ECS510 “ADSOOF”

Using Objects

The `DrinksMachine` examples

Code vs Execution (Static vs Dynamic)

- Code (i.e. what we write when we write programs) is a description
- When we execute code on a computer, something is created according to that description
- In the algorithmic view, what is created is the computer’s performance of a task
- In the object-oriented view, what is created is “objects” which interact with each other

Classes vs objects methods vs method calls

- A class is a description of an object, a “blueprint” or a “recipe”
- An object is what is created when the code is run, like a machine built to a blueprint, or a cake made from a recipe
- Objects interact through methods
- A method is a description of an interaction
- A method call is what happens when the interaction takes place according to the description in the method

Parameters vs arguments

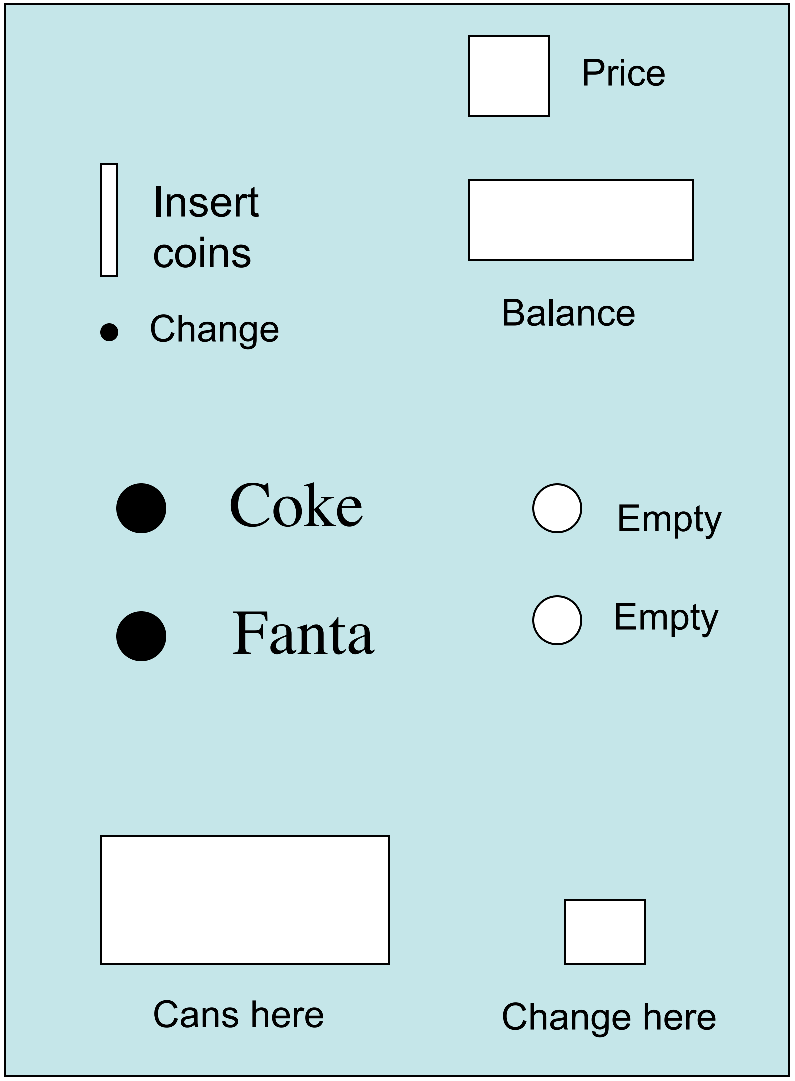
- A parameter is something in a description that can be varied to tailor what we build from that description
- For example, a recipe for a cake might have a parameter “fruit” we could fill in with whatever fruit we like, or a parameter “amount” we could fill in depending on what size cake we want
- An argument is the value we fill the parameter with when we use the description to make something

Constructors and methods

- The constructor of an object is the “blueprint” or “recipe” used to create it
- A class contains one or more constructors, the constructors have parameters used to tailor the objects created by them
- A method in a class describes an interaction with an object of that class, it may also have parameters used to tailor the interaction

Abstraction and example

- The classes we write in algorithms and data structures tend to be abstract, describing things like “list” or “set” or “comparison operator”
- To start off, though, we’ll look at something more “concrete”
- We can often be helped to understand abstract concept by considering more concrete examples
- The details of the examples are not important, the more more general principles they are trying to illustrate are



+ additional operations for machine owner

Objects as machines

- Many machines (objects) built to the same design (class)
- We can use the machine without knowing how it works inside
- But we know how it interacts with us (specification)
- We can only interact with it through its public interface (methods)

`DrinksMachine` methods

```
void insert(int n)
int getBalance()
int getPrice()
boolean cokesEmpty()
boolean fantasEmpty()
int pressChange()
Can pressCoke()
Can pressFanta()
void loadCoke(Can can)
void loadFanta(Can can)
void setPrice(int p)
int collectCash()
```

An object method may …

- Take arguments (inserting coin into machine)
- Return a value (show balance, show emptiness, deliver can)
- Change the state of the object it is called on (machine behaves differently next time)
- Change the state of an argument (no direct equivalent with drinks machine)

  The types tell us what sort of thing can be inserted where and what sort of thing will be shown/ delivered

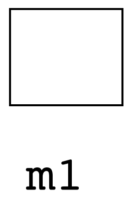
Take/return v. read/write

- Do not confuse “take as argument” with “read”
- or “return” with “write”
- although we sometimes (sloppily) say “input” and “output” for both
- In this course we are concerned with methods interacting with other methods - invisibly
- Good programming practice is that the part of program which interacts with outside world (GUI/ files/network) is separate from the rest (Model-View-Controller)

Objects and variables

In Java, a variable name always <u>refers to</u> an object of its type (or is `null`)

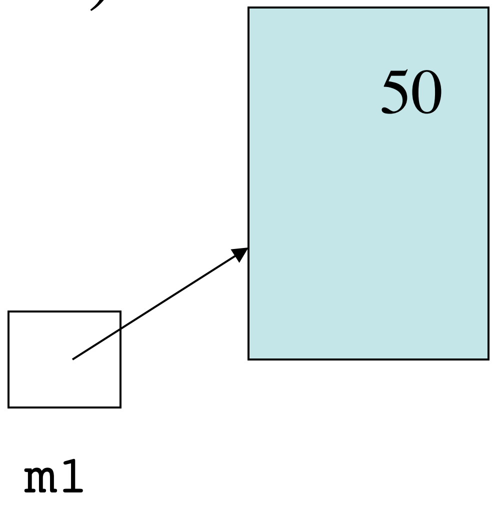
```
DrinksMachine m1;
```



Objects and variables

In Java, a variable name always refers to an object of its type (or is `null`)

```
DrinksMachine m1;
```



```
m1 = new DrinksMachine(50);
```

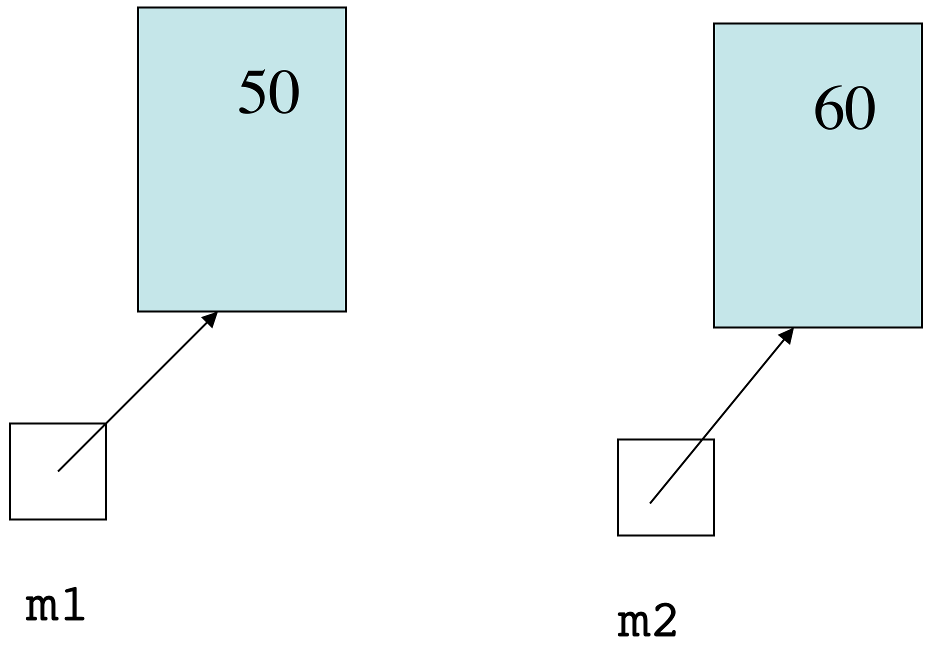
Constructors

- A constructor creates a new object of its class
- A constructor is called by the word `new` followed by the name of the class followed by arguments

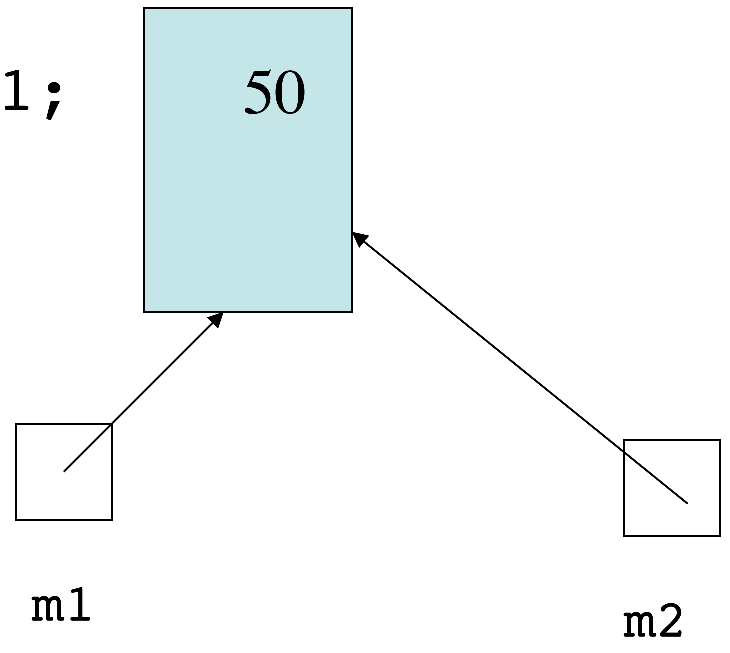
`DrinksMachine` constructors

```
DrinksMachine(int p)
DrinksMachine(int p, int c, int f)
```

```
DrinksMachine m1;
m1 = new DrinksMachine(50);
DrinksMachine m2 = new DrinksMachine(60);
```



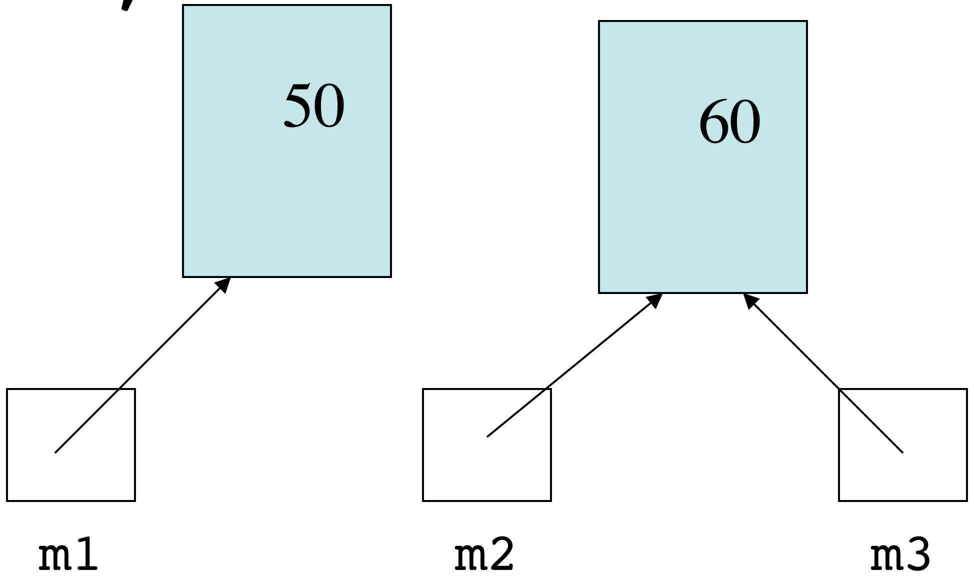
```
DrinksMachine m1;
m1 = new DrinksMachine(50);
DrinksMachine m2 = new DrinksMachine(60);
…
m2=m1;
```



Aliasing

before

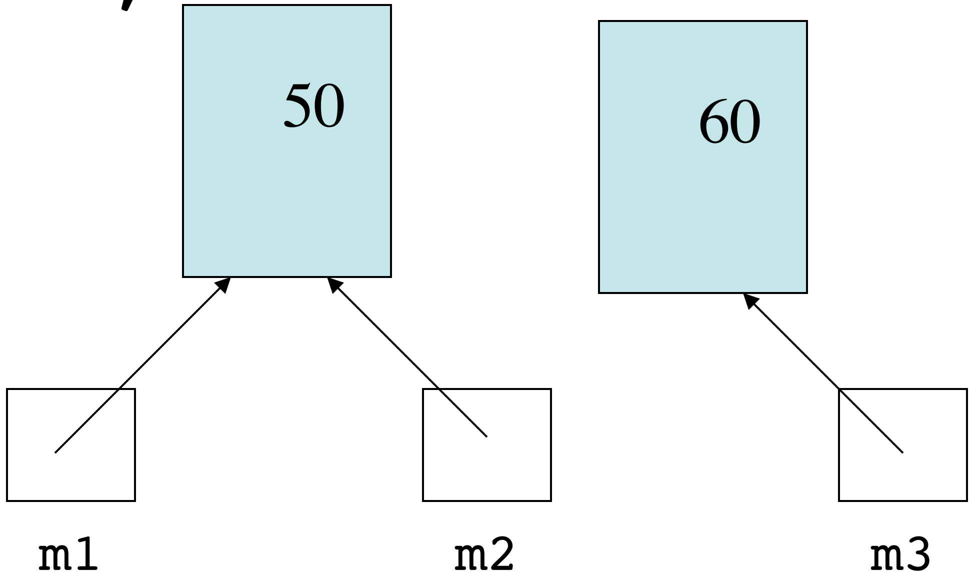
```
m2=m1;
```



Aliasing

after

```
m2=m1;
```



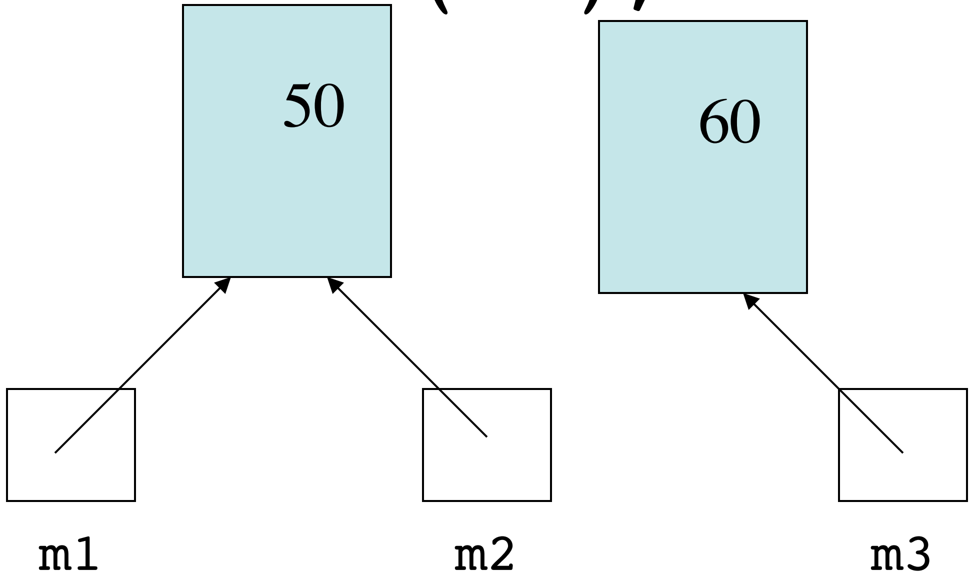
Method calls

- A call to a non-static method must be "attached" to a reference to an object, usually a variable name
- The class of the variable says what methods can be called on it
- The method works on the object the variable it is called on refers to
- The arguments in the method call must be of the correct types as given by its signature

Side effect

before

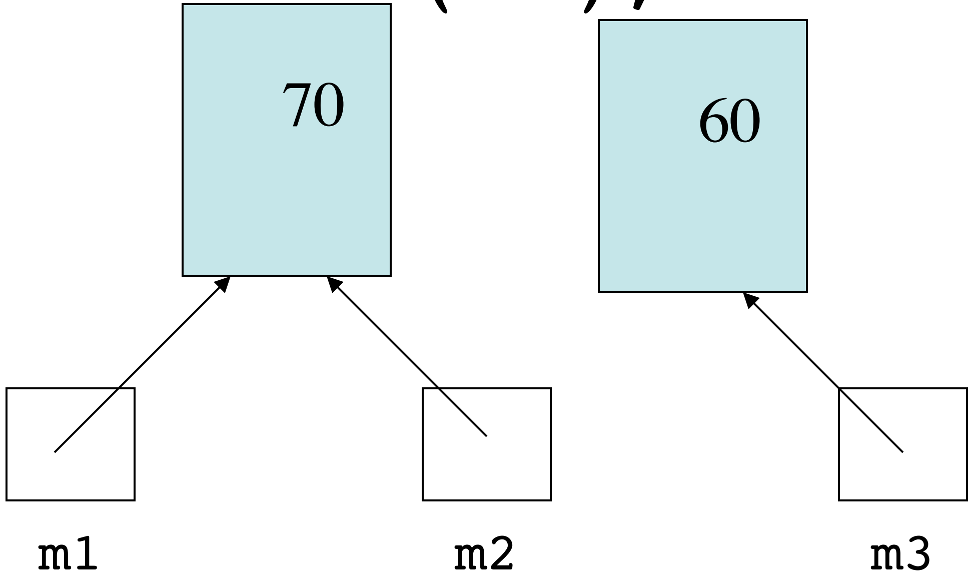
```
m1.setPrice(70);
```



Side effect

after

```
m1.setPrice(70);
```



Mutable v. Immutable

- If a class has no methods which can change the state of objects of that class, objects of that class are called **immutable**.
- Otherwise, objects are mutable
- Immutable objects do not suffer from the problem of side-effects
- A method call with particular arguments on an immutable object will always return the same value

Return values

- A method call which changes the state of the object it is called on can be a statement on its own:

```
m.setPrice(70);
```

- A method call which returns a value is normally used as part of a statement which uses that value:

```
int p=m.getPrice();
…
System.out.println(m.getPrice());
```

Static methods

- When a static method is called, it is not attached to an object
- A static method call runs in its own <u>environment</u>
- This means it uses only its own variables named from the method header and local declarations
- Each static method call creates a new environment, there is no connection between variables in this environment and variables of the same name in another environment (even for the same method)

How static method calls work

- Method call:

```
cans = spendOnCokes(amount,machine);
```

- Method header (signature):

```
public static
 int spendOnCokes(int sum,DrinksMachine mach)
```

- Effect:

  `sum=amount;`

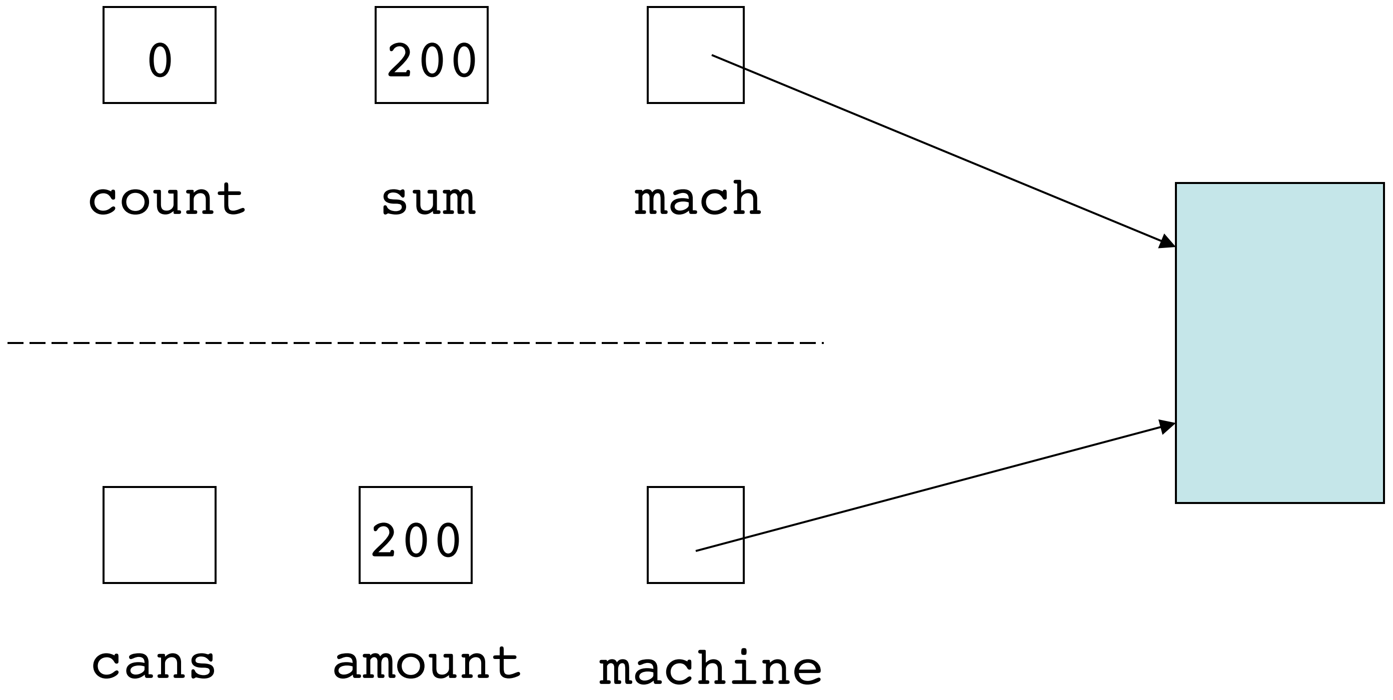
  `mach=machine;`

  obey code in method using new environment

  `cans =` *return value*

```
public static int
spendOnCokes(int sum,DrinksMachine mach)
{
 int count=0;
 mach.insert(sum);
 while(!mach.cokesEmpty()&&
     mach.getBalance()>=mach.getPrice())
    {
     mach.pressCoke();
     count++;
    }
 return count;
}
```

How method calls change object arguments



Scope of objects and factory methods

- An object remains in existence so long as at least one variable refers to it
- An object created in a method will remain in existence if (a reference to) it is returned from the method:

  ```
  m1 = cheaperBy10p(m2);
  ```
- A method which creates and returns an object is sometimes known as a “factory method”

```
public static DrinksMachine
 cheaperBy10p(DrinksMachine mach1)
{
 int p = mach1.getPrice()-10;
 DrinksMachine mach2 = new DrinksMachine(p);
 return mach2;
}
```

Constructive v. Destructive

Be careful to distinguish between the previous method and the following:

```
 public static void
 reduceBy10p(DrinksMachine mach1)
{
 int p = mach1.getPrice()-10;
 mach1.setPrice(p);
}
```

The previous method constructed and returned a new `DrinksMachine` object, this method changes the actual `DrinksMachine` object passed to it.

Method calls leading to aliasing

- A method call may return (a reference to) an object that was passed to it as an argument:

  `mach3 = cheaper(mach1,mach2);`
- This leads to aliasing, here `mach3` could be an alias of `mach1` or `mach2`
- We can test for aliases:

  `if(mach3==mach1) …`
- Note == with objects <u>only</u> tests for aliasing, `mach3.equals(mach1)` <u>may</u> be more general

```
public static DrinksMachine
cheaper(DrinksMachine m1,DrinksMachine m2)
{
 if(m1.getPrice()<m2.getPrice())
    return m1;
 else
    return m2;
}
```

Exceptions

`c = mach.pressCoke();`

with `c` of type `Can`, `mach` of type `DrinksMachine` normally returns a (reference to a) `Can` object

How do we represent “no can returned”?

- Return `null` - simple, but poor Java
- Throw an exception - good Java

- Signature of method which throws exception:

```
Can pressCoke() throws EmptyMachineException
```

- Code for catching exception:

```
try {
     …
     c = mach.pressCoke();
     …
    }
catch(EmptyMachineException e)
    {
     …
    }
```

Checked and unchecked exceptions

- Checked exceptions <u>have</u> to be caught in a try-catch statement, or indicated as thrown in the method signature
- Unchecked (“run-time”) exceptions can be caught in a try-catch statement, but it’s not compulsory

Using objects

- This section concentrated on using objects, nothing on writing classes to define objects
- Most of it should have been revision
- Practice on the code examples to make sure you understand the principles
- Generalise - `DrinksMachine` was used to illustrate more general principles