Inf1-OP
Classes and Objects

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

School of Informatics

February 26, 2018
Software engineering as managing change

The computer doesn’t care how code is organised – but humans care a lot. Changing code is hard and expensive, and because the world changes, essential. We want to make it easy and cheap, by:

- minimising the amount of code that must change
- making it easy to work out which code must change
- having the code that must change live together (changing 6 lines in one file is typically cheaper than changing 1 line of code in each of 6 files).
How can we make change easier and cheaper?

Key idea: hide certain information inside well-defined pieces of code, so that users of that piece of code don’t depend on it, and don’t need to change if it changes.
E.g. caller of a function doesn’t know how the function works: just what argument types it takes and what type it returns. This was an innovation in its time! (Look up “GOTO considered harmful”.) But what else should be hidden?
Data representation

Recall: a **data type** is a set of values and operations on those values. May be

- primitive, built into the language with operations defined in the compiler/runtime, e.g. `int`, `double`, `boolean`
- or user-defined, with operations defined in the programming language itself, e.g. `PrinterQueue`, `HotelRoom`, ...

In Java, `String` is an interesting intermediate case: you may have thought of it as primitive, but it’s defined in Java (google `java String.java` to see how). And then there are the standard libraries.
You shouldn’t need to know how a data type is implemented in order to use it. It should suffice to read the documentation: what operations are there, what do they do? Then you can write code that won’t need to change if the implementation changes. (Whether you may write code that does depend on the implementation is another matter. If not, the data type is encapsulated.) The general idea is not specific to OO, but Java does it differently from Haskell.
Towards object oriented programming...

So far in this course, we’ve been doing **Procedural programming** [verb oriented]
  ▶ tell the computer to do this, then
  ▶ tell the computer to do that.
You know:
  ▶ how to program with primitive data types e.g. `int`, `boolean`;
  ▶ how to control program flow to do things with them, e.g. using `if`, `for`;
  ▶ how to group similar data into arrays.
You’ve glimpsed OO: e.g., all our code is organised inside classes, and you have used **Strings**.
Philosophy of object orientation

The problem with structuring your software based on what it must do is that that changes a lot. The domain in which it works changes much less. Key insight: structuring your software around the things in the domain makes it easier to understand and maintain. (Many key names: look up Alan Kay, Rebecca Wirfs-Brock, Grady Booch, Barbara Liskov, Bertrand Meyer for writing on all this...) Object Oriented programming (OOP) [noun oriented]

- Things in the world know things: instance variables.
- Things in the world do things: methods.

In other words, objects have state and behaviour.
Intuition

Client

API
▶ adjust volume
▶ switch channel
▶ switch to standby

Implementation
▶ cathode ray tube
▶ 20” screen, 22 kg
▶ Sony Trinitron KV20M10

client needs to know how to use API

implementation needs to know what API to implement

Implementation and client need to agree on API ahead of time.
Intuition

Client

▶ adjust volume
▶ switch channel
▶ switch to standby

API

Implementation

▶ HD LED display
▶ 37” screen, 10 kg
▶ Samsung UE37C5800

client needs to know how to use API
implementation needs to know what API to implement

Can substitute better implementation without changing the client.
TV Example

Using the TV API

TV mytv = new TV();
mytv.setVolume(up);
mytv.setChannel(Film4);
mytv.standBy();

Note that we have two independent ideas here:

▶ Conceptual objects such as mytv are directly present in the program;
▶ They have static (compile-time) types that define their behaviour.
Classes: user-defined data types

- Java is a class-based object-oriented language.
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
- An object is an instance of a class.
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
- An object is an instance of a class.
- The code in a Java system is organised into classes.
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
- An object is an instance of a class.
- The code in a Java system is organised into classes.
- The system runs by objects sending messages to one another, and reacting to receiving messages, possibly by sending other messages...
  the sender trusts the receiver to do the right thing
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
- An object is an instance of a class.
- The code in a Java system is organised into classes.
- The system runs by objects sending messages to one another, and reacting to receiving messages, possibly by sending other messages...
  the sender trusts the receiver to do the right thing
- (Most) variables refer to objects.
Java is a class-based object-oriented language.
A class is a kind of data type that you can define yourself.
An object is an instance of a class.
The code in a Java system is organised into classes.
The system runs by objects sending messages to one another, and reacting to receiving messages, possibly by sending other messages... the sender trusts the receiver to do the right thing
(Most) variables refer to objects.
Classes: user-defined data types

- Java is a class-based object-oriented language.
- A class is a kind of data type that you can define yourself.
- An object is an instance of a class.
- The code in a Java system is organised into classes.
- The system runs by objects sending messages to one another, and reacting to receiving messages, possibly by sending other messages...

  the sender trusts the receiver to do the right thing

- (Most) variables refer to objects.

Java also comes with a lot of standardised classes, such as `String`. Let’s look at how `String` fits into this picture.
Constructors and Methods

String s;
s = new String("Hello, World!");
System.out.println(s.substring(0, 5));

- declare a variable (object name)
- call a constructor to create an object
- object name
- call an instance method that operates on the object's value
Static Methods vs. Instance Methods

**Static Methods:**
- Associated with a **class**.
- Identifying a method in a separate class requires name of the class:
  \[ \text{Math.abs()}, \text{PointDistance.distance()} \].

**Instance Methods:**
- Associated with an **object**.
- Identifying an instance method requires an object name:
  \[ \text{s.substring()} \]
## String: basis for text processing

Underlying **set of values**: sequences of Unicode characters.

In Java **Strings** are **immutable**: none of the operations change the value.

```java
public class String {
    String(String s)  // create a string with same value as s
    char charAt(int i)  // character at index i
    String concat(String t)  // this string with t appended
    int compareTo(String t)  // compare lexicographically with t
    boolean endsWith(String post)  // does string end with post?
    boolean equals(Object t)  // is t a String equal to this one?
    int indexOf(String p)  // index of first occurrence of p
    int indexOf(String p, int i)  // as indexOf, starting search at index i
    int length()  // return length of string
    String replaceAll(String a, String b)  // result of changing all as to bs
    String[] split(String delim)  // result of splitting string at delim
    boolean startsWith(String pre)  // does string start with pre?
    String substring(int i, int j)  // from index i to index j – 1 inclusive
}
```

[http://docs.oracle.com/javase/8/docs/api/java/lang/String.html](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html)
Typical String Processing Code

**is the string a palindrome?**

```java
public static boolean isPalindrome(String s) {
    int N = s.length();
    for (int i = 0; i < N / 2; i++) {
        if (s.charAt(i) != s.charAt(N - 1 - i))
            return false;
    }
    return true;
}
```

**extract filenames and extensions from a command-line argument**

```java
String s = args[0];
int dot = s.indexOf( ".");
String base = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

**print all lines from standard input containing the string "info"**

```java
while (!StdIn.isEmpty()) {
    String s = StdIn.readLine();
    if (s.contains("info"))
        System.out.println(s);
}
```

**print all ac.uk URLs in text file on standard input**

```java
while (!StdIn.isEmpty()) {
    String s = StdIn.readString();
    if (s.startsWith("http://") && s.endsWith("ac.uk"))
        System.out.println(s);
}
```
Gene Finding

Pregenomic era: Sequencing the human genome [Human Genome Project, 2003].

Postgenomic era: Analyse the data and understand structure.

Genome: Represent genome as a string over \{A, C, T, G\} alphabet.

Gene: A substring of genome that represents a functional unit.
  ▶ Preceded by ATG (start codon).
  ▶ Multiples of three nucleotides (codons other than start/stop).
  ▶ Succeeded by TAG, TAA or TGA (stop codons).

Goal: find all genes
Gene Finding: Algorithm

**Algorithm**: Scan left-to-right through genome.

- If start codon, then set \( \text{beg} \) to index \( i \).
- If stop codon and substring is multiple of 3:
  - output gene
  - reset \( \text{beg} \) to \(-1 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>start</th>
<th>stop</th>
<th>( \text{beg} )</th>
<th>gene</th>
<th>remaining portion of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>-1</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGATGTC</td>
</tr>
<tr>
<td>1</td>
<td>TAG</td>
<td>-1</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGATGTC</td>
</tr>
<tr>
<td>4</td>
<td>ATG</td>
<td>4</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
<tr>
<td>9</td>
<td>TAG</td>
<td>4</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
<tr>
<td>16</td>
<td>TAG</td>
<td>4</td>
<td>CATAGCGCA</td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
<tr>
<td>20</td>
<td>TAG</td>
<td>-1</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
<tr>
<td>23</td>
<td>ATG</td>
<td>23</td>
<td></td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
<tr>
<td>29</td>
<td>TAG</td>
<td>23</td>
<td>TGC</td>
<td>ATAGATGCATAGCGC</td>
<td>ATAGATGCATAGCTAGCTAGATGTC</td>
</tr>
</tbody>
</table>
Strings and Equality

**equals()**: are the characters inside a String object the same?

**==**: do the two object references refer to the same instance?

In all the coursework for this course, you should be using **equals()** for checking whether two strings are ‘the same’.
Possible memory representation of a string

- genome = "AACAAGTTTACAAGC"

\[
\begin{array}{cccccccccccc}
D0 & D1 & D2 & D3 & D4 & D5 & D6 & D7 & D8 & D9 & DA & DB & DC & DD & DE \\
\end{array}
\]

\[
\begin{array}{cc}
A0 & A1 \\
\end{array}
\]

```
import java.lang.String;

String genome = "AACAAGTTTACAAGC";

String s = genome.substring(1, 5);
String t = genome.substring(9, 13);

(s == t) // false
(s.equals(t)) // true
```

s and t are different strings that share the same value "ACAA"
Possible memory representation of a string

- `genome = "AACAAGTTTACAAAGC"

- `s = genome.substring(1, 5);`
- `t = genome.substring(9, 13);`

- `(s == t)` is false
- `(s.equals(t))` is true

`s` and `t` are different strings that share the same value "ACAA"
Next time, we’ll see how to define a \texttt{Circle} class (in several variants). Let’s start by seeing how we might use one. Suppose its API is:

\begin{verbatim}
public class Circle
  Circle(double radius) \textit{constructor}
  double getArea()
  void enlarge(int scaleFactor)
  boolean equals(Object o) \textit{true iff o is a Circle of same size}
\end{verbatim}

Unlike \texttt{String}, \texttt{Circle} is mutable: its state can be changed by sending it message \texttt{enlarge}.
Using Circle

Circle c1 = new Circle(1);
double a1 = c1.getArea(); // pi

Circle c2 = new Circle(2);
double a2 = c2.getArea(); // 4 pi

Circle c3 = c1; // two references to same object
double a3 = c3.getArea(); // pi

System.out.println (c1 == c2); // false
System.out.println (c1.equals(c2)); // also false

System.out.println (c1 == c3); // true
System.out.println (c1.equals(c3)); // also true
Using Circle, continued

c1.enlarge(2);

double a1new = c1.getArea(); // now 4 pi
double a2new = c2.getArea(); // still 4 pi
double a3new = c3.getArea(); // now 4 pi

System.out.println (c1 == c2); // still false
System.out.println (c1.equals(c2)); // now true

System.out.println (c1 == c3); // still true
System.out.println (c1.equals(c3)); // also still true
Local* variables of primitive type use stack space and are created like this:

```java
public void foo() {
    int i = 5;
    //...
}
```

Things of reference type use heap space† and are created with `new`, which allocates the right amount of space and, for objects, calls a constructor. E.g.

```java
public void bar() {
    int[] a = new int[5]; // allocate space for 5 ints
    Circle c = new Circle(2); // allocate space for a Circle
    //...
}
```
Possible sources of confusion

1. * Primitively-typed attributes of objects (e.g. the radius of the circle), or primitively-typed elements of arrays (e.g. the contents of the int array), are on the heap with their owning object. (Otherwise putting the owning object on the heap would be pointless: it would collapse when its innards went out of scope!)

2. † a local reference to a thing of reference type lives on the stack, even though the thing itself is in the heap: once method bar the stack memory containing references a and c is reclaimed.
   
   If there are other references to the same heap objects elsewhere in the program – e.g., a or c have been passed to a method still in progress, or stored as attributes of an object – those references can still be used.

   If there are no other references, the heap objects will now be available for garbage collection.
Space for an array is different from space for its contents!

Note: allocating space for the array (of references) vs allocating space for an object. If the element type of the array is itself a reference type, we have to do both:

```java
// Allocate space for 5 refs to Circles:
Circle[] someCircles = new Circle[5];

// Put a ref to a pre-existing Circle in the array:
someCircles[0] = c;

// Allocate, create, put in array:
someCircles[3] = new Circle(7);
```
What about freeing that heap space?

Bit of history (look up e.g. *C dynamic memory management* if you want to know more).

- In C, programmers use(d) `malloc` and `free` to request heap memory and say when they’ve finished with it. Highly efficient when done well, but very error prone.

- Meanwhile, functional languages like Lisp and ML had *garbage collection*, freeing programmers from that duty: major performance problem, especially for predictability.

- C++ improved C by introducing `new` and `delete` which at least did more of the work of working out how much memory was needed and exactly what to free. `new` also invoked a constructor to build an object.

- Work on incremental garbage collection essentially solved the performance problems (memory leaks still possible)...

- Java (and Haskell, and every modern language) has garbage collection. `new` is retained essentially as in C++.
Summary: Object

An object has state, behaviour and identity.
Summary: Object

An object has state, behaviour and identity.

- State: the data that the object encapsulates

  - Behaviour: the messages the object understands, and what it does in response

    - exactly what an object does in response to a message may depend on its state

- Identity: two objects may currently have identical data, but be different objects

  - e.g. o1 and o2 could be in the same state now, but not after you sent a message to o1, which affects o1's state but not o2's.
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object *encapsulates*

- **Behaviour**: the messages the object understands, and what it does in response

- **Identity**: two objects may currently have identical data, but be different objects e.g. \(o_1\) and \(o_2\) could be in the same state now, but not after you sent a message to \(o_1\), which affects \(o_1\)’s state but not \(o_2\)’s.
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object *encapsulates*
- **Behaviour**: the messages the object understands, and what it does in response
- **Identity**: two objects may currently have identical data, but be different objects
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object *encapsulates* the object may decide to modify its state in response to a message

- **Behaviour**: the messages the object understands, and what it does in response

- **Identity**: two objects may currently have identical data, but be different objects
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object encapsulates. The object may decide to modify its state in response to a message.

- **Behaviour**: the messages the object understands, and what it does in response. Exactly what an object does in response to a message may depend on its state.

- **Identity**: two objects may currently have identical data, but be different objects.
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object *encapsulates* the object may decide to modify its state in response to a message

- **Behaviour**: the messages the object understands, and what it does in response exactly what an object does in response to a message may depend on its state

- **Identity**: two objects may currently have identical data, but be different objects e.g. o1 and o2 could be in the same state now, but not after you sent a message to o1, which affects o1’s state but not o2’s.
Summary: Object

An object has state, behaviour and identity.

- **State**: the data that the object *encapsulates*; the object may decide to modify its state in response to a message; but immutable objects never do this.

- **Behaviour**: the messages the object understands, and what it does in response. Exactly what an object does in response to a message may depend on its state.

- **Identity**: two objects may currently have identical data, but be different objects. E.g. o1 and o2 could be in the same state now, but not after you sent a message to o1, which affects o1’s state but not o2’s.
Summary: Why use object orientation?

OO has taken the world by storm. Why?
Summary: Why use object orientation?

OO has taken the world by storm. Why?
For *programming*, OOP is nice, but not clearly better than FP, say.

*for most applications, E&OE, don't tell PW I said so :-)*
Summary: Why use object orientation?

OO has taken the world by storm. Why?
For *programming*, OOP is nice, but not clearly better than FP, say.
For *software engineering*, OO is definitely better*

- use objects to model real-world things
- use classes to model *domain concepts*.
- These change more slowly than specific functional requirements,
- so what OO does is to put things together that change together as requirements evolve.

Change is the thing that makes software engineering hard and interesting; OO helps manage it.
* for most applications, E&OE, don’t tell PW I said so :-)*
Summary: in Java

A variable can have

- a primitive type e.g., boolean, int, double; or
- a reference type: any class, e.g. String, Picture, Color, any array type.

Things of reference type are created using `new` and destroyed by being garbage collected automatically.

Variables of reference type contain references to those things, not the things themselves. So:

- Two references can refer to the same thing.
- Copying the reference does not copy the thing, so...
- ...when you pass the reference into a method, and the method uses the reference to change the state of the thing, the change is visible outside.
- Need to distinguish `==` (these references refer to the same thing) from `.equals` (these references refer to things that are currently equal).
Learning Outcomes for this week

▶ What are the State, Behaviour and Identity of an object in OO?
▶ What is immutuability?
▶ Difference between instances and classes.
▶ Difference between references and primitive types.
▶ Difference between instance methods and static methods. How do I use them?
▶ Difference between heap and stack memory and when it is used.
▶ Comparing references with ‘==’ and contents with ‘.equals’.
Reading

Java Tutorial
pp99-121, i.e. continuing with Chapter 4 *Classes and Objects*,
stopping at *Nested Classes*
pp287-308, i.e. Chapter 9 *Numbers and Strings* from *Characters* to
the end.