Objects First With Java
A Practical Introduction Using BlueJ

More sophisticated behaviour

Using library classes to implement some more advanced functionality

Main concepts to be covered

- Using library classes
  - More than just `ArrayList` ...
- Reading code documentation
- Writing code documentation

The Java class library

- Thousands of classes
- Tens of thousands of methods
  - Many useful classes that make life much easier
  - Many you will probably never use …
- A qualified Java programmer must be able to work with the library.
  - It’s just easier, more reliable, more economic, …

Working with the library

You should:

- know some important classes by name (such as `ArrayList`);
- know how to find out about other classes (methods, parameters);
- read the Java library's documentation (available in html).

Remember:

- We only need to know the interface, not the implementation (hiding the details …).

The big objective …

- Prepare your own classes of “library quality”.
- Others can use them – just as they use library classes.
- That’s typical for real-life software development (long-term, large-scale, big teams, …).

Project: A Technical Support System

- A textual dialog system
  - Provide technical support for customers.
  - Online communication mimics real support.
  - That is: let's cheat again!
- Idea based on the AI project *'Eliza'* by Joseph Weizenbaum (MIT, 1960s)
- classes: `SupportSystem` (main class), `InputReader`, `Responder`.
- In the following: `SupportSystem`.

- *(Explore in BlueJ ...)*

Main loop structure
(method `start` in `SupportSystem`)

```
boolean finished = false;

while(!finished) {

    do something                    read next input, e.g.

    if(exit condition) {            everything processed, e.g.
        finished = true;
    }
    else {
        do something more           generate response, e.g.
    }
}
```

Main loop body

from class `InputReader`

```
String input = reader.getInput();
...
String response = responder.generateResponse();
System.out.println(response);
```

from class `Responder`

- read some input
- ask responder to generate a response
- print that response

The exit condition

```
String input = reader.getInput();

if(input.startsWith("bye")) {
    finished = true;
}
```

- Where does ‘startsWith’ come from?
- What is it? What does it do?
- How can we find out?
- What happens with “Bye” or “ bye”?

Reading class documentation

- Documentation of the Java libraries in HTML format
- Readable in a web browser
- Class API: *Application Programmers' Interface*
- Interface description for all library classes

Interface vs. implementation

The documentation includes

- the name of the class;
- a general description of the class's purpose;
- a list of the class's constructors and methods
- return values (types, classes) and parameters for each constructor and method
- a description of the purpose of each constructor and method

the *interface* of the class
(this is "abstraction in action!")

Interface vs. implementation

The documentation ***does not*** *include*

- private fields (most fields are private)
- private methods
- the bodies (source code) for each method

the *implementation* of the class

Side note: String equality

```
if(input == "bye") {
    ...
}
```

*tests **identity***

effect: do left- and right-hand side refer to the same object ?
***NOT*: do they have the same value?**

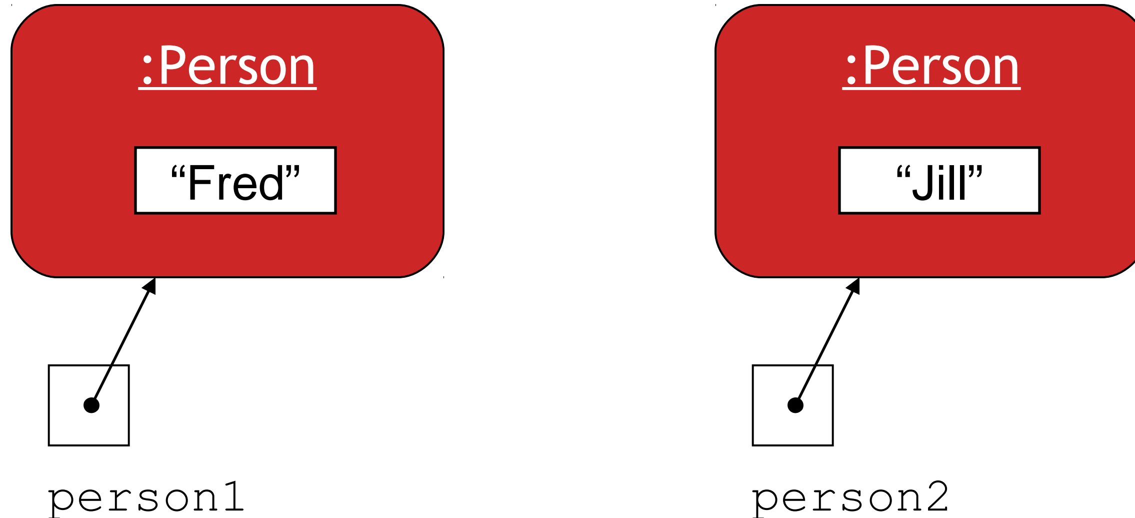
```
if(input.equals("bye")) {
    ...
}
```

*tests **equality***

- Strings should (almost) always be compared with **`.equals`**

Identity vs. equality 1

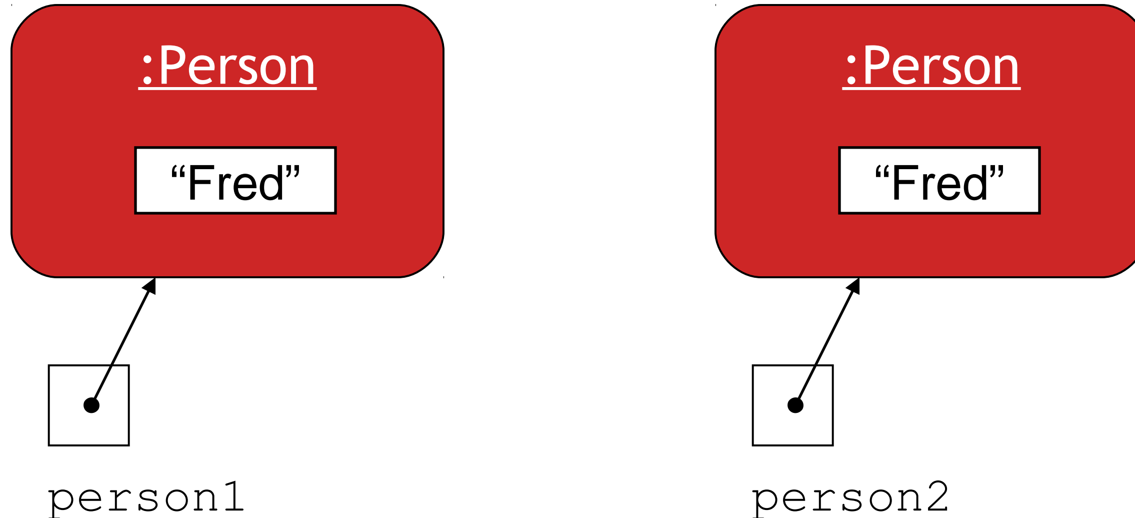
Other (non-String) objects:



`person1 == person2 ?`

Identity vs. equality 2

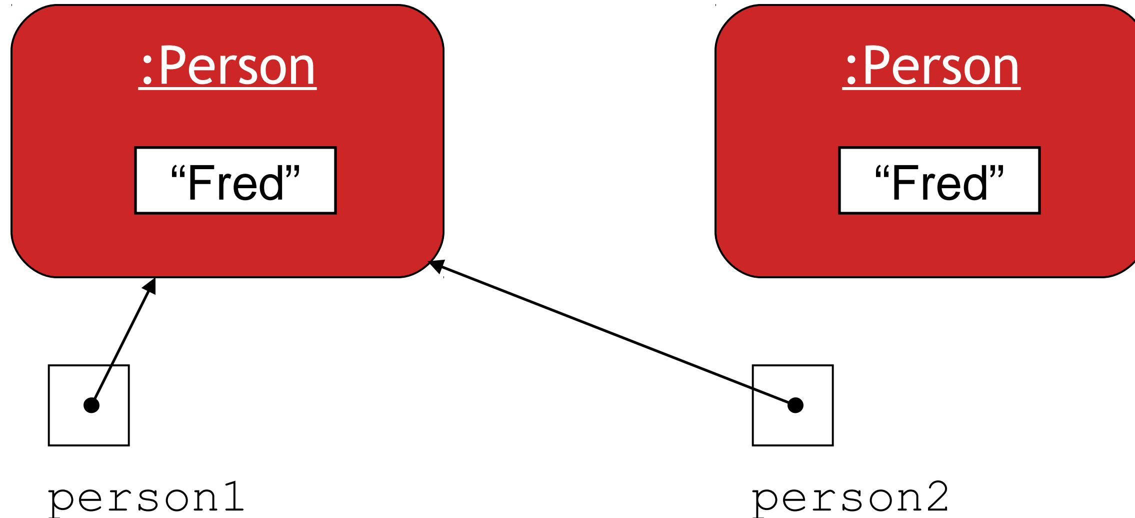
Other (non-String) objects:



`person1 == person2 ?`

Identity vs. equality 3

Other (non-String) objects:

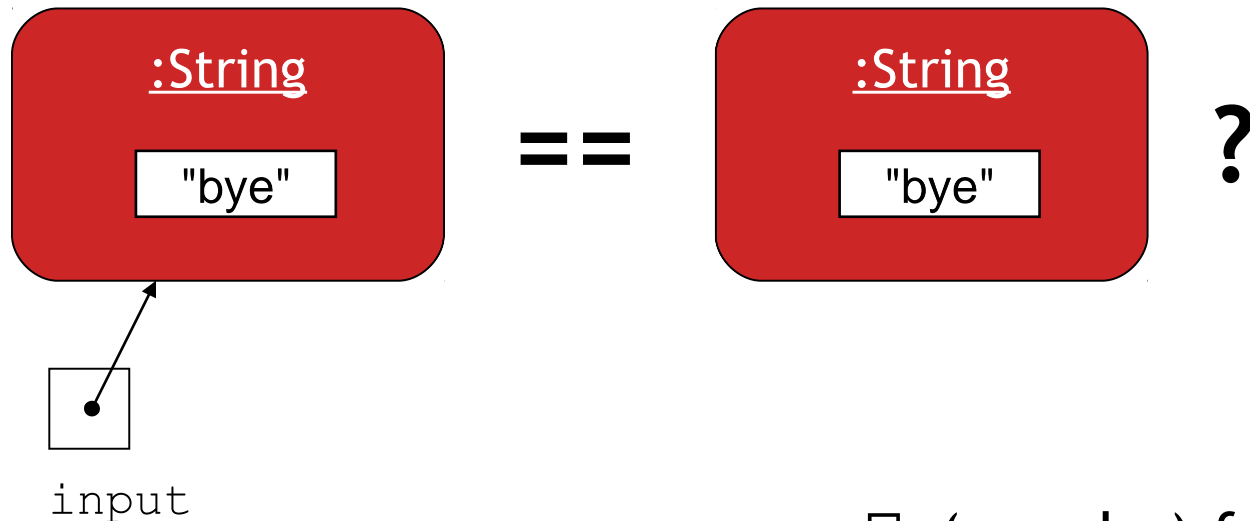


`person1 == person2 ?`

Identity vs. equality (Strings)

```
String input = reader.getInput();
if(input == "bye") {
    ...
}
```

== tests identity

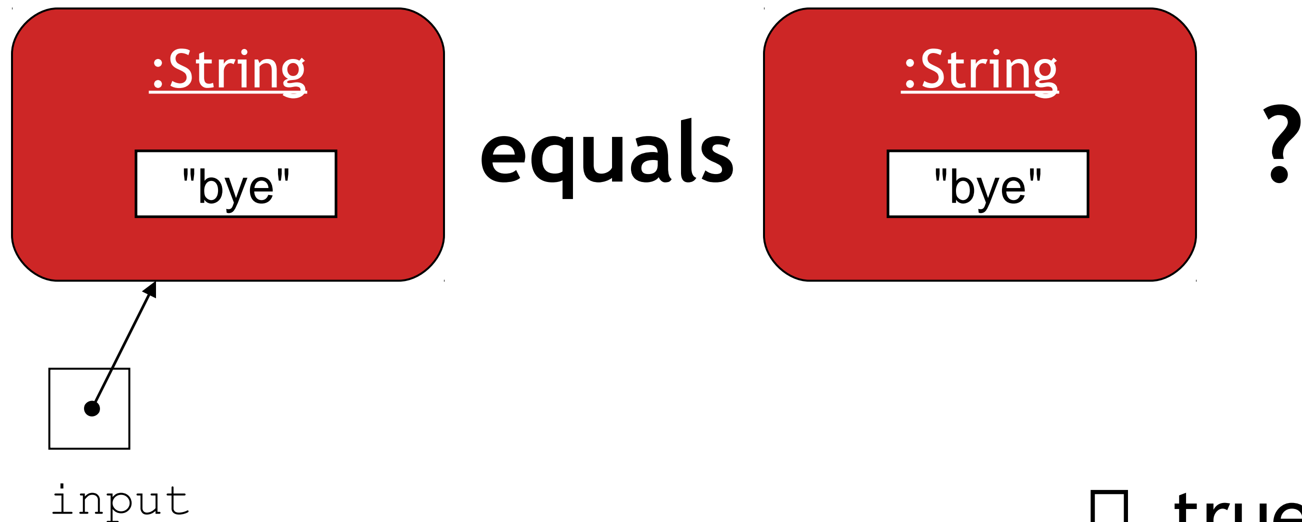


 (may be) false!

Identity vs. equality (Strings)

```
String input = reader.getInput();
if(input.equals("bye")) {
    ...
}
```

equals tests equality



Using Random

- The library class `Random` can be used to generate (pseudo) random numbers

```
import java.util.Random;
...
Random randomGenerator = new Random();
...
int index1 = randomGenerator.nextInt();
int index2 = randomGenerator.nextInt(100);
```

... over the whole range of integers

... over a limited range of integers (here: 0..99)

No need to create a new `Random` object any time you need a number - just call `nextInt`!

Generating random responses

```
public Responder()
{
    randomGenerator = new Random();
    responses = new ArrayList();
    fillResponses();
}

public String generateResponse()
{
    int index = randomGenerator.nextInt(responses.size());
    return (String) responses.get(index);
}

public void fillResponses()
...
```

… for the random numbers

… for storing the possible responses

… for creating some possible responses

Maps

- Maps are collections that contain a flexible number of pairs of values.
- Pairs consist of a key and a value.
- Lookup works by supplying a key (instead of an index) and retrieving a value.
- Efficient implementation of put, get.
- An example: a telephone book.

Using maps

- A map with Strings as keys and values

:HashMap

"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

- Particular implementation: `HashMap`

Using maps

```
HashMap phoneBook = new HashMap();

phoneBook.put("Charles Nguyen", "(531) 9392 4587");
phoneBook.put("Lisa Jones", "(402) 4536 4674");
phoneBook.put("William H. Smith", "(998) 5488 0123");
```

casting: any type is possible in a `HashMap`

```
String number = (String)phoneBook.get("Lisa Jones");
System.out.println(number);
```

most important methods of `HashMap`: `put` , `get`

BlueJ -> example “tech-support-complete”

Sets

- Sets are collections that contain a flexible number of objects.
- Each individual element is stored at most once.
- Important methods: set, contains
- A set does not maintain a specific order.
- Lookup is done via an Iterator (no order, no index, no key).

Using sets

```
import java.util.HashSet;
import java.util.Iterator;
...
HashSet mySet = new HashSet();

mySet.add("one");
mySet.add("two");
mySet.add("three");
mySet.add("one");

Iterator it = mySet.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

one variation of a set: `HashSet`

effect?

Compare this to ArrayList code!

Collections so far

- Lists
- Maps
- Sets
- What is different, what is equivalent?

Writing class documentation

- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- Reading the documentation must take less time than reading the complete code does!
- Crucial: how detailed??
- Make your class a 'library class'!

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number (with date)
- the authors' names
- documentation for each constructor and each method
- Rationale: why doing sth this way, what other ways have been tested/dropped (why?)

Elements of documentation

Documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned
- Rationale: why doing sth this way, what other ways have been tested/dropped (why?)

Javadoc – the Java documentation generator

Class comment:

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author      Michael Kölling and David J. Barnes
 * @version     1.0  (1.Feb.2002)
 */
```

javadoc

Method comment:

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return  A set of Strings, where each String is
 *          one of the words typed by the user
 */
public HashSet getInput(String prompt)
{
   ...
}
```

Creating Java documentation

- Command Line
  - command `javadoc`

- In BlueJ
  - Use Tools -> Project Documentation

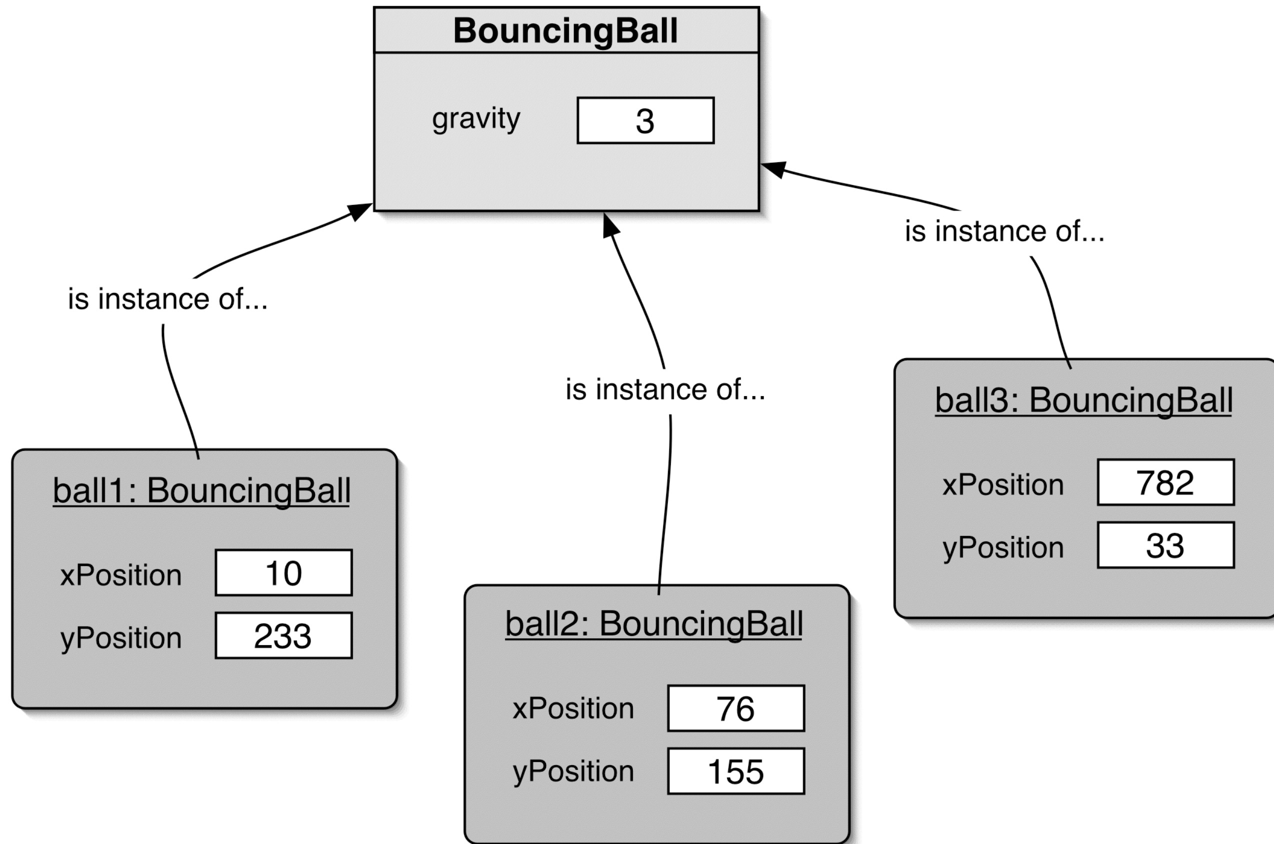
Public vs. private

- Remember access modifiers; they define the visibility of fields, methods, etc.
- Public attributes (fields, constructors, methods) are accessible to other classes.
- Fields should not be public.
- Private attributes are accessible only within the same class.
- Only methods that are intended for other classes should be public.

Information hiding

- Data belonging to one object is hidden from other objects.
- Know what an object can do, not how it does it.
- Information hiding increases the level of *independence*.
- Independence of modules is important for large systems and maintenance.
- Things one doesn’t need to know vs. things one even should not know (side effects!).

Class variables



Constants

```
private static final int gravity = 3;
```

- `private`: access modifier, as usual
- `static`: class variable
  - one (shared) variable (copy) for all instances
- `final`:
  - initialised only **once**:
    - when declared (as above) or
    - in the constructor
  - value won't change throughout the execution of an application – similar to a **constant**

Review

- Java has an extensive class library.
- A good programmer must be familiar with the library.
- The documentation tells us what we need to know to use a class (interface).
- The implementation is hidden (information hiding).
- We document our classes so that the interface can be read on its own (class comment, method comments).