Programming Foundations FIT9131

Object creation and interaction

Week 4

Lecture outline

- **Design :**
  - **Abstraction & modularisation**
  - **Class and object diagrams**

- **Object Interactions :**
  - **Primitive types and object types**
  - **Creating new objects**
  - **Passing information between objects**
  - **Formal parameters and actual arguments**
  - **Return values from methods**
  - **Method overloading**
    - **Multiple constructors**

Abstraction and models

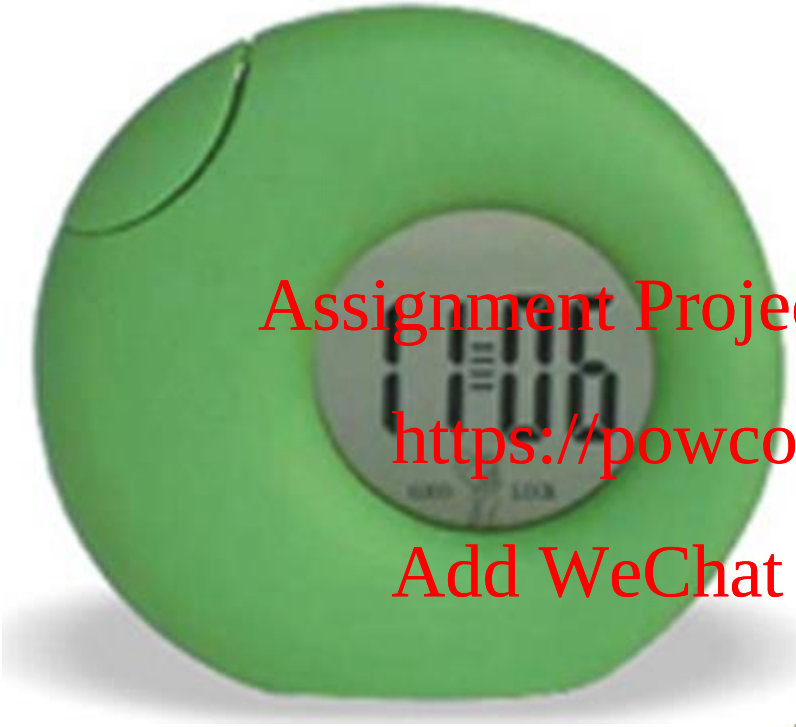
- *Abstraction* is the ability to ignore details of parts to focus attention on a higher level of a problem.
- In a program, we choose the attributes and behaviour that are relevant to the problem we are trying to solve – and we ignore the rest.
- Using the process of abstraction we create a *model* that is a simplification of the real situation, strictly relevant to the problem to be solved. E.g. imagine if you are booking an movie ticket online – what information do you need to provide to the cinema's booking system? What information is not needed?

Modularisation

- *Modularisation* is the process of dividing a whole entity (eg. a problem to be solved) into smaller and well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
  E.g. consider the process of designing a car – is it better/easy to design and build the entire car at once, or design the components individually and then combining them?

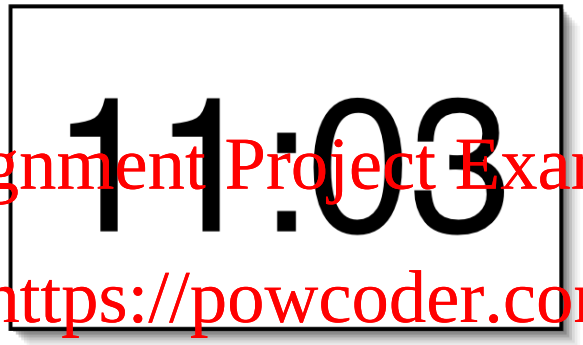
- In programming, the techniques of *Modularisation* and *Abstraction* are often used to deal with complexity. As programs become larger and more complicated these techniques become increasingly important.

Example : Digital clocks



Two different designs – but we can see some obvious ***common properties***

Design : A simple digital clock



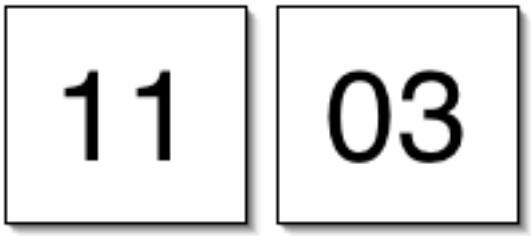
What attributes would a clock have?
What behaviour would you want for a clock?

Modularising the clock display

Choices :

One four-digit display?

Or, 2x two-digit displays?



Implementation - NumberDisplay

```
// represents a 2-digit display
public class NumberDisplay
{
    private int limit;
    private int value;

    Constructor and
    methods omitted...
}
```


note how these consist of 2 x **int** type attributes : one to represent the **2-digit** display, and another to represent an **upper limit** for the digit display.

This example is in Chapter03 of the sample projects.

Implementation - ClockDisplay

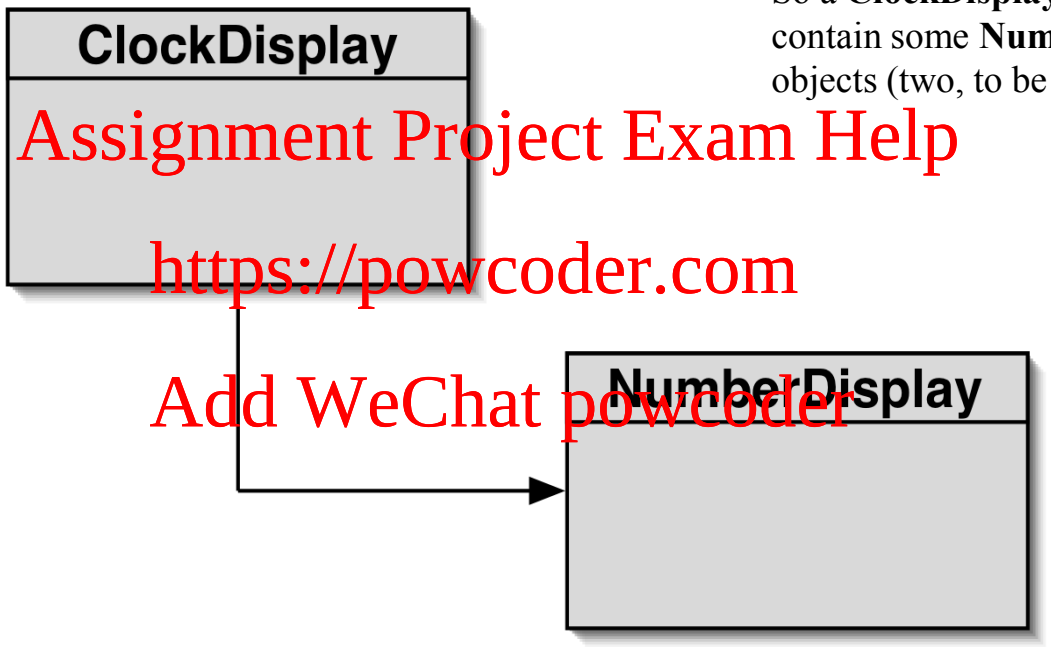
```
// represents a Clock, which consists
// of 2 x 2-digit displays
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    Constructor and
    methods omitted...
}
```

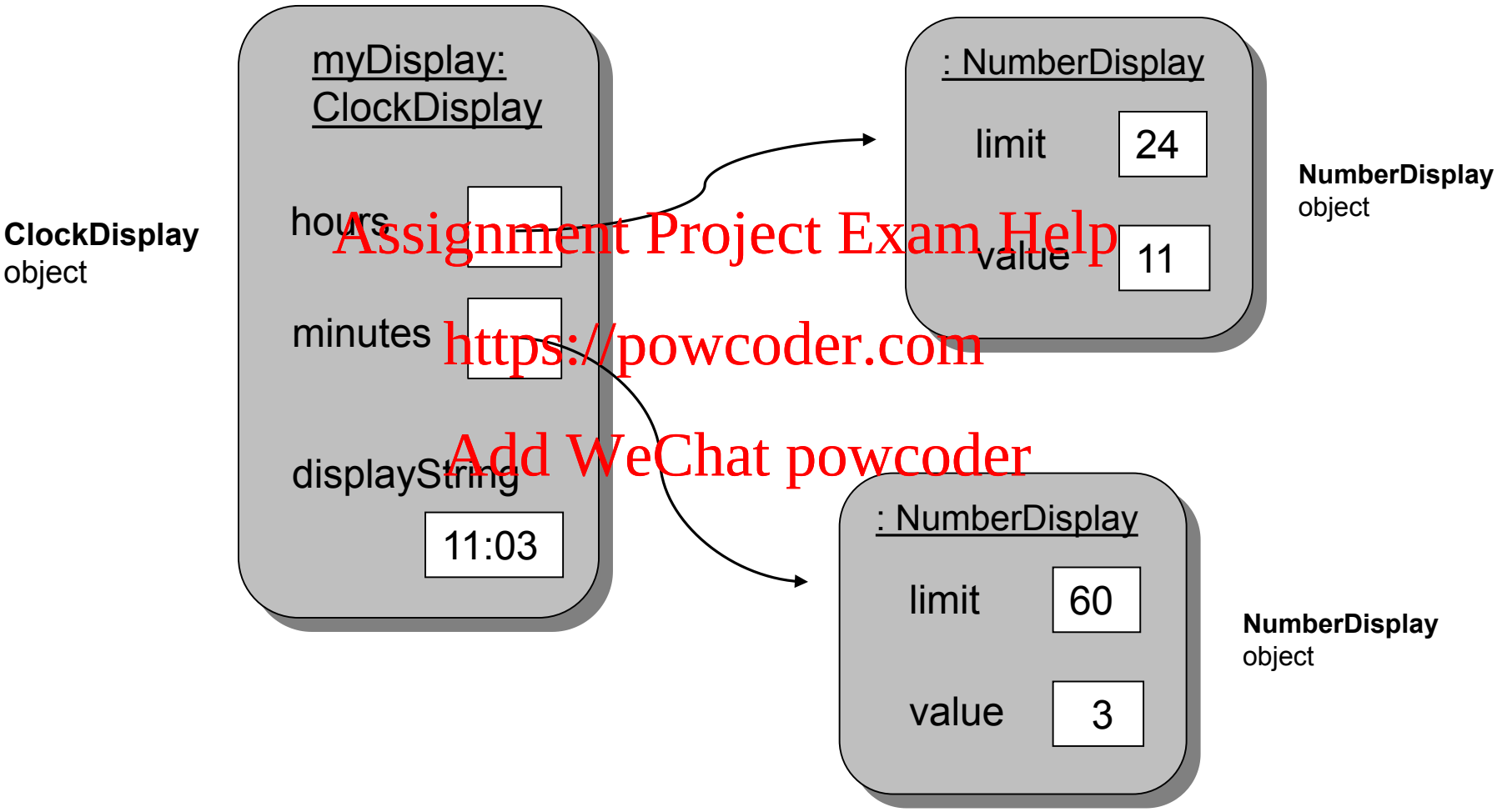

note how these consist of 2 x **NumberDisplay** type attributes

Class relationship diagram

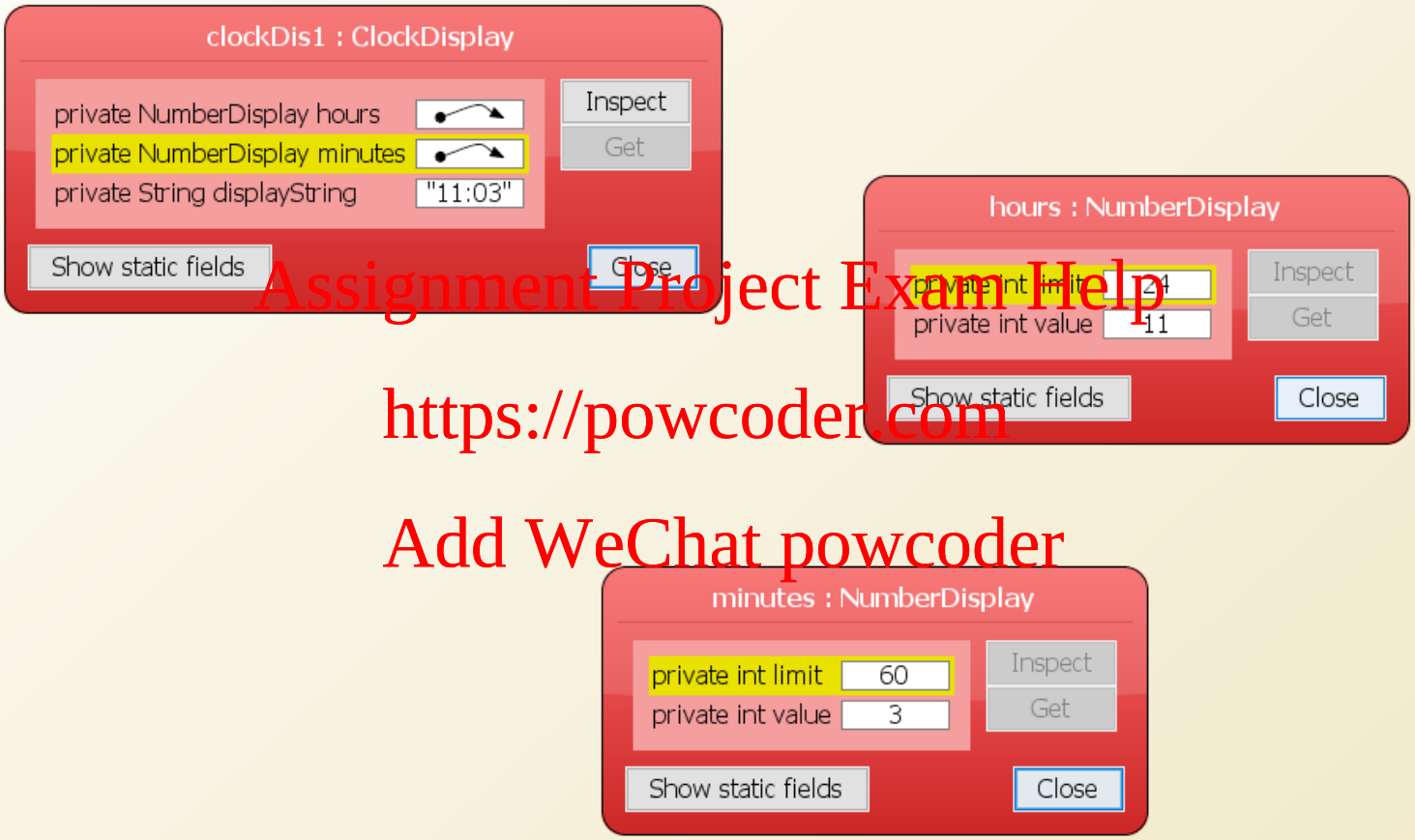
So a **ClockDisplay** object will contain some **NumberDisplay** objects (two, to be exact)



Object diagram



Object diagram (BlueJ Inspector)



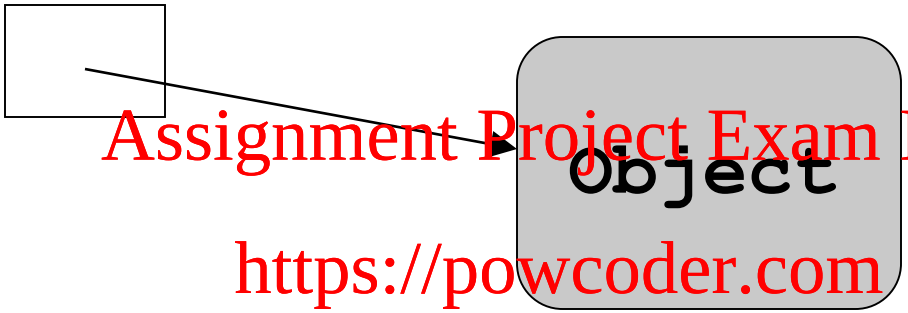
Variables of primitive types vs. object types

A *primitive type* is one of the Java built-in basic data types, e.g. **int**, **double**, **char**, **boolean**. The name of the variable is a memory location that stores the actual *value* of that variable.

An *object type* may be pre-defined in the Java system (e.g. **String**) or user-defined (e.g. **NumberDisplay**). The name of the variable is a memory location that stores the *address* of that object (which stores the values of its fields). These are also called *reference variables*.

Primitive types vs. object types

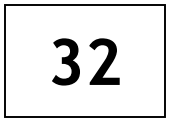
```
ObjectType a;
```



note : the variable name is a ***reference*** to the actual object

object type (any object type, eg. String, Student, etc)

```
int a;
```



note : the variable stores the ***actual value***

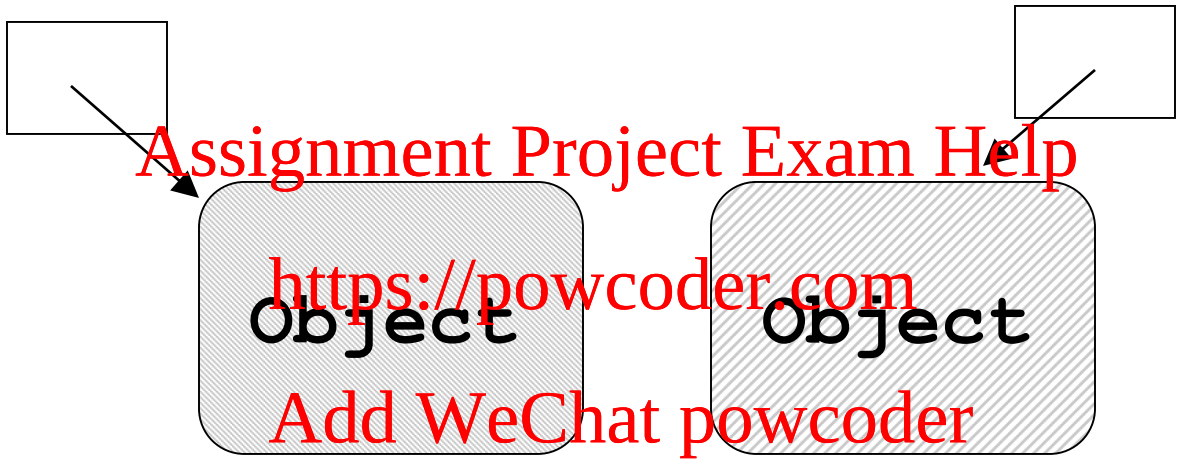
primitive type

Primitive types vs. object types

Consider these 2 variables :

```
ObjectType a;
```

```
ObjectType b;
```



---

and compare them to these 2 :

```
int a;
```

```
int b;
```

Primitive types vs. object types

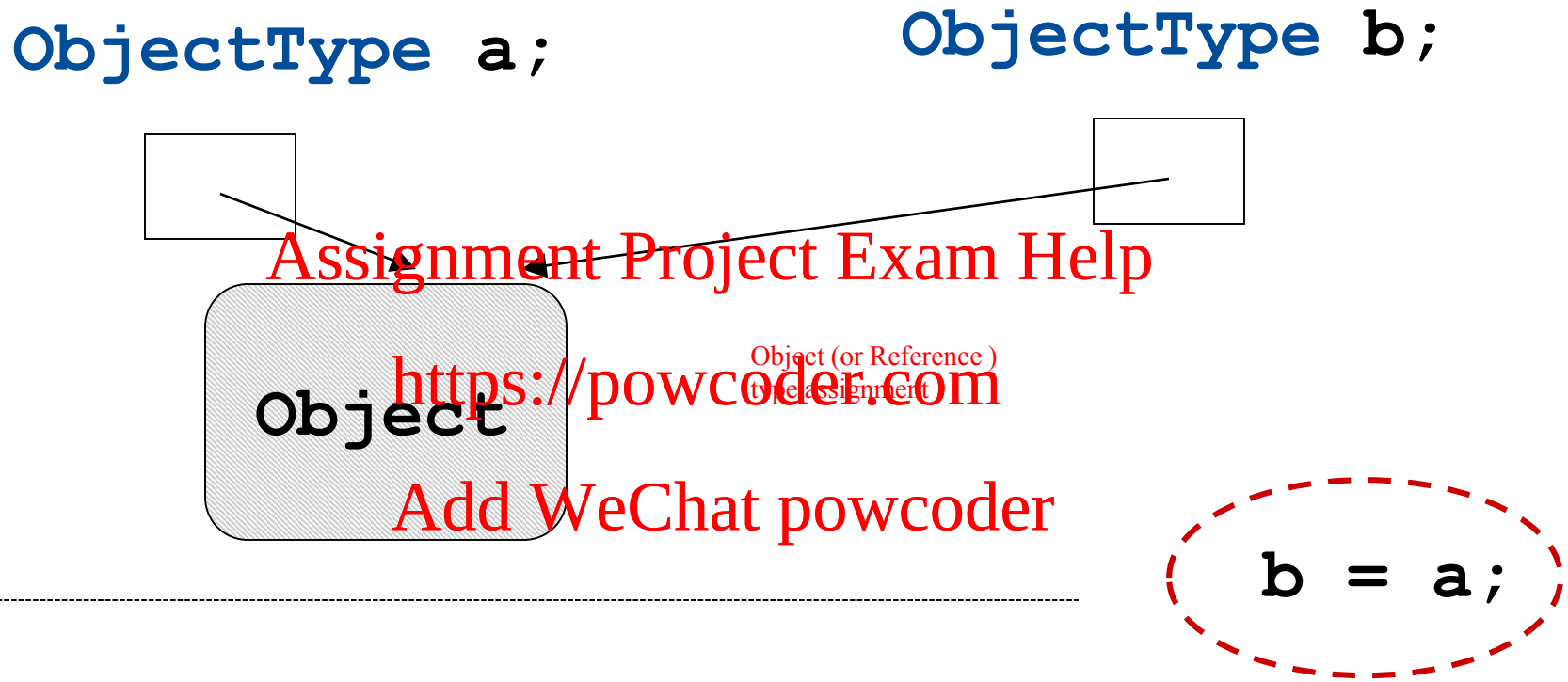
On the previous slide, we had 4 variables, 2 of primitive types (`int`) and 2 of object types (`ObjectType`).

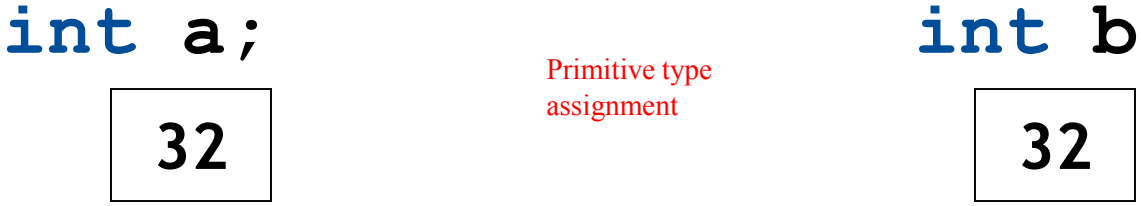
What would happen if we now execute this code :


```
b = a;
```


for the primitive variables, versus for the object variables?

Results of the assignments





Source code: NumberDisplay

```
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}

public void increment()
{
    value = (value + 1) % limit;
}
```


What is happening here?

Source code: NumberDisplay

```
public String getDisplayValue()
{
    if (value < 10)
        return "0" + value;
    else
        return "" + value;
}
```


What is happening here?

Objects creating other objects

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
}
```

declaring other objects as attributes

initialising those attributes, by creating 2 new objects

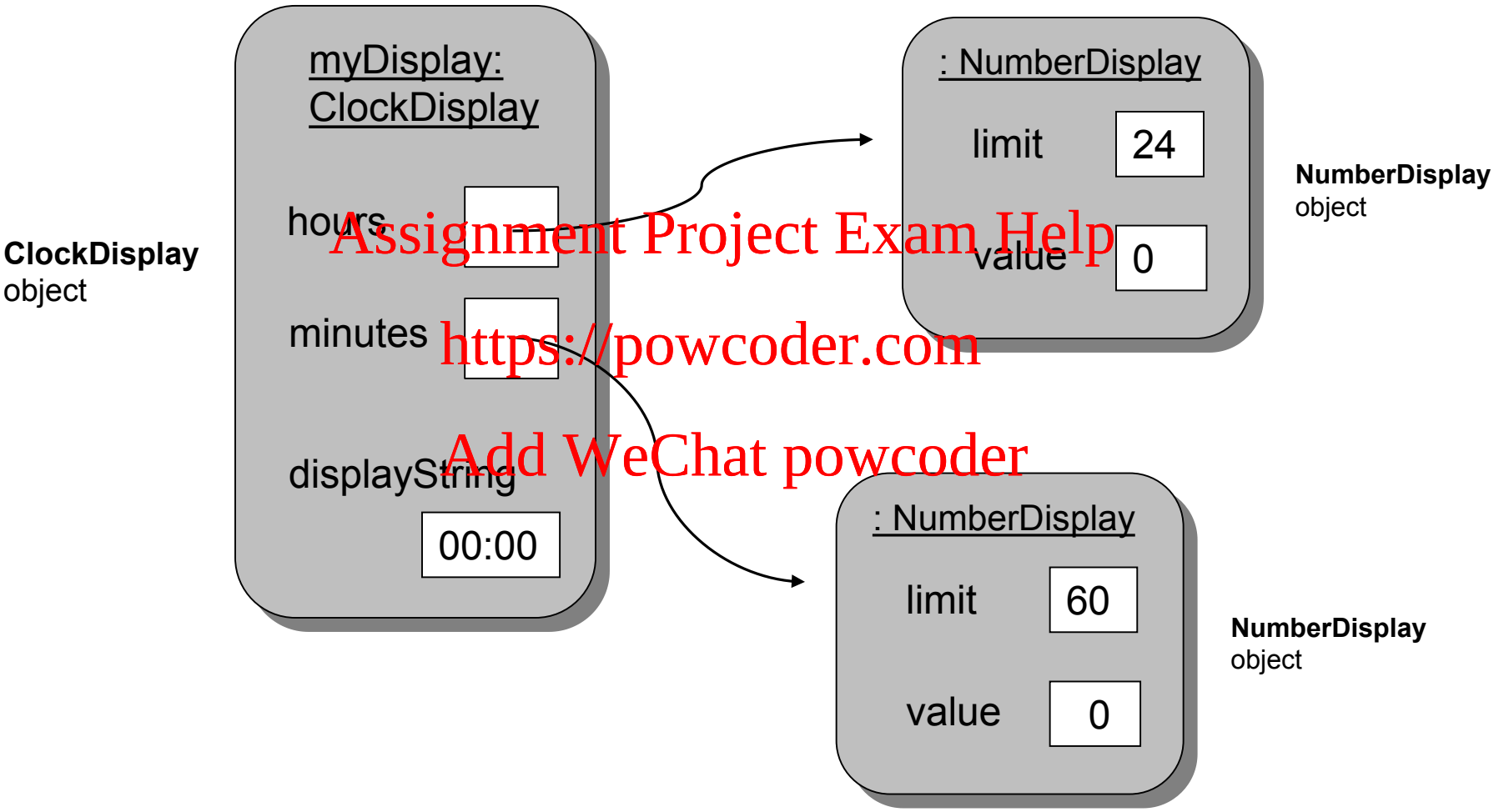
Calling the **NumberDisplay** constructor twice, with different values

What does this mean???

On the previous slide, we declare that two of the attributes of the class `ClockDisplay` are of type `NumberDisplay`. We then initialise those attributes by creating two `NumberDisplay` objects within the `ClockDisplay` constructor.

What this mean is : whenever a `ClockDisplay` object is created, it actually contains 2 `NumberDisplay` objects (as shown on the next slide) as its attributes.

Object diagram



ClockDisplay object

NumberDisplay object

NumberDisplay object

Formal Parameters versus Actual Arguments

in class NumberDisplay:

```
public NumberDisplay(int rollOverLimit)
```


formal parameter

in class ClockDisplay:

```
hours = new NumberDisplay(24);
```

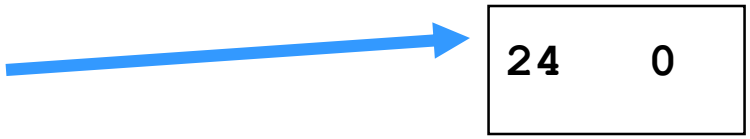
actual argument

Creating a new object

When we instantiate an object from a class, we use the `new` operator. Eg :

```
new NumberDisplay(24)
```


This statement allocates a block of memory big enough to hold a `NumberDisplay` object, and calls the constructor to initialise the values of its fields. The `new` operator returns the *address* of the beginning of that block of memory.



Naming a new object

Usually, we want to give an object a name so that we can refer to it afterwards. E.g :

```
hours = new NumberDisplay(24);
```


This statement puts aside enough memory to hold the address of a NumberDisplay object, and "name" that memory location `hours`.

We say that `hours` is a *reference* to the actual `hours` object. This is also called a **reference variable**.

Object interaction

Objects communicate with each other by sending messages. There are three components of a message sent to an object:

- the **name** of the object that is the receiver of the message
- the **action** that the receiver is requested to take
- in parentheses, any extra **information** the receiver needs to know

In Java, the *syntax* for a message depends on whether it is an *internal* or *external* method

Method calling example

Consider this method in **ClockDisplay**

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0)
        // it just rolled over!
        hours.increment();
    updateDisplay();
}
```

External method call

Internal method call

Method calls

internal method call – when the method is called from within another method of the same class, e.g.

```
updateDisplay();
```

`updateDisplay()` is in the same class (`ClockDisplay`) as `timeTick()`

external method call – when calling a method of a different class, e.g.


```
minutes.increment();
System.out.print(minutes.getValue());
hours.increment();
```

These methods are from a different class (`NumberDisplay`). External method calls use the dot (.) notation :

object . *methodName* (*parameter-list*)

Passing information in a message

When an object needs some extra information in order to fulfill a request, this information can be passed with the message.

The extra information that is passed is called *actual arguments*. This information is transferred to the *formal parameters* defined in the method header.

We have seen these many times before.

Example : Formal parameters

A method has a *header* and a *body*.

Consider the `setValue` method of the `NumberDisplay` class:

```
public void setValue(int replacementValue)
{
    if ((replacementValue >= 0) &&
        (replacementValue < limit))
        value = replacementValue;
}
```

← *Method header*

} *Method body*

This method has a *formal parameter* called `replacementValue` which is an `int`. *When this method is called, it expects to be passed an integer value.*

`replacementValue` is called a *formal parameter*. It is a placeholder for a value that will be supplied when the method is invoked.

Example : Actual arguments

The method is called by using a `NumberDisplay` object and asking it to invoke the method, eg :

```
hours = new NumberDisplay(24);
hours.setValue(12);
```

The integer values 24 & 12 are the *actual arguments* that were given to the methods when they are invoked.

Another example :

```
int newValue = 12;
hours.setValue(newValue);
```


The `setValue` method sees only the value 12. It does not see the name of the variable (ie. `newValue`) that contains the value of the actual argument.

Methods with multiple parameters

The method `setTime(...)` in `ClockDisplay` changes the fields of a `ClockDisplay` object.

```
public void setTime(int hour, int minute)
{
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}
```

2 formal parameters

A call to this method would need to pass two **actual arguments**, e.g.

```
aClockDisplay.setTime(13, 49);
```

Rules for Matching formal parameters with actual arguments

The actual arguments passed to a method must match the formal parameters in *all* of the following:

- *number*
- *type*
- *order*

The compiler assumes that the first actual argument matches the first formal parameter, the second actual argument matches the second formal parameter, etc.

The names of formal parameters do not have to match the names of the actual arguments. The called method does not see the names, it sees only the values being passed in.

For example, on the previous slide, the parameter `hour` will be assigned the value 13 and `minute` will be assigned 49.

Passing basic data types as parameters

If a parameter is of a primitive data type, when the formal parameter comes into existence it is a *copy* of the actual argument.

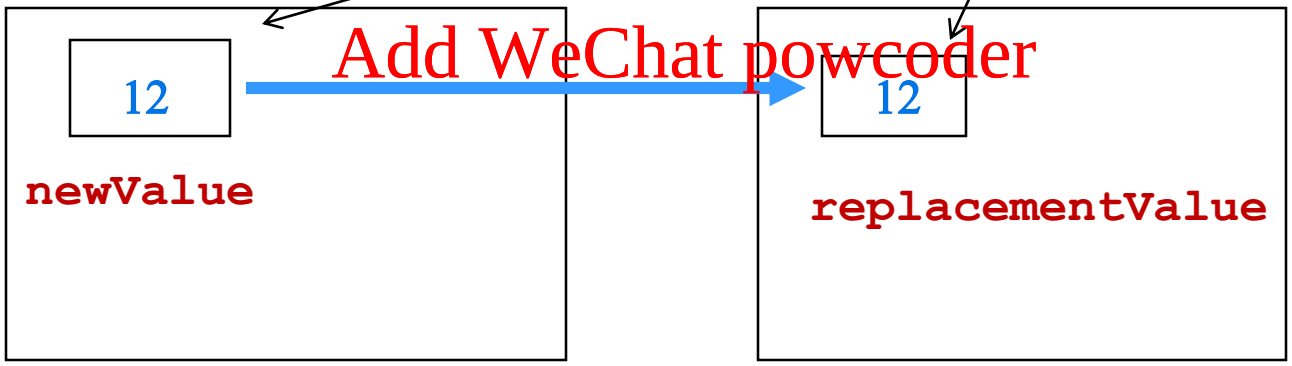
Therefore, any changes made to the formal parameter inside the method will not affect the value in the caller's memory area. This kind of parameter passing is known as *call-by-value*.

Example : Call-by-value

For example, if the `setValue` method is called by some other method, like this :

```
int newValue = 12;
hours.setValue(newValue);
```

note : these are 2 separate "variables" – they just happen to have the same value as they are copies of each other

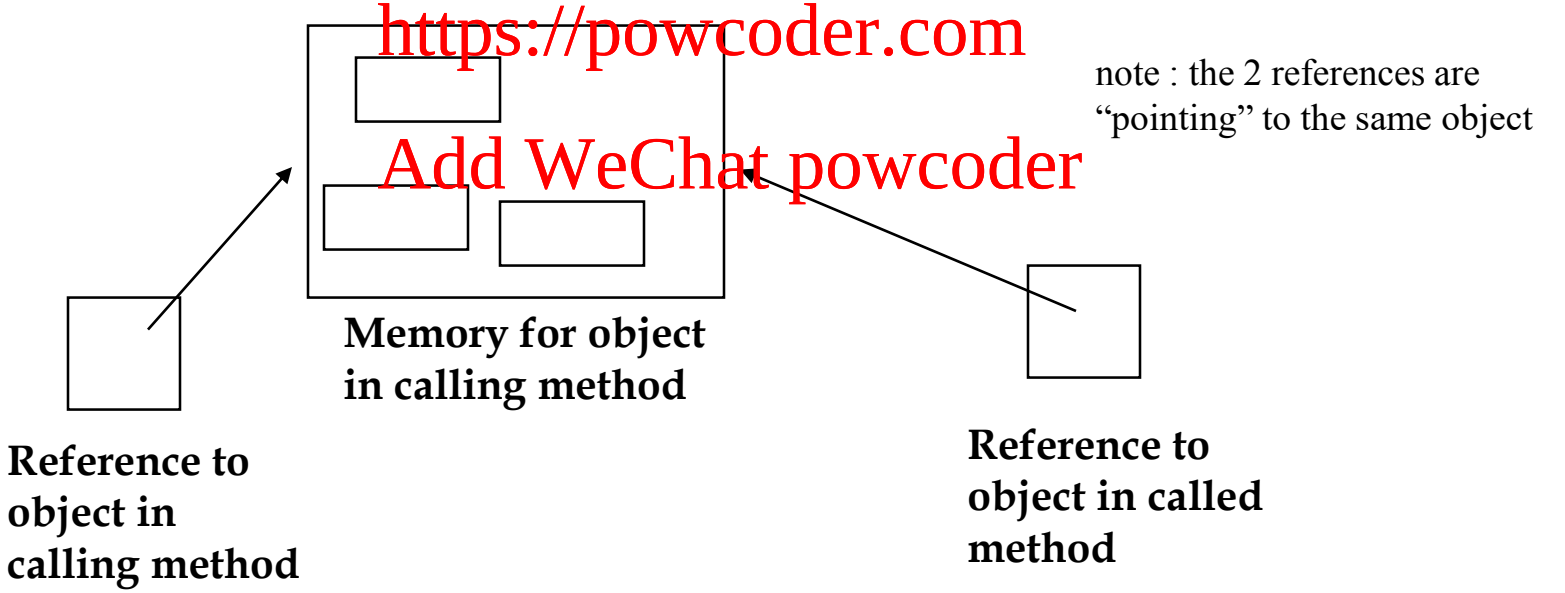


Memory belonging to the other (calling) method

Memory belonging to `setValue`

Passing objects as parameters

When an object is passed as a parameter, what is passed to the method is a *copy of the reference* to the object. A change to the state of the formal parameter produces a change in the state of the actual argument. This kind of parameter passing is known as *call-by-reference*.



Returning values from methods

A value returned from a method *replaces the call to the method*. Eg :


```
NumberDisplay aNumberDisplay = new NumberDisplay();
System.out.println(aNumberDisplay.getValue());
```

becomes


```
  System.out.println(0);
```



```
ClockDisplay aClockDisplay = new ClockDisplay();
if (minutes.getValue() == 0)   ...
```

becomes

```
  if (0 == 0)   ...
```

becomes

```
  if (true)   ...
```

Methods with no parameter and no return values

Eg : this method in the `ClockDisplay` class:

```
public void updateDisplay()
{
        displayString =
              hours.getDisplayValue() + ":" +
               minutes.getDisplayValue();
}
```

no formal parameter

This method can be invoked by an expression like :

```
updateDisplay();
```

no actual argument

Method signatures

A method has a *signature* that specifies what parameters it takes when it is called.

```
public int getAge()
public String getName()
public boolean setAge(int anAge)
private boolean validAge(int anAge)
```


The signature is the method name plus the parameter list. The compiler cares only about the types in the parameter list, not the names of the formal parameters.

Method Overloading

It is possible to have more than one method (or constructor) with the same name, but different signatures, in a class declaration. This is called *method overloading*.

The compiler knows which method you want to call by looking at the signature of the methods. If it can't find a signature that matches the way you have called that method, it gives you an error message.

An example of overloading can be seen with the multiple constructors in the `ClockDisplay` class :

```
public ClockDisplay()
public ClockDisplay(int hour, int minute)
```

Default constructor of `ClockDisplay` *class*

```
public ClockDisplay()
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    updateDisplay();
}
```


NB. A Default Constructor has no formal parameter(s)

If you do not write a constructor for a class, Java puts one in for you. Good programming practice – always write your own constructor!

Another `ClockDisplay` constructor ("overloaded")

```
public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```


As long as the signatures are all different, the compiler has no problem with these.

Overloading other methods

We can also overload any other methods in exactly the same way, as long as the two methods with the same name have different signatures.

Eg - consider this method :


```
public boolean setGrade(char aCharacter)
{
    if (validGrade(aCharacter)) // check if parameter is valid
    {
        grade = aCharacter; // assume grade is an attribute
        return true;
    }
    return false;
}
```

An overloaded version of setGrade

```
public boolean setGrade(String aString)
{
    char aCharacter = aString.charAt(0);
    if (validGrade(aCharacter))
    {
        grade = aCharacter;
        return true;
    }
    return false;
}
```

Returns the first character in the string

OR, more efficiently (same result):

```
public boolean setGrade(String aString)
{
    char aCharacter = aString.charAt(0);
    return setGrade(aCharacter);
}
```

Which method is called?

When you call the `setGrade` method, Java will pick one version or the other to execute, depending on what type of data is passed to it. eg :

```
setGrade('P');
```


will cause the first version to be executed, and


```
setGrade("Pass");
```


will cause the second version to be executed.

As the two versions have different signatures, there is no confusion for the compiler.

The this keyword

The this keyword is used resolve name conflicts. It means "*the current object*" i.e., the object that has been asked to invoke this method.

It can be used to access a attribute (field) when there is a more 'closely' defined variable with the same name. For example, in the following constructor the three parameters have the same names as the attributes – so the code would not work :


```
public MailItem(string from, String to, String message)
{
   from = from;
   to = to;
   message = message;
}
```

Problem : Java cannot differentiate between the attributes and the parameters, since their names are the same

The this keyword

We need something to differentiate the attributes from the formal parameters :

```
public MailItem(string from, String to, String message)
{
   this.from = from;
   this.to = to;
   this.message = message;
}
```


No confusions here!!!

The `this.from` refers to the `from` attribute in the current object, etc.

The `from` refers to the `from` formal parameter in the current method, etc.

Cleaner solution?

A cleaner (& recommended) solution is : use different names for the attributes and the formal parmeters – eg. we could have named the parameters above "`sender`", "`receiver`", "`newMessage`". Then we wouldn't need to use the `this` keyword at all here.

Eg. Use different names for different things:

```
public MailItem(String sender, String receiver, String newMessage)
{
   from = sender;
   to = receiver;
   message = newMessage;
}
```

Even less confusions here!!!

Assignment 1

Hopefully all of you have downloaded Assignment 1.

Progress:

- From Week 2, class skeletons can be done for all classes.
- Classes with arrays can use single value attributes for now.
- From Week 3, input can be accepted via the keyboard for the classes.
- From this week, objects can be created within code to facilitate interaction.

- Demo, if time permits.