Inf1B
Creating Classes

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics
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

Class Foo

- instance variables
- constructor
- instance methods
Classes and Clients

Class Foo
- instance variables
- constructor
- instance methods

Class FooTester

main {
    Foo f = new Foo(...);
    baz = f.doSomething();
}

client of Foo
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`.
Create a Circle object \texttt{c1}.

Call a method to get the area of that object: \texttt{c1.getArea()}

```java
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 %.2f\n", area1);

        Circle c2 = new Circle(5.0);
        double area2 = c2.getArea();
        System.out.printf("Area of circle c2 is %.2f\n", area2);
    }
}
```

Expected Output

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

```java
public class Circle {

  instance variables

  constructor

  instance methods
}
```
The Circle Class: Instance Methods

| public class Circle { |
| instance variables |
| constructor |

```java
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 Methods

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

- **radius** is an instance variable of the class **Circle**.
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.
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.
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 initialised; they get default values (e.g., 0 for `int`, `false` for `boolean`, `null` for all reference types).
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 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

```java
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

```java
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

```java
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

public class Circle {
    private double radius;

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

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

---

**Constructors**
- `public Circle(double newRadius)`
  - declares a private instance variable: `radius`.
  - sets the `radius` to the value passed to the constructor.

**Methods**
- `public double getArea()`
  - returns the area of the circle, calculated as `radius * radius * Math.PI`.
The Circle Class: Constructors

Alternative notation:

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

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

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

public class CircleTester {
    public static void main(String[] args) {
        Circle c1 = new Circle(1.0);
        double area1 = c1.getArea();
        System.out.printf("Area of circle c1 is %.2f\n", area1);

        Circle c2 = new Circle(5.0);
        double area2 = c2.getArea();
        System.out.printf("Area of circle c2 is %.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 \( \text{Foo} \) with the keyword \texttt{new};
  - we initialise an object of type \( \text{Foo} \) by calling the constructor for that type;
  - the constructor is used to store data values in the object’s instance variables.
Let’s practise that

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it
public class Number {
    public int x;
    public Number() {
    }
}

public class Main {
    public static void main(String[] args) {
        Number a = new Number();
        System.out.println(a.x);
        a.x = 4;
        System.out.println(a.x);
        Number b = a;
        b.x = 5;
        System.out.println(a.x);
    }
}

Prints 0 4 5 because default initialisation of int and copying reference rather than object.
What does it print?

```java
public class Number {
    public int x;
    public Number() { }
}

public class Main {
    public static void main(String[] args) {
        Number a = new Number();
        System.out.println(a.x);
        a.x = 4;
        System.out.println(a.x);
        Number b = a;
        b.x = 5;
        System.out.println(a.x);
    }
}
```

Prints 0 4 5 because default initialisation of int and copying reference rather than object.
public class Operation{
    public int data;
    public Operation(int d) {
        data = d;
    }
    public void change(int d){
        data = d + 100;
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change" + op.data);
        op.change(500);
        System.out.println("after change" + op.data);
    }
}
What does it print?

```java
public class Operation{
    public int data;
    public Operation(int d) {
        data = d;
    }
    public void change(int d){
        data = d + 100;
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change" + op.data);
        op.change(500);
        System.out.println("after change" + op.data);
    }
}
```

Prints **before change 50 - after change 600** because old data value is replaced.
What does it print?

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

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

Runtime error NullPointerException because default value of name is null.
public class Person {
    public String name;
    public Person() {}
    public void assignName(String n) {
        if (name.length() == 0) name = n;
    }
}

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

Runtime error NullPointerException because default value of name is null.
public class Person {
    public String name = "";
    public Person() { }
    public void assignName(String n) {
        if (name.length() == 0) name = n;
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.assignName("Lee");
        System.out.println(p.name);
    }
}
What does it print?

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

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

Prints **Lee** because initialised to empty String with declaration and then set in method.
public class Person {
    public String name;
    public Person() {}
    public void assignName(String n) {
        if (name.equals(null)) name = n;
    }
}

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

Runtime error NullPointerException. Not even .equals can be called on null.
public class Person {
    public String name;
    public Person() {
    }
    public void assignName(String n) {
        if (name.equals(null)) name = n;
    }
}

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

Runtime error NullPointerException. Not even .equals can be called on null.
public class Person {
    public String name;
    public Person() { }
    public void assignName(String n) {
        if (name == null) name = n;
    }
}

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

Prints Lee because == comparison works.
public class Person {
    public String name;
    public Person() {
    }
    public void assignName(String n) {
        if (name == null) name = n;
    }
}

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

Prints **Lee** because == comparison works.
public class Person {
    public String name = "John Doe";

    public Person(String n) {
        System.out.println(name);
        name = n;
    }
}

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

    public Person(String n) {
        System.out.println(name);
        name = n;
    }
}

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

Prints **John Doe - Lee**. Initialisation with declaration is executed before the constructor body.
Let’s look at a longer example.
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.
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();
            }
        }
    }
}
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();
            }
        }
    }
}
Goal: create a data type to manage hotel bookings
Set of values:

<table>
<thead>
<tr>
<th>type</th>
<th>value</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>room number</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>room rate</td>
<td>expressed in £</td>
</tr>
<tr>
<td>boolean[]</td>
<td>booked dates</td>
<td>true at index i iff room is booked for day i</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

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.
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 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.

```java
public class HotelRoom {
    private final int roomNumber;
    private int roomRate;
    .
    .
    .
}
```
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?
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: £%s.00\n", roomNumber, roomRate);
    }
}
% 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
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: %s
Room Rate: £%s.00", roomNumber, roomRate);
    }
}
call an external utility method which randomly flips false to true.
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
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\textsuperscript{th}, so room doesn’t have to be free on day 15.
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

```java
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

Circle1: Omitting the constructor

```java
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

```java
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

```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.
Let’s practise some more

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it
What does it print?

```java
public class Operation{
    private int data;
    public Operation(int d) {
        data = d;
    }
    public void change(int data){
        data = data + 100;
    }
    public String toString() {
        return "" + data;
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before\nchange\n"+op.toString());
        op.change(500);
        System.out.println("after\nchange\n"+op.toString());
    }
}
```

Prints

before change 50 - after change 50 because change method modifies local field.
What does it print?

```java
public class Operation{
    private int data;
    public Operation(int d) {
        data = d;
    }
    public void change(int data){
        data = data + 100;
    }
    public String toString() {
        return "" + data;
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change \n"+op.toString());
        op.change(500);
        System.out.println("after change \n"+op.toString());
    }
}
```

Prints **before change 50 - after change 50** because change method modifies local field.
What does it print?

```java
public class Operation{
    private int data;
    public Operation(int d) {
        data = d;
    }
    public void change(int data){
        this.data = data + 100;
    }
    public String toString() {
        return "" + data;
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change " + op.toString());
        op.change(500);
        System.out.println("after change " + op.toString());
    }
}
```

Prints **before change 50 - after change 600** because change method modifies local field. Can be fixed with **this**.
public class Operation{
    private int data;
    public Operation(int d) {
        data = d;
    }
    public void change(Operation op){
        op.data = op.data + 100;
    }
    public String toString() {
        return data + "";
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change"+op);
        op.change(op);
        System.out.println("after change"+op);
    }
}
public class Operation{
    private int data;
    public Operation(int d) {
        data = d;
    }
    public void change(Operation op){
        op.data = op.data + 100;
    }
    public String toString() {
        return data + "";
    }
}

public class Main {
    public static void main(String[] args){
        Operation op = new Operation(50);
        System.out.println("before change "+op);
        op.change(op);
        System.out.println("after change "+op);
    }
}

Prints **before change 50 - after change 150** operates on reference to itself and **toString** is called automatically by `System.out.println`. 
**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.
Objects First
Chapter 2 *Understanding Class Definitions*

Java Tutorial
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.