G51OOP

7. Inheritance

Object Oriented Programming

Colin Higgins
(based on notes by Steve Bagley)

1. Background

- Look more closely at how objects relate to each other
- Objects composed of other objects
- Delve into inheritance
- But first a history lesson…

Review - Procedural Code

- Code executed statement-by-statement
- Conditionals and loops can change the order
- Statements modify data structures in memory

Procedural Dangers

- Data structures are open to modification
  - By any piece of code
- Hard to track down errors
- Or update the code to handle changes to the data structures
- Result: Program crashes…

Object-oriented Code

- Object-Oriented Programs tend to work in a different manner
- Programs tend to be structured in terms of objects interacting with each other
- But what is an object?

Objects

- Basic building block of program
- Collection of data and code that model something
- Used by other objects in the program to get the job done
- Once written, we can use it several times

Example

- Think about storing the details of a Person
  - Surname
  - Forename
  - Address
- Could store these as separate strings
- Messy to manage for lots of people

```
public class People {
    public static void main(String args[]) {
        String surname[256];
        String forename[256];
        String postcode[256];
        int houseNumber[256];

        surname[0] = "Higgins";
        forename[0] = "Colin";
        postcode[0] = "NG8 1BB";
        houseNumber[0] = 42;
    }

}
```

Problems

- Only thing that links the details together is the array index
- Have to pass everything separately to other methods to process
- Lots of typing!
- Reliant on the Programmer to get it right!
- Solution, create a Person class

```
public class Person {
    private String forename;
    private String surname;
    private String postcode;
    private int houseNumber;

    public Person(String forename, String surname,
                  String postcode, int houseNumber) {
        this.forename = forename;
        this.surname = surname;
        this.postcode = postcode;
        this.houseNumber = houseNumber;
    }
…
}
```

```
public class Person {
    …
    public String getForename() {
        return forename;
    }

    public String getSurname() {
        return surname;
    }

    public String getPostcode() {
        return postcode;
    }
}
```

Person Object

- Gathered all the data for one person in one place (a `Person` object)
- Treat it as an atomic unit when necessary
- But also (by using the methods) access the data within it when we need it
- Now reliant on the compiler to link the data together

Person Object

- External Code no longer has to worry about how the data is stored
- Just what it wants to do with it
- The internal representation of an object can change without effecting the external users

```
public class People  {
    public static void main(String args[]) {
        Person persons[256];

        person[0] = new Person(“Higgins”, “Colin”,
           “NG8 1BB”, 42);
        System.out.println(
                person[0].getSurname());
    }

}
```

2. Object Relationships

- Three main object relationships
  - **Has-a** One object includes an instance of another object
  - **Is-a** One object is a specialization of another type of object
  - **Uses** One object uses another object to complete its task

Has-a relationship

- Suppose we were to implement the software for a Bank
- Model details of each Bank Account
  - Details of the Customer
  - Account balance
- Represent each account as an object

```
public class Account  {
    private String customerSurname;
    private String customerForename;
    private String customerPostCode;
    private int customerHousenumber;

    private int balance;
    …
}
```

Bank Accounts

- Class interface is messy
- Lots of methods that have nothing to do with bank accounts
- `Account` and `Person` have fields in common
- Why not define `Account` as containing a `Person` object?

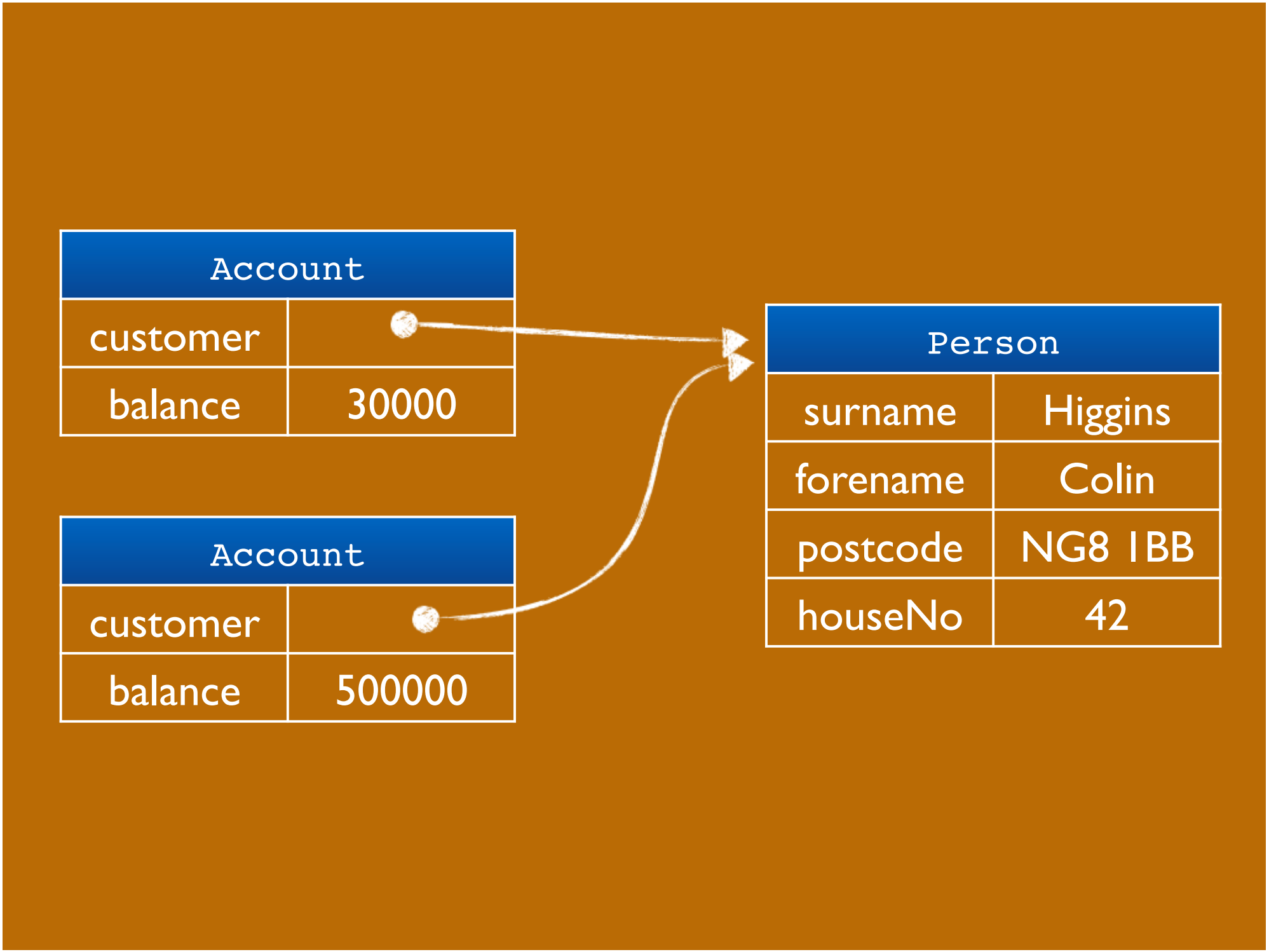
```
public class Account {
    private Person customer;
    private int balance;

    …
}
```

Object composition

- Advantages of object composition:
  - Cleaner object implementation
  - No repeated code
  - Shared objects

Shared Objects

- Suppose a customer has two bank accounts
- How do we know that it is the same customer?
- Both Account objects could reference the *same* Person object


Account
customer
balance
30000
Account
customer
balance
500000
Person
surname
Higgins
forename
Colin
postcode
NG8 1BB
houseNo
42

Object Sharing

- Advantages
  - Less memory usage
  - Easy to tell if it is the same customer
  - Only one place to update
- Disadvantages
  - Changes to `Person` effect both `Accounts`

Bank Account

- `Account` object just handles account info
  – Customer + balance
- `Person` object handles person data
  – Name + Address
- Work together to provide the full system
- Note: simple test code to test objects independently (see JUnit for testing)

3. Extending the system

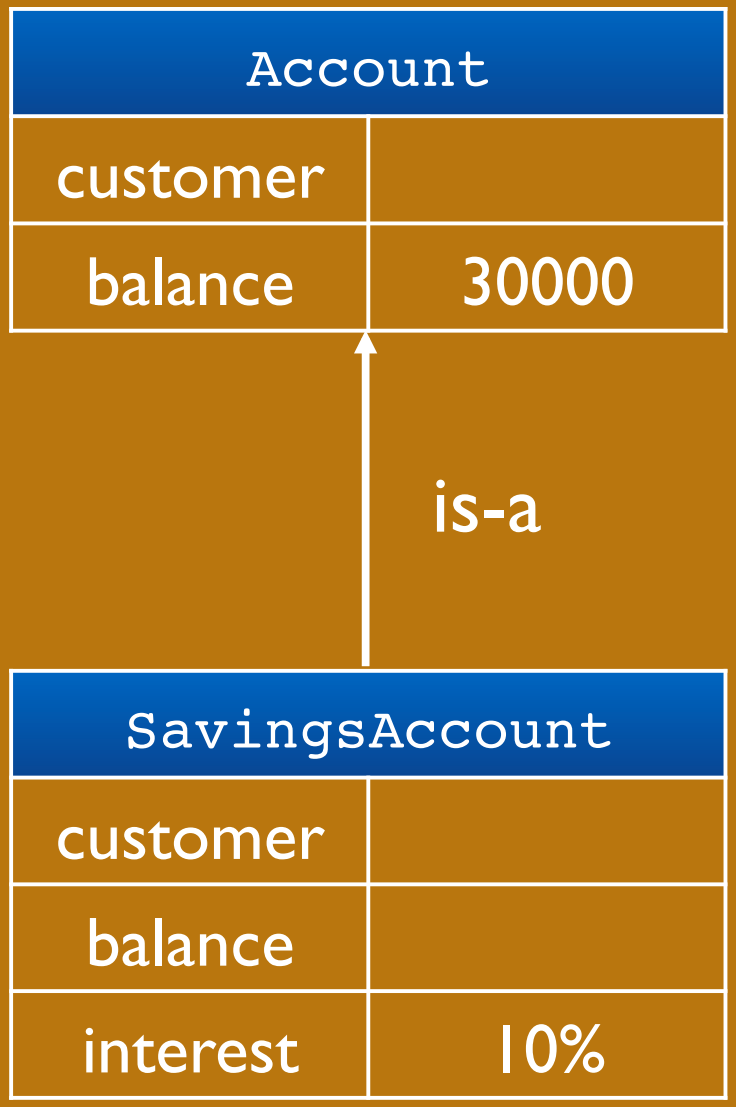
- Suppose the Bank wants to start providing savings account which give interest
- How would we add this to our system?
- Obviously, a new class of objects, `SavingsAccount`

Savings Account

- Has all the same state and operations as bank account
- Plus new operations:
  `calculateInterest()`
  `setInterestRate()`
  `getInterestRate()`
- New state – the interest rate
- Do we have to re-implement all of `Account`?

Inheritance

- OO programming allows us to extend classes to add new functionality
- This is called *Inheritance*: the *is-a* relationship `SavingsAccount` is an `Account`
- The new class, called a sub-class, *inherits* all the functionality of the original *super-class*


Account
customer
balance
30000
is-a
SavingsAccount
customer
balance
interest
10%

Inheritance

- New functionality (methods or data) can be added
- Existing functionality can be overridden

Inheritance in Java

- Class defined as extends the super class
  `public class SavingsAccount extends Account`
- New features defined as you would expect for any class
- A Class has one super class
- A class may have many sub-classes

Access Protection

- Objects are encapsulated
- Data only modified by methods
- Can sub-classes modify the super classes data? Should they be able to?
- Answer, if the super class lets them

Access restriction

- Methods/data can be defined as:
  - `public` — anything can access
  - `private` — only this class can access
  - `protected` — this class and the *immediate* subclasses

Overriding Methods

- Just provide a new definition
- If you need to access the super class version user the `super` keyword
- `super.doSomething()` — call the super classes implementation of doSomething()
- `doSomething()` — calls the object's classes implementation of doSomething

4. Interfaces – brief review

- Sometime we need to "inherit" from multiple classes - Java provides a partial solution
- As well as inheritance, classes can also *implement* an *Interface*
- Interfaces are a collection of abstract methods
- A class can implement multiple interfaces

Sortable Interface

```
interface Sortable {
    public int compareTo(Sortable other);
}
```

All methods are abstract so don't need to declare it explicitly

Interfaces

- Very important programming technique
- Defines a contract or protocol that an object supports
- Describes how one object interacts with other objects
- When those objects are not known

Interfaces

- Use the `implements` keyword to show that a class supports an interface
- Remember to make the methods in the class public or you can't call them!
- Class can support multiple interfaces
- Interfaces can be extended using inheritance

Interfaces

- Java uses interfaces a lot
- Threads — `Runnable` interface
- Sorting — `Comparable` interface
- Cloning — `Cloneable` interface (weird!)
- Event Handling — several e.g. `ActionListener`