Lecture 19

- Covers
  - Methods that return a value
  - void methods
  - Parameter passing (call-by-value)
- Reading: Savitch 4.1

Example - bank accounts

- Create and test a class representing a bank account that
  - Has an id, name, and balance
  - Allows a user to deposit, withdraw, and get the balance
- Rules
  - Id's and names must not be missing
  - Balance must not be negative (no overdraft)

Defining class BankAccount

- To define the BankAccount class and test it, we go through the following typical steps
  1. Sketch a model of the class
  2. Define the class header
  3. Define the attributes
  4. Define the constructors
  5. Define the methods

Sketching a model of the class

BankAccount	class name
String accountNumber String customerName double balance	attributes
BankAccount(String accNo, String custName) void deposit(double amount) void withdraw(double amount) double getBalance( ) String toString( )	methods

Specifying the class header

```
public class BankAccount
{

}
```

Declaring the attributes

```
public class BankAccount
{
    private String accountNumber;
    private String customerName;
    private double balance;
}
```

Defining the constructors

```
public BankAccount(String accNo, String custName)
{
    accountNumber = accNo;
    customerName = custName;
    balance = 0;
}
```

Defining the methods

- To make a deposit
- To withdraw money
- To get the balance
- To display a BankAccount object

Method to get the balance

```
public double getBalance( )
{
    return balance;
}
```

Methods that return a value

- Sometimes a method's invocation should result in a value being sent back to the sender of the invoking message
- When a method is expected to send a result back to the sender, it is said to return a value
- Each method must define a return type, i.e. the type of the value it sends back

Methods that return a value

- A method may have many variables and attributes with which it works, and many calculations it performs
- The method has to specify which value or calculation it sends back
- This is called the return value

Methods that return a value

- A return value is specified by a return statement
- In the previous example, balance was the return value, so therefore, the value stored in the balance attribute of the object is sent back as the result of the method's invocation

Methods that return a value

- Processing of a method terminates at a return statement
- The value specified by the return statement is returned to the call site (the place in the program that invoked the method)
- When a method terminates, processing in the program then continues at the call site

Method to make deposits

```
public void deposit(double amount)
{
    if (amount <= 0)
    {
        System.out.println(amount + " is not a valid deposit!");
        return;
    }
    else
    {
        balance = balance + amount;
    }
}
```

void methods

- Sometimes a method is not expected to return a value, but rather updates some attributes or performs some I/O actions
- Methods that do not return a value are called void methods (or sometimes "procedures")
- We specify a method does not return a value by giving it the return type void

return in void methods

- There is no need to place a return statement inside a void method as it does not need to specify a return value
- However, the return statement can be used in void methods to terminate the execution of the method at that point
- The return statement in this case does not specify a return value

Parameter passing

- When we expect a message invoking a method to send it some data, the message is said to expect arguments
- When a method that expects arguments is invoked, the actual values with which it should deal are specified
- In the header of a method definition, what the method expects as arguments must be specified inside brackets

Parameter passing

- The type of each argument and its identifier (the name by which it will be referred in the method) are needed to specify the expected arguments
- The order of the expected arguments and the actual values used in a method invocation must match

```
public void deposit(double amount)
{
    …
}
```

Parameter passing

- The expected arguments in the method definition are called the formal parameters of the method
- When we invoke this method on an object and specify values to “plug-in” to the formal parameters, the values are called the actual parameters or arguments

Parameter passing

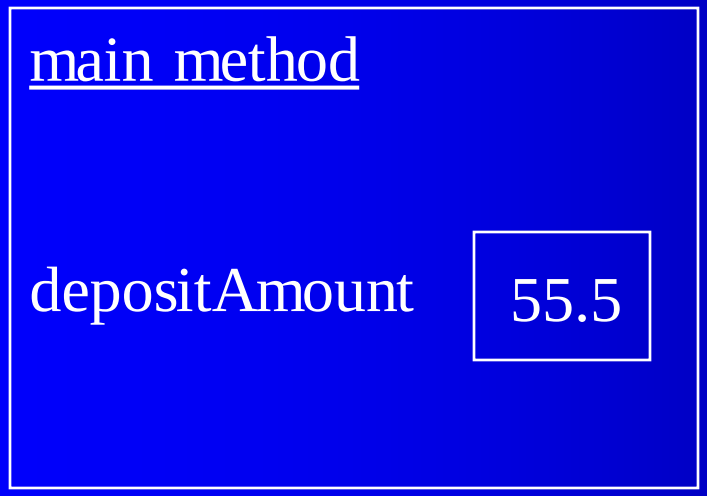
```
public void deposit(double amount)
{
   …
}

BankAccount b = new BankAccount("3210 4359", "B. Gates");
b.deposit(150.55);
```

formal parameters

actual parameters or arguments

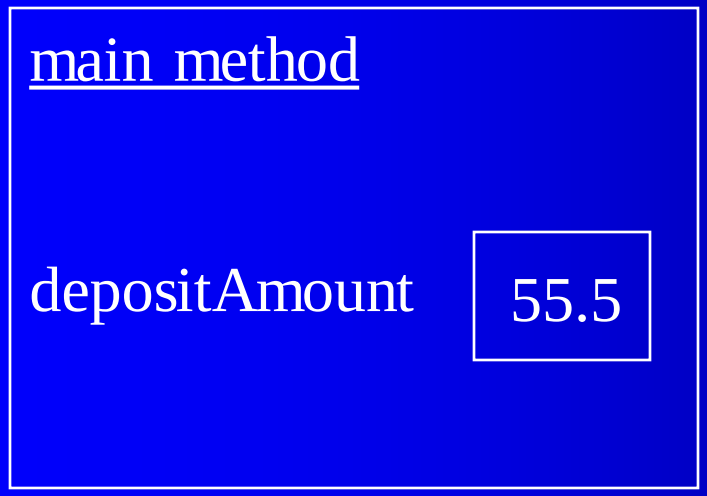
Parameter passing

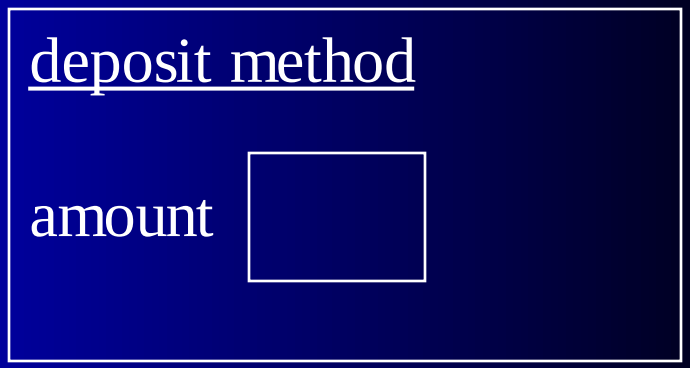
```
double depositAmount = 55.55;
b.deposit(depositAmount);
```



Parameter passing

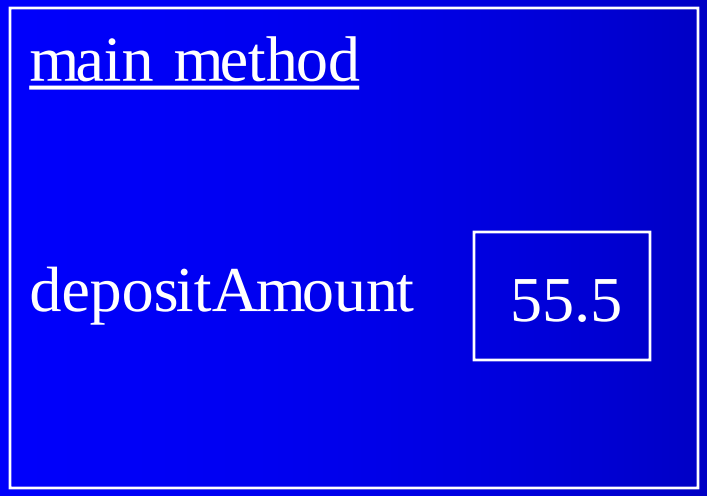
```
double depositAmount = 55.55;
b.deposit(depositAmount);
```

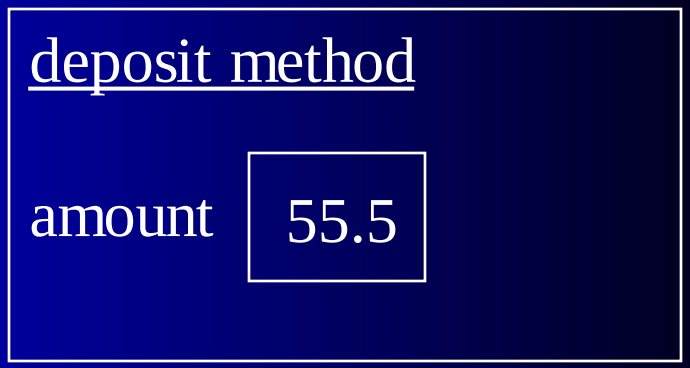




Parameter passing

```
double depositAmount = 55.55;
b.deposit(depositAmount);
```





Method to withdraw money

```
public void withdraw(double amount)
{
    if (amount <= 0) { return; }
    if (amount > balance) { return; }
    balance = balance - amount;
}
```

Overuse of return statements

- As with break statements in loops, using return statements many times within a method leads to unstructured programming
- A good practice is to use a local variable to store and calculate the return value in methods that return a value
- Try to place a single return only in a method and make it the last statement

Method to withdraw money

- Alternate implementation

```
public void withdraw(double amount)
{
    if (amount <= 0)
    {
        System.out.println("Invalid amount for withdrawal specified");
    }
    else if ( amount > balance )
    {
        System.out.println("Amount would cause account to be overdrawn");
    }
    else
    {
        balance = balance - amount;
    }
}
```

Method to display a BankAccount object

```
public void displayAccountDetails( )
{
    System.out.println("BankAccount details: "
                + "\n\taccountNr: " + accountNumber
                + "\n\tcustomername: " + customerName
                + "\n\tbalance: " + balance);
}
```

toString method

- Often we wish to display all the details of an object to the screen
- One approach is a display method as on the previous slide
- Sometimes we want to send the details to the screen (or somewhere else such as a file or device)
- The toString method is a special method that returns the content of the object as a String variable

toString method

```
public String toString( )
{
    String description = "BankAccount["
                + " accountNr: " + accountNumber
                + " customername: " + customerName
                + " balance: " + balance
                + " ]";
    return description;
}
```

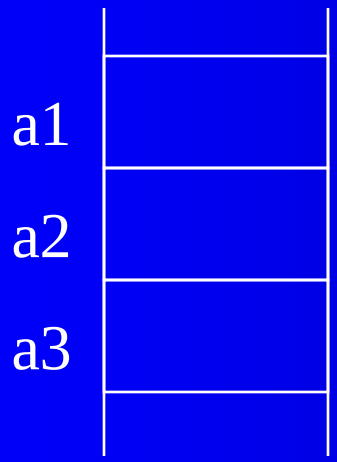
- *This method is very handy for testing*

Object instantiation

- As with Strings, objects are created by instantiating a class
- A variable of class type holds only the memory location of an object
- Creating a new object to work with involves three parts
  - Creating a variable to refer to the object
  - Creating an object
  - Assigning the memory location of the object to the variable

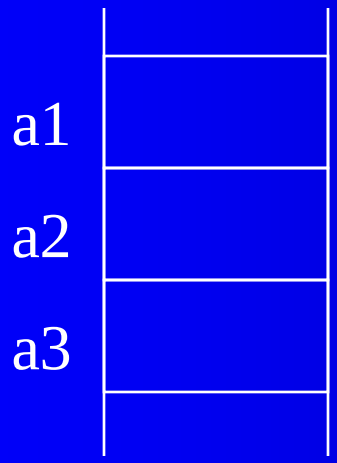
Object instantiation

```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```



Object instantiation

```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```

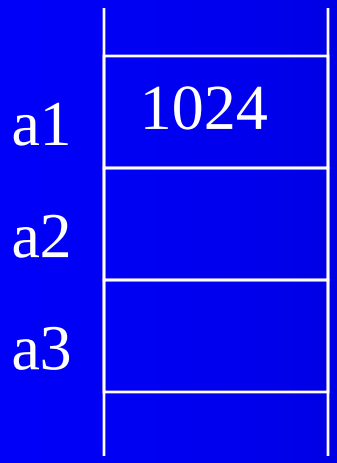


1024

accountNumber:	“3210”
customerName:	“marvin”
balance:	0

Object instantiation

```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```

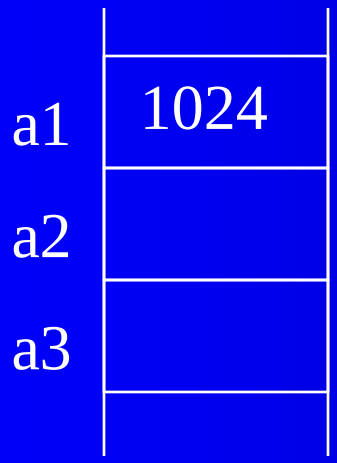


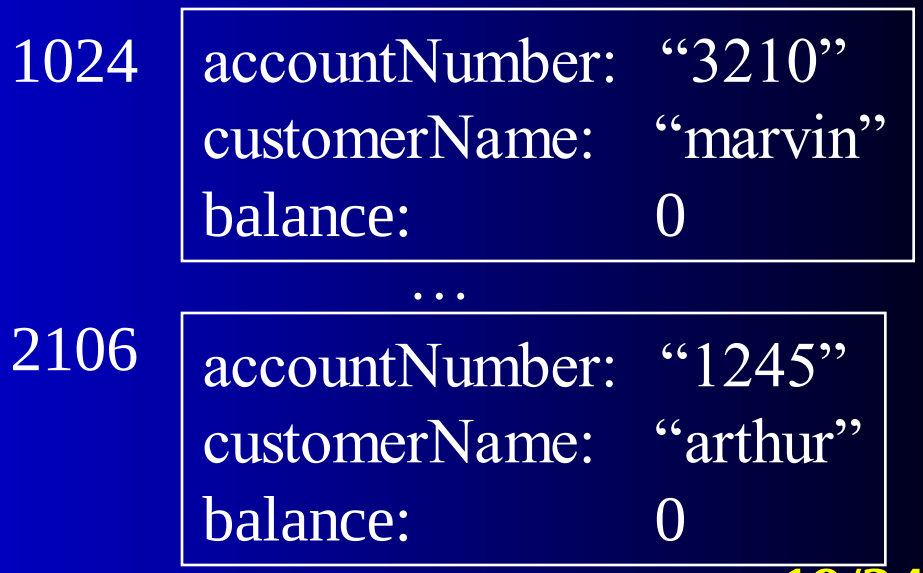
1024

accountNumber:	“3210”
customerName:	“marvin”
balance:	0

Object instantiation

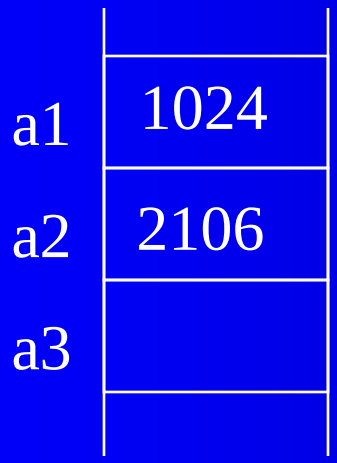
```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```

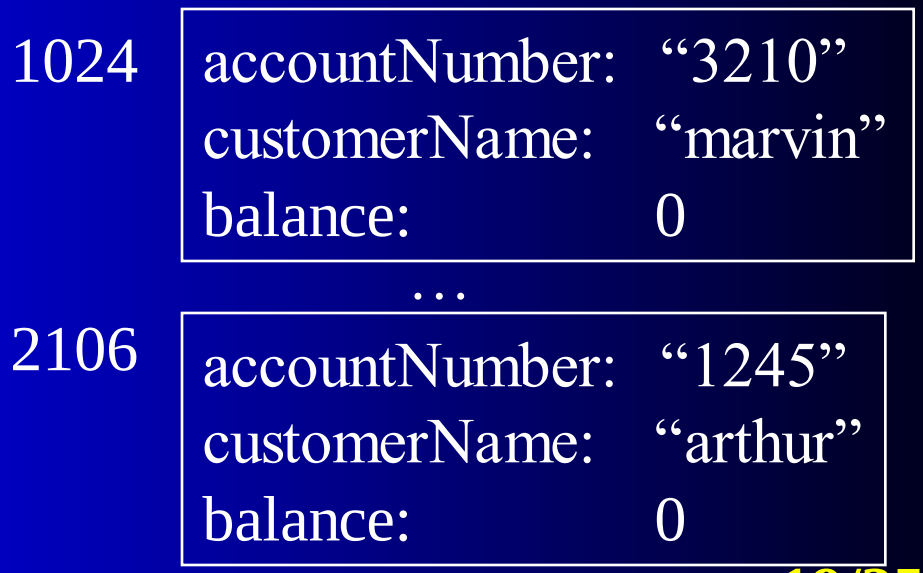




Object instantiation

```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```

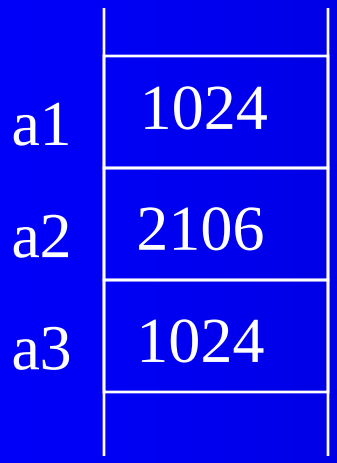


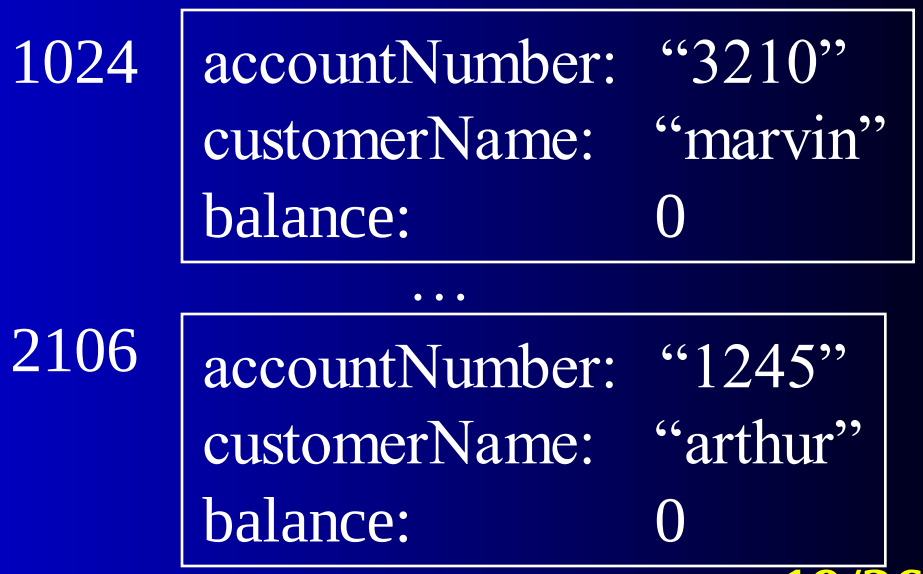


Object instantiation

```
BankAccount a1, a2, a3;
a1 = new BankAccount("3210", "marvin");
a2 = new BankAccount("1245", "arthur");
a3 = a1;
```

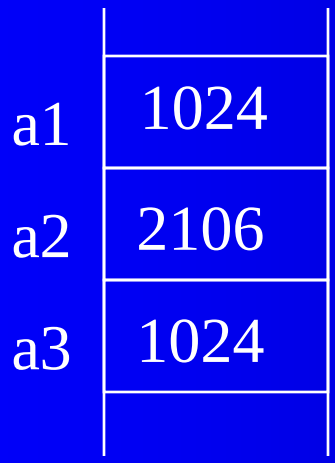
* *the variables a1 and a3 now refer to the same object*





Object instantiation

```
a1.deposit(100);
a2.deposit(300);
a3.withdraw(20);
```



1024

accountNumber:	“3210”
customerName:	“marvin”
balance:	100

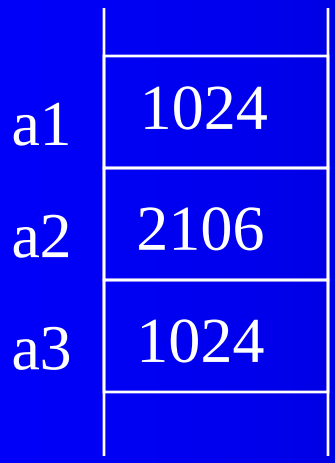
. . .

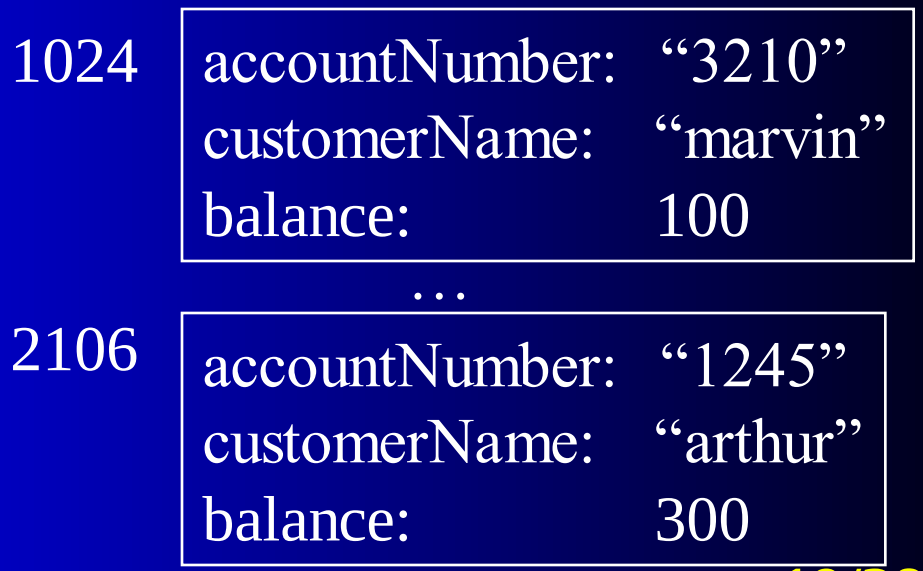
2106

accountNumber:	“1245”
customerName:	“arthur”
balance:	0

Object instantiation

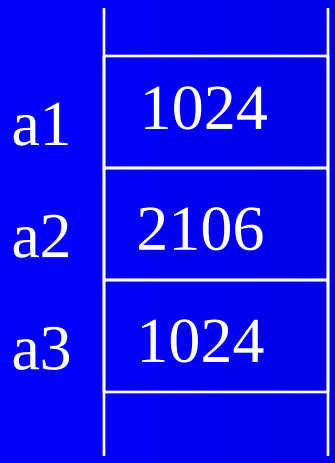
```
a1.deposit(100);
a2.deposit(300);
a3.withdraw(20);
```





Object instantiation

```
a1.deposit(100);
a2.deposit(300);
a3.withdraw(20);
```



1024

accountNumber:	“3210”
customerName:	“marvin”
balance:	80

. . .

2106

accountNumber:	“1245”
customerName:	“arthur”
balance:	300

Testing the BankAccount class

- Define a launcher class to test BankAccount
- This class has a main( ) method
- In the main method, we create instances of BankAccount and send messages to them

Testing

```
// Test 1 - create an account and display it
BankAccount a1 = new BankAccount( "A10", "Smith");
System.out.println(a1.toString( ) );
```

- *Observe the output*
- *Note that statement* System.out.println(a1) *would produce the same effect*

Testing

```
// Test 2 - make a deposit and display the account
a1.deposit(200);
System.out.println(a1);
```

- *Observe the output*

Testing

```
// Test 3 - make an invalid deposit request and see
// how the object handles it
a1.deposit(-100);
System.out.println(a1);
```

- *Observe how the object rejects the request*
- *Better ways of handling error and exceptional conditions will be learnt later*

Testing

```
// Test 4 - make a withdrawal
a1.withdraw(100);
System.out.println(a1);
```

Testing

```
// Test 5 - make an invalid withdrawal request
a1.withdraw(-100);
System.out.println(a1);
```

Testing

```
// Test 6 - make another invalid withdrawal request
a1.withdraw(300);
System.out.println(a1);
```

Testing

```
// Test 7 - get the balance
double balance = a1.getBalance( );
System.out.println("balance: " + balance);
```

- *Note that* balance *in the statements above is a local variable of the* main *method*

Testing

```
// Test 8 – display the account details
a1.displayAccountDetails( );
```

Class exercise - employees

- Create and test a class representing an employee that
  - Has a name and the details associated with their pay
  - Allows users to update the amount of hours they should be paid, and pay them
- Rules
  - Employees have a standard number of hours they work a week
  - They can work a standard week or a non-standard week

Defining the class Employee

- To define the Employee class and to test it, we go through the following typical steps
  1. Sketch a model of the class
  2. Define the class header
  3. Define the attributes
  4. Define the constructors
  5. Define the methods

Next lecture

- Information hiding and encapsulation
- Access modifiers
- Class interfaces
- javadoc