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

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`.
The Circle Class: Instance Methods

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

The Circle Class: Constructors

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

The Circle Class: Instance Variables

- `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 initialized; 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: Anatomy

```java
public class Circle {
    private double radius;

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

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

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 initialize 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.
Instance variable defaults: Person class

```java
public class Person {
    String name;

    public void assignName(String n) {
        ...
    }

    public static void main(String[] args) {
        Person p = new Person();
        p.assignName("Lee");
        System.out.println(p.name);
    }
}
```

Two versions of Person

**Version 1:**
```java
public class Person {
    String name;
    public void assignName(String n) {
        if (name.length() == 0)
            name = n;
    }
}
```

**Version 2:**
```java
public class Person {
    String name = "";
    public void assignName(String n) {
        if (name.length() == 0)
            name = n;
    }
}
```

Another two versions of Person

**Version 3:**
```java
public class Person {
    String name;
    public void assignName(String n) {
        if (name.equals(null))
            name = n;
    }
}
```

**Version 4:**
```java
public class Person {
    String name;
    public void assignName(String n) {
        if (name == null)
            name = n;
    }
}
```

Comparing versions of Person

Which of the versions will execute properly?
- ▶ Version 1
- ▶ Version 2
- ▶ Version 3
- ▶ Version 4
Brief interlude: Format Strings

How to gain more fine-grained control over print strings.

println can be Clunky

The student named 'Lee' is aged 18.
Using string concatenation

System.out.println("The student named "+ name
+ ", is aged "+ age
+ ");

String with Format Specifiers, 1
Target String
"The student named 'Lee' is aged 18."

String with Gaps
"The student named '_' is aged '_'."

String with Format Specifiers
"The student named '%s' is aged %s."

▶ %s is a placeholder for a string.
▶ Called a format specifier.
▶ Each format specifier in a string gets replaced by an actual value.

String with Format Specifiers, 2
String.format("The student named '%s' is aged %s.", name, age);

arg1
arg2
String with Format Specifiers, 3

Define a Format String

String str =
    String.format("The student named '%s' is aged %s.",
                name, age);
System.out.println(str);

Output
The student named 'Lee' is aged 18.

printf, 1

Shorter version

System.out.printf("The student named '%s' is aged %s.",
                name, age);

Output
The student named 'Lee' is aged 18.

printf, 2

Convert char to String

System.out.printf("'%s' is for Apple.", 'A');

Output
'A' is for Apple.

printf, 2

Round to 2 decimal places

System.out.printf("The value of pi is %f", Math.PI);
System.out.printf("The value of pi is %.2f", Math.PI);

Output
The value of pi is 3.141593
The value of pi is 3.14

include a newline

System.out.printf("The value of pi is %f
", Math.PI);
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 Room Data Type

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

<table>
<thead>
<tr>
<th>Set of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>boolean[]</td>
</tr>
</tbody>
</table>

### Hotel Room Data Type

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

**API:**

```java
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**

```java
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**

```java
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`.

### 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
```

How do we get a more informative output string when we call `System.out.println()` on a `HotelRoom` object?
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("Room Number: %s
Room Rate: £%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("Room Number: %s
Room Rate: £%s.00", roomNumber, roomRate);
    }
}

another external utility method
% Rooms available from 12 to 15
==============================
Room Number: 2
Room Rate: 65.00
1: [ ] [X] [ ] [X] [X] [X] [X]
8: [ ] [ ] [X] [X] [ ] [ ] [ ]
15: [X] [ ] [ ] [X] [ ] [ ] [ ]
22: [X] [X] [X] [ ] [ ] [ ] [X]
29: [X] [X]

Recall that guests will leave on morning of 15th, 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;
    }
}
```

▶ 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.

No-arg Constructor: Version 1

```java
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

```java
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.

Encapsulation

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

Dalek Encapsulation: Unprotected Dalek

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

Disabling the Dalek:

```java
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!

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

Disabling the Dalek:

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

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.

Immutability

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

<table>
<thead>
<tr>
<th>mutable</th>
<th>immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picture</td>
<td>Dalek</td>
</tr>
<tr>
<td></td>
<td>String</td>
</tr>
<tr>
<td>Java arrays</td>
<td>primitive types</td>
</tr>
</tbody>
</table>

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 be \texttt{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.

```java
public class Dalek {
    private final int mark;
    private double batteryCharge;
    ...
}
```

this value doesn't change once the object is constructed

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 \texttt{private}

```java
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;
    }
}
```

we cannot read this variable!

we cannot assign to 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 \texttt{myData}, use \texttt{getMyData()} method to read the data, and \texttt{setMyData()} method to write to the data.
- Often called ‘getters’ and ‘setters’ respectively.
Getters and Setters

```java
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;
    }
}
```

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 initialize 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.
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**: amounts to public for current purposes.

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

Java Tutorial
Reread anything up to pp121 that you’re not happy with yet. We haven’t talked about inheritance, interfaces or packages (yet), but everything else should be looking familiar.