Resources

- More help with this course
  - Moodle `http://moodle.bath.ac.uk/course/view.php?id=30475`
  - E-mail - programming1@lists.bath.ac.uk
- Online C and Java IDE
  - `https://www.codechef.com/ide`
  - Remember to select Java from the drop down menu.

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - The A labs
  - One remaining B lab
  - The Drop in Sessions
- please note that some of the labs were previously scheduled at awkward times
- to the (limited) extent that we could, we have rearranged the awkward labs in order to make them more accessible
- please check the details on Moodle (and let us know if you cannot now get to a lab that you would otherwise have attended)

- If you struggling with the exercises, pace of the course and/or coding in general
- Please come and see Rachid or Michael

- If, on the other hand, you are finding the pace a little slow
- You can sign up for the Advanced Programming Labs
- When and Where
  - Friday 11.15 - 13.15
  - EB 0.9

Last Week

- Complex Collections (In C)
- Abstract Data Types (In C)

This Week

- First Objects (In Java)
- Interacting Objects (In Java)
- Short Feedback Session

- This week marks a major turning point in the course
  - We will move from using C
  - To using second programing language: Java

- That change means that you will have to learn
  - new ways to write the instructions that we want the computer to execute and
  - a new way to think about the programs that we write whilst using those instructions: Object Oriented Programming

- As you make these changes, however, it is worth noting that you are not starting from scratch
  - When writing small Java programs, the programs that you write can appear reasonably similar to those that you wrote in C
  - The Java version of HelloWorld, for example, bears some similarity to the C version

```
public class HelloWorld {

    public static void main(String[] args) {

        //print HelloWorld to the terminal
        System.out.println("Hello, World");
    }

}
```

- To some extent, you will also see similarities between
  - the ways in which you compile and run a Java program and
  - the ways in which you compile and run a C program
- These similarities can be misleading, however
  - On closer examination, the steps taken to prepare a Java program for execution
  - Are different from those taken when preparing a C program

The Programming Language Java

- The Java language is both compiled and interpreted.
- Instead of translating Java programs into a machine language, the Java compiler generates Java byte code for its Virtual Machine
  - Byte code is easy (and fast) to interpret, like machine language,
  - but it is also portable, like a high-level language.
- Thus, it is possible to compile a Java program on one machine, transfer the byte code to another machine over a network, and then interpret the byte code on the other machine.
- This ability is one of the advantages of Java over many other high-level languages.

Compiling and Running Simple a Program I

- We can use compilation/interpetation of the HelloWorld program on an earlier slide as an example:
- Before you start, the filename in which you save your main method should match the name of the class with the extension .java. In this case, HelloWorld.java
- Note: Java is case sensitive, just like C.

Compiling and Running a Simple Program II

- To run the code:
- we need to compile it: javac HelloPrinter.java
- This will generate a file HelloPrinter.**class**, containing the virtual machine byte code
- We can now run the code:

```
$javac HelloWorld.java
$java HelloWorld
Hello, World
```

Compiling and Running Programs Consisting of Multiple Classes

- If you write a program involving multiple classes,
- you would normally save each class in a different file
- Note: Each file must have a name of the form [classname].java
- where [classname] is the name of the class contained in the file
- If you have multiple files, you can compile tham all with one command.
- i.e. You can use javac $*$.java to compile all .java files in the current directory in one go.
- To run the program, you then need to use the interpreter command (java) on the class that contains the main method.
- We will return to multi-class programs later in the course

```
$javac *.java
$java HelloWorld
Hello, World
```

- Just as you did with C,
- We would strongly suggest that you adopt an iterative approach to developping Java programs i.e.
  - start with a program that works
  - (such as the Java version of HelloWorld introduced earlier)
  - and extend/debug it in small increments
  - until you have a program that does what you want

- In order to go beyond our HelloWorld program, however
- You will need to understand that Java is an Object Oriented (OO) language
- (Sometimes described as an Object Oriented Programming (OOP) language)

- This begs two obvious questions:
- What does it mean to be "Object Oriented" when writing code?
- And, in a programming context, what is an Object anyway?

- The (Data) Structures and Abstract Data Types (ADT's) that we considered last week provide us with a useful starting point when trying to understand Objects and Object Orientation.

- The US National Institute of Standards and Technologies (NIST) starts it's definition of a data structure as follows:
  - An organization of information, usually in memory, for better algorithm efficiency
- By this definition, an Array is a data structure.
- As we saw last week, an Array is an organisation (or organization) of information in memory

- The NIST definition of an abstract data type is as follows:
  - A set of data values and associated operations that are precisely specified independent of any particular implementation

- It is useful to focus on two parts of this definition:
  - "independent of any particular implementation"
    - ⋆ i.e. an abstract data type is defined once but can be implemented in multiple ways
  - A set of data values and associated operations
    - ⋆ i.e. an abstract data type defines both data and operations

- By this definition, a queue is a data type or collection
    - entities in the collection (values) are kept in order
    - and the principal (only?) operations on the collection are
        - the addition of entities to the rear
        - and removal of entities from the front
        - (first in, last out)

- Object Oriented Programming maintains this interest in associating data with the operations that can be performed on it:
- Object Oriented Programming: A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (methods) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions.
    - http://www.webopedia.com/TERM/O/object_oriented_programming_OOP.html

- C allowed us to use
  - data structures (Structs) and
  - functions which allow us to manipulate those structures (Structs)
- Java forces us to use
  - combinations of data structures and functions (Objects)
    - some of which are pre-defined for us
    - and some of which we can define ourselves

- Also, just as C allowed us to
    - define a data structure once
    - and then create multiple examples of that structure
- Java allows us to
    - define a combination of data and operations once (a Class)
    - and then create one or more examples (instances) of that class (Objects)

- More accurately, each time we want to use a new combination of data and operations, Java forces us to
  - start by defining a Class which combines that data and those operations
  - before creating one or more examples of that class (one or more Objects)
- You can think of this process as
  - defining a template (the Class)
  - and using it to create one or more Objects using that template

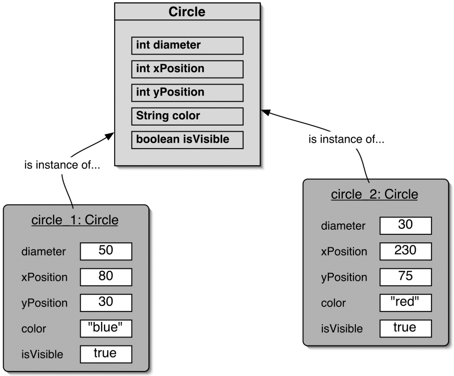
Exercise

TelephoneNumber	lord-of-the-rings
BankAccount	Diary
harry-potter-and-the-Philosopher-Stone	myDiary
01225-38-5053	michaelWright
Book	Lecturer
rachidHourizi	myAccount

Attributes and Fields

- Just as C Structures contained values stored in member variables
- an Object has attributes: i.e. values stored in fields. (The data you encapsulate)
- the class defines what fields an Object has, but each Object stores its own set of values. The set of values stored in a single Object is called the state of the object.

State


Circle
int diameter
int xPosition
int yPosition
String color
boolean isVisible
is instance of...
is instance of...
circle_1: Circle
diameter 50
xPosition 80
yPosition 30
color "blue"
isVisible true
circle_2: Circle
diameter 30
xPosition 230
yPosition 75
color "red"
isVisible true

Methods and Parameters

- Objects/classes also have operations which can be invoked. They are called methods
- If, for example, we wanted to be able to
  - draw examples of the circle class (Circle Objects) on screen and then
  - move those circles a specified distance to the right
- we could include
  - a drawCircle() method in our definition of the Circle class and
  - a moveRight() method in our definition of the Circle class

```
void moveRight(int distance){
        ...some code...
}
```

- The first piece of code used to implement a method e.g.
  - void moveRight(int distance)
- provides a short description of what the method does
- that short description is called the method's signature
- The collection of methods in a class is referred to as the interface of that class

- Methods are called or invoked
- At the point of being called, they may need to recieve information as input if they are to perform the desired operation
  - for example, our moveRight() method may need to be given a distance in order to move a Circle Object as intended
- As in C, that information can be passed in the form of one or more parameters
- in the moveRight(int distance) example, int distance is an example a parameter

Data Types

- Both Fields and Parameters have types. A type defines what kinds of values a parameter can take.
- In Java (as in C) you have to specify the type.
- e.g. we know that void moveRight(int distance) takes an integer as input because the distance parameter is defined as being of type int.

- In Java, everything has a type.
- Examples of types: int, String, Circle, . . .
- Java is staticly typed language
- i.e. once you have given a field or parameter a type, that type cannot change
- Note: Each time you create a new class, you have defined a new type

Source Code

- Each class has source code associated with it that defines its details (fields and methods).
- The code defining each class determines the structure and the behavior of each of its instances (i.e. the structure and behaviour of each Object created using the class template).
- This source code is compiled and interpreted by Java.

Return Values

- Methods may return a result via a return value.
- Example: String getName() This method returns a String.
- Example: **void** changeName() Void indicates that this method does not return anything

Developing Java Programs

- To learn to develop Java programs, one needs to learn how to write class definitions, including fields and methods, and how to put these classes together
- We will look at these issues in more detail during the remainder of this course

Coding Conventions

- Classes: Uppercase to start, merge words, consecutive words uppercase, nouns E.g. Car, Number, BankAccount
- Objects: Lowercase to start, merge words, consecutive words uppercase, nouns E.g. myBlueCar, Rational
- Methods: Lowercase to start, merge words, consecutive words uppercase, verbs E.g. moveLocation, deposit

Ticket Machines An External/User View

Exploring the behaviour of a typical ticket machine.

- Use the naive-ticket-machine project.
- Machines supply tickets of a fixed price.
- How is that price determined?
- How is money entered into a machine?
- How does a machine keep track of the money that is entered?
- How is a ticket provided?

Ticket Machine Fields and Initial State

```
public class TicketMachine
{
private int price = 500;
private int balance = 0;
private int total = 0;

... more code

}
```

Ticket Machine Methods: The Interface

```
public class TicketMachine
{

... fields ....

public int getBalance()
public int getPrice()
public void insertMoney()
public void printTicket()

...more code

}
```

Ticket Machines An Internal/Programmer view

- Interacting with an object gives us clues about its behavior.
- Looking at the source code allows us to determine how that behavior is provided or implemented.
- There is a recognised structure to the source code associated with each class.

TicketMachine Source Code

```
/**
 * TicketMachine models a naive ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * It is a naive machine in the sense that it trusts its users
 * to insert enough money before trying to print a ticket.
 * It also assumes that users enter sensible amounts.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002.02.06
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;


    //continued on next slide
```

```
//continued from previous slide

/**
 * Create a machine that issues tickets of the given price.
 * Note that the price must be greater than zero, and there
 * are no checks to ensure this.
 */
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}

//continued on next slide
```

```
    //continued from previous slide

    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }
}
```

Basic class structure

```
public class TicketMachine
{
    Inner part of
    the class omitted.
}
```

Basic class structure

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

Comments/Documentation

- As with C, Comments make Java source code easier to read for humans. No effect on the functionality.
- Three sorts:
  - // *comment*: single-line comments
  - /∗ *comments* ∗/: multiple-lines more detail
  - /∗∗ ∗/: similar to previous, but used when documentation software is used.

Fields

Recap:

- Fields store values for an object.
- They are also known as instance variables.
- Fields define the state of an object.
- Fields have an associated type.

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructor and methods omitted.
}
```

Constructors

- Constructors create and initialize an object.
- Then assign the necessary memory to the created object
- They have the same name as their class.
- They store initial values into the fields.
- They often receive external parameter values for this.

```
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}
```

Creating Objects

- Constructors are used to create and initialise a new object

```
public class TrainSimulator {

    public static void main(String[] args) {

        //create a new TicketMachine Object and store it in the variable machine
        TicketMachine machine = new TicketMachine(500);
    }

}
```

- This creates a new TicketMachine object and stores it in a variable named machine which is of type TicketMachine.

Parameters

- Parameter names inside a constructor or method are referred to as Formal Parameters
- Parameter values provided from the outside are referred to as Actual Parameters.
- In the constructor TicketMachine(**int** ticketCost)
- ticketCost is a formal parameter.
- When the constructor is called, using TicketMachine(500),
- 500 is an actual parameter.

Scope and Lifetime

- The scope of a variable/parameter defines the section of the code from which it can be accessed.
- For instance variables (fields) this is the entire class.
- For parameters, this is the constructor or method that declares it.
- Trick: find the enclosing {}, this is the scope.
- The lifetime of a variable/parameter describes how long the variable continues to exist before it is destroyed.

Assignment

- Similar to C:
- Values are stored into fields (and other variables) via assignment statements:
  - variable = expression ;
  - e.g. price = ticketCost ;
- Both sides of the assignment should have the same type, e.g. int, double, String, TicketMachine, ...
- A variable stores a single value, so (as in C) any previous value is lost after an assignment.

Accessor Methods I

- All methods implement object behaviour.
- And all methods have a structure consisting of a header and a body.
- The header defines the methods signature e.g.

```
public int getPrice()   {

     ... method body ...

}
```

- The body encloses the methods statements e.g.

```
... header{

    return price;

}
```

- We can, however, be more precise about the kind of behaviour implemented by particular methods
- Accessors, for example are methods which provide information about an object.
- i.e. they allow users, other programs or other parts of your program to gain information about an object's state

```
public int getPrice()
{
    return price;
}
```

Mutator Methods

- Mutators have a similar method structure: header and body.
- Used to mutate (i.e., change) an objects state.
- Achieved through changing the value of one or more fields.
    - Typically receive parameters.
    - Typically contain assignment statements.

Mutator methods

```
public void insertMoney(int amount)
{
    balance += amount;
}
```

Data Types

- As we have seen in the code examples above, Java (like C) provides us with common data types
- e.g. integers:
  - int distance =7;
- you won't be surprised to learn that Java also provides us with chars
  - char myChar ='c';
- and with booleans
  - boolean a = false;

Data Types

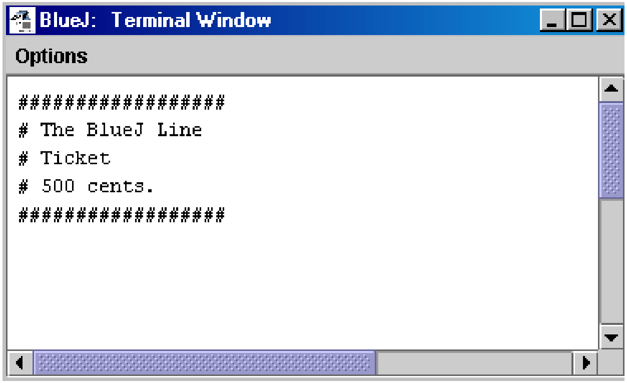
- Java also provides us with ways to implement Strings and Arrays
- We will return to these data types
- ..but for now be a little careful with them
- ..Java and C treat them in rather different ways

- Each time that we define a class, however, we define a new data type
- Can be used as parameter, field and return types
- The internal is hidden from the user
  - No direct access to fields (unless special reason)
  - Access to state via accessor and mutator methods
- User does not need to know how the class is implemented to use/instantiate it
- The usage of a class is defined by its methods

Printing from methods

```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total += balance;
    // Clear the balance.
    balance = 0;
}
```

Output



```
##################
# The BlueJ Line
# Ticket
# 500 cents.
##################
```

Reflecting on the ticket machines

- Our first implementation of a TicketMachine class provides some useful functionality
- But the behaviour of the TicketMachine objects we create using that class is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need more sophisticated behaviour.

- We can use conditionals to make choices in our Java code (Just as we did in C)

```
public void insertMoney(int amount)
{
    if(amount > 0) {
        balance += amount;
    }
    else {
        System.out.println("Use a positive amount: " +
                               amount);
    }
}
```

Making choices

```
if(perform some test) {
    Do the statements here if the test gave a true result
}
else {
    Do the statements here if the test gave a false result
}
```

Coding Convention

- If statement
  - Always use { , even if there is only one statement
  - In case there is an else statement, start on a new line and use {
- Indentation
  - Always indent your code, even if your text editor does not do it automatically
- Document your code, the sooner the better.

Boolean Tests

- $==$: equality
- $>$: greater than
- $<$: less than
- $<=$: less or equal than
- $>=$: greater or equal than
- $! =$: not equal

Local variables

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
  - A bit like global variables in C
- Methods can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.
  - Like function variables in C

Local variables

```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

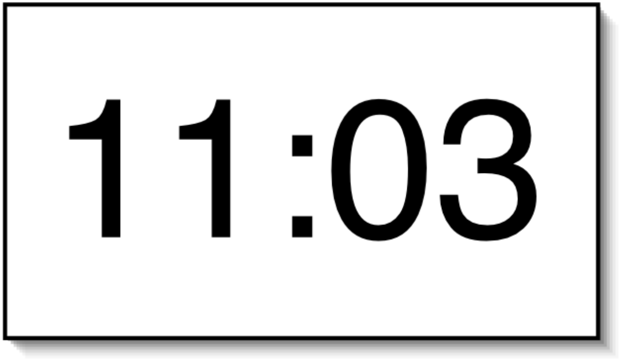
Review

- Class bodies contain fields, constructors and methods.
- Fields store values that determine an objects state.
- Constructors initialize objects.
- Methods implement the behaviour of objects.
- Mutators (mutator methods) change the state of a object
- Accessors (accessor methods) provide information about the state of an object

Review

- Fields, parameters and local variables are all variables.
- Fields persist for the lifetime of an object.
- Parameters are used to receive values into a constructor or method.
- Local variables are used for short-lived temporary storage.
- Objects can make decisions via conditional (if) statements.
- A true or false test allows one of two alternative courses of actions to be taken.

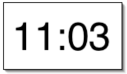
A digital clock



Abstraction and modularization

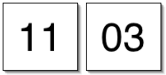
- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.
- Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Modularizing the clock display



One four-digit display?

Or two two-digit displays



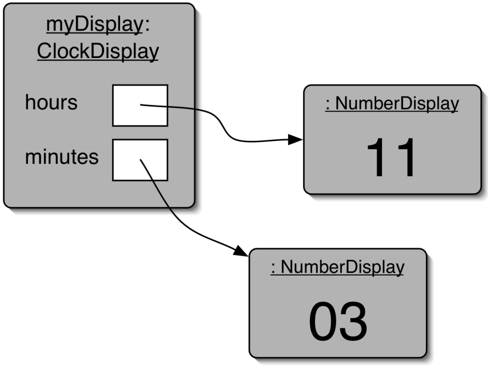
Implementation: NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

    Constructor and
    methods omitted.
}
```

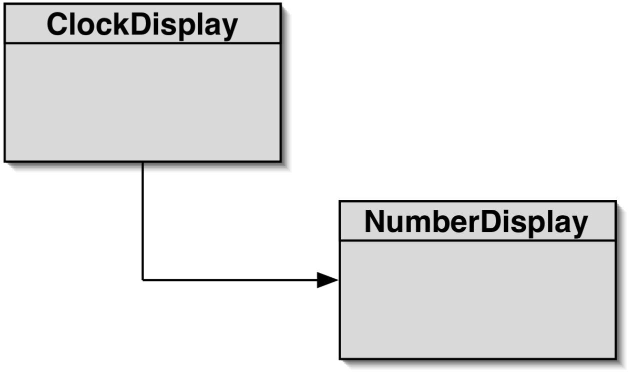
Implementation: ClockDisplay

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

    Constructor and
    methods omitted.
}
```

Object diagram



Class diagram



Diagrams

- Class Diagrams
  - Shows the classes of an application and the relationships between them
  - Gives information about the source code
  - Static view of the program
- Object Diagrams
  - Shows objects and their relationships at one moment in time during the execution of the program
  - Dynamic view of the program

Primitive types vs. object types

- Java defines two very different kinds of type: primitive types and object types.
- Primitive types are predefined by Java.
- Object types originate from classes.
- Variables and parameters store references to objects.
- The primitive types are non-object types.
- This is the reason why Java is not a completely object oriented languages

Call-by-reference and Call-by-value

- There are two ways of passing arguments to methods in many programming languages: call-by-value and call-by-reference.
- Call-by-value: A copy of the actual parameter is passed to the formal parameter of the called method. Any change made to the formal parameter will have no effect on the actual parameter.
- Call-by-reference: the caller gives the called method the ability to directly access to the callers data and to modify that data if the called method so chooses.
- Just like C Java uses call-by-value
- For objects, the value is a reference to memory (like in C)

Source code: NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

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

Source code: NumberDisplay

```
public int getValue()
    {
        return value;
    }

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

Logical Operators

- && : and, operands are tested, left to right, until conclusion can be reached
- || : or, operands are tested, left to right, until conclusion can be reached
- ! : not
- & : and, both operands are tested
- | : or, both operands are tested

Source code: NumberDisplay

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

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

String Concatenation

- Addition:
  - $12 + 24$
- String Concatenation:
  - “Java” $+$ “ and C” $->$ “Java and C”
  - “answer’: ” $+ 42 ->$ ”answer: 42”

String toString() method

- String toString() method: Java provides a way of transforming every Object into a String.
- To tailor this to your own preference write a method toString() returning a String representation of your class/object.

```
public String toString()
{
        return ``value: '' + value + `` with limit
}
```

The Modulo Operator

- % : the modulo operator calculates the remainder of an integer division
    - 27 % 4 $->$ 3
- Division in Java: if both arguments are integers, division will result in an integer.
    - double res = 5 / 2 $->$ res = 2
    - double res = 5 / (2.0) or 5 / (2 * 1.0) $->$ res = 2.5

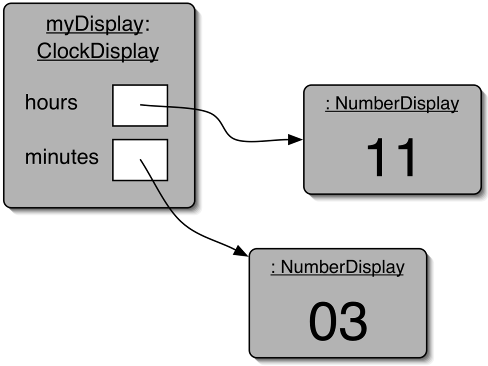
Objects creating objects

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

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

Objects creating objects

1. new ClassName(parameter-list)
    - It creates a new object of the named class
        - ★ here NumberDisplay
        - ★ this involves creating sufficient memory to store the values of primitive instance variables and references to object instance variables.
2. It executes the constructor of that class

ClockDisplay object diagram



Method Overloading

- Multiple Constructors of ClockDisplay:
  - **new** Clockdisplay()
  - **new** Clockdisplay(hour, minute)
- It is common for class definitions to contain alternative versions of constuctors or methods that provide various ways of achieving a particular task via their distinctive sets of parameters.
- This is known as overloading.

Method calling

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

Internal method

```
/**
 * Update the internal string that
 * represents the display.
 */
private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + ":" +
        minutes.getDisplayValue();
}
```

Method calls

- internal method calls

```
updateDisplay();
private void updateDisplay()
```

  - methodName(parameter–list)

- external method calls

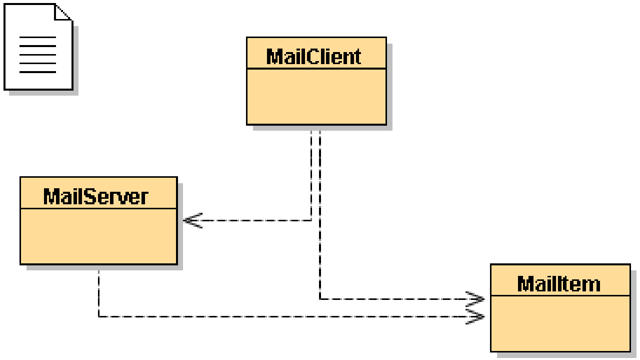
```
minutes.increment();
```

  - object.methodName(parameter-list)

Public and Private Methods

- Public methods:
  - **public void** increment()
  - can be called externally
- Private methods
  - **private void** updateDisplay()
  - can only be called internally
  - used for auxiliary methods

The Mail System



The this Keyword

```
public class MailItem
{
        private String from;
        private String to;
        private String message;

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

The this Keyword

- **this**.from = from
  - name overloading: the same name is used for two different entities: instance variable and formal parameter.
  - this is used to go out of the scope of the constructor to class level
  - **this** always refers to the current object.
  - can also used for methods
  - for internal methods calls and access to instance fields Java automatically inserts this: updateDisplay −> **this**.updateDisplay

Implementing a Test Program I

- The purpose on a test program is to verify that one or more methods have been implemented correctly
- A test program calls methods and checks that they return the expected results.
- It contains the following steps:
  1. Provide a tester class
  2. Supply a main method
  3. Inside the main method, create one or more objects
  4. Apply methods to the objects
  5. Display the results of the method calls - if needed
  6. Display the valued that you expect to get - if possible

Implementing a Test Program II

- Consider the Shapes project. It contains allows you to draw circles, squares and triangles on a canvas.
- To this extend it contains the classes: Circle, Squares, Triangle and Canvas
- To test if the implementation is correct we can write a test class

```
public class ShapesTest
{
    public static void main(String[] args)
    {
        Canvas c = Canvas.getCanvas();
        Circle c1 = new Circle();
        Square s1 = new Square();
        Triangle t1 = new Triangle();
        c1.makeVisible();
        s1.makeVisible();
        t1.makeVisible();
        ...
```

Implementing Applications

- the main method of your application class should be relatively short
- normally a few objects are created and a few methods are invoked.
- the invoked methods will determine the behaviour of your application.