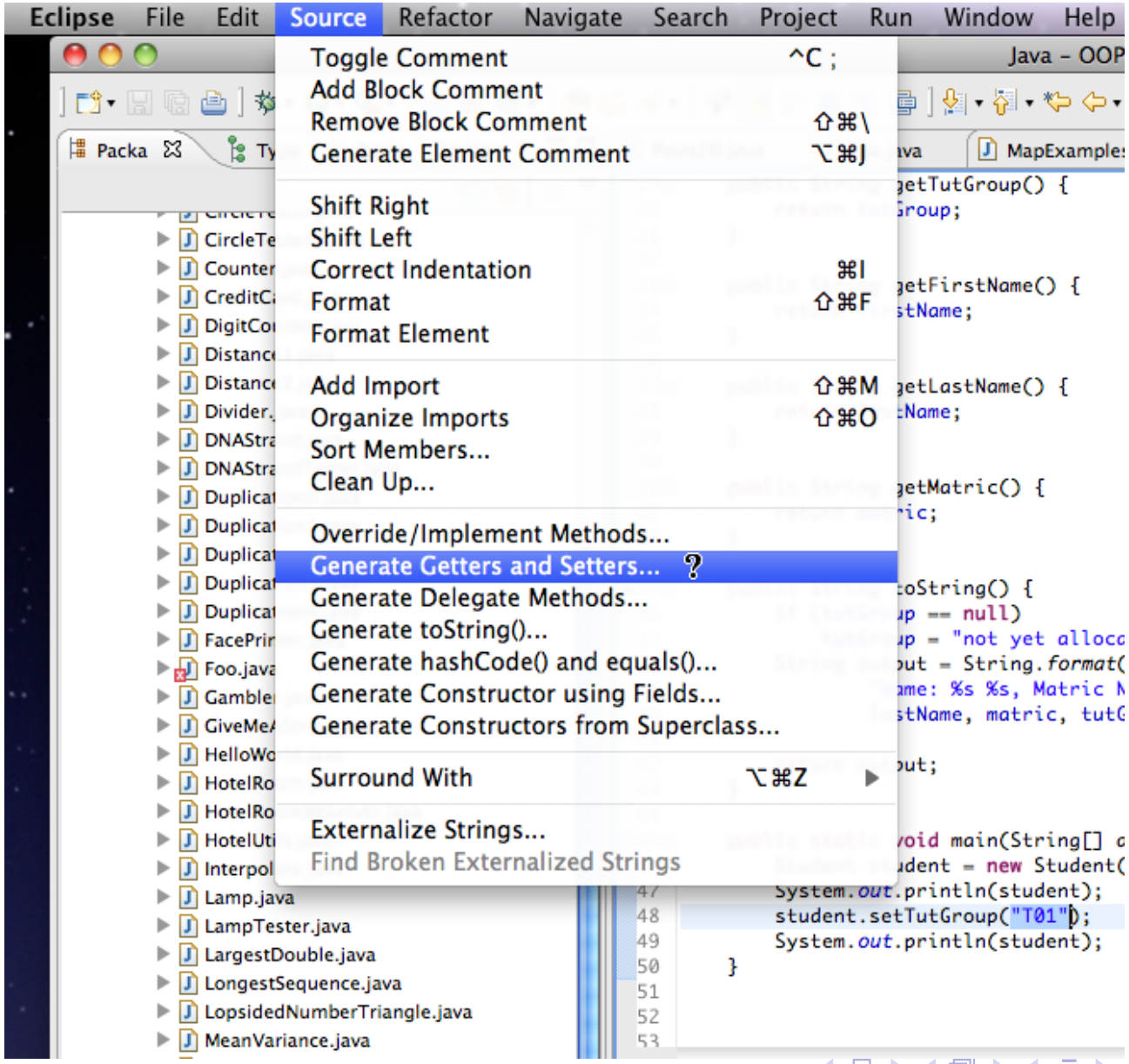
- We use instance methods to mediate access to the data in private instance variables, as needed.
- Accessor methods: just read the data
- Mutator methods: modify the data
- Java convention: given an instance variable `myData`, use
  - `getMyData()` method to read the data, and
  - `setMyData()` method to write to the data.
- Often called 'getters' and 'setters' respectively.

Getters and Setters

```
public class Student {

    private String firstName, lastName, matric, tutGroup;

    public Student(String fn, String ln, String m) {
        ...
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getMatric() {
        return matric;
    }
 }
```

Getters and Setters

Eclipse will generate setters and getters for you!



Summary: Object Orientation

Data type: set of values and collections of operations on those values.
In OOP: classes.
Simulating the physical world

- Java objects can be used to model real-world objects
- Not necessarily easy to choose good modelling primitives, or to get model that reflects relevant parts of reality.
- Examples: geometric figures, hotel rooms, ...

Extending the Java language

- Java doesn't have a data type for every possible application.
- User-defined classes enable us to add our own abstractions.

Summary: designing a Java class

- Use client code to motivate and test classes.
- instance variables:
  - represent data that is particular to an object (i.e., an instance!);
  - have scope over the whole class;
  - can hold mutable state;
  - can be manipulated by any instance method in the class.
- instance methods:
  - like static methods, but can only be called on some object `o`;
  - have access to the data that is specific to `o`.
- constructors:
  - we create a new object of class `Foo` with the keyword `new`;
  - we initialise an object of type `Foo` by calling the constructor for that type;
  - the constructor can be used to store data values in the object's instance variables.

Summary: Access Control

Encapsulation and visibility: All the instance variables and methods (i.e., members) of a class are visible within the body of the class.

Access modifiers: control the visibility of your code to other programs.

`public`:	member is accessible whenever the class is accessible.
`private`:	member is only accessible within the class.
default:	member is accessible by every class in the same package.

Benefits of encapsulation:

- Loose coupling
- Protected variation
- Exporting an API:
  - the classes, members etc, by which some program is accessed
  - any client program can use the API
  - the author is committed to supporting the API

Reading

pp99-121, i.e. continuing with Chapter 4 *Classes and Objects*, stopping at *Nested Classes*
We haven't talked about inheritance or interfaces (yet), but everything else should be looking familiar.