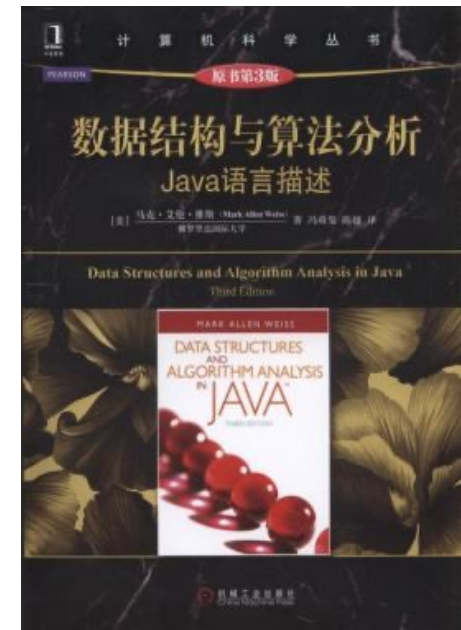
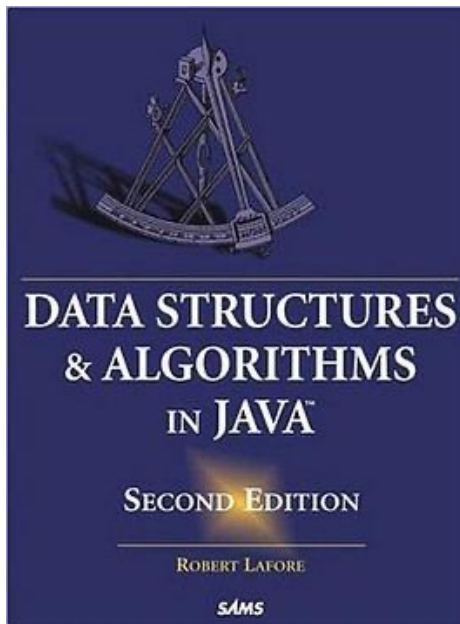
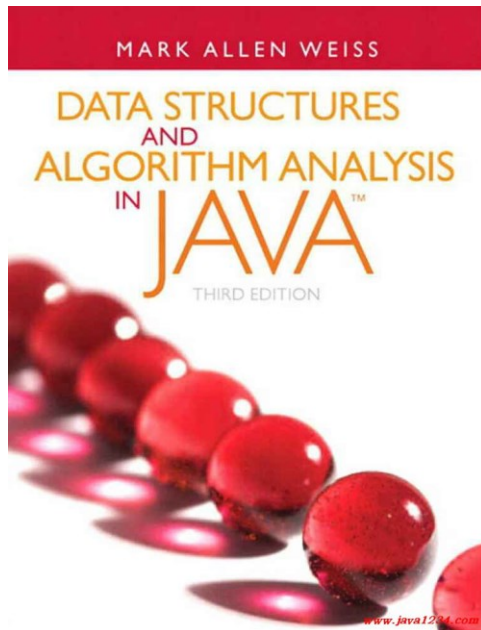


Topic 3 – Methods and Objects



Topics

- Introduction
- Programming Revision
- **Methods and Objects**
- Arrays and Array Algorithms
- Big O Notation
- Sorting Algorithms
- Stacks and Queues
- Linked Lists
- Recursion
- Bit Manipulation

Java program structure



Java programs are built up of multiple files called **classes**



There are advantages in splitting a program into distinct files rather than keeping it in one big chunk

The different components can be easily re-used

The same piece of code can be re-run as many times as you want without re-typing it



Classes are made up of **variables** and **methods**

Variables store information

Methods are contained chunks of code which do a specific job and then return a result

A big chunk of code

```
//Calculating the circumference & area of a circle
import java.util.Scanner;
public class MyProgram {
    public static void main(String args[]) {
        final double PI = 3.1414;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter radius");
        double radius = in.nextDouble();
        double circumference = radius*2*PI;
        System.out.println("Circumference="+circumference);
        double area = PI*radius*radius;
        System.out.println("Area="+area);
    }
}
```



Problems with the chunk

- The program does what it's supposed to, calculating the *circumference* and *area*
- However, the code is not very re-usable as it's hard to separate out the different bits and pieces
- When the programs gets big it becomes more and more important to break into **distinct components**
- A **method** is a piece of code that takes in some values, does a calculation using them and sends back a result

Method structure

- First is the access modifier
 - **public** if **any program can call** the method
 - **private** if the method can only be called from within that file
 - You also need to include the word **static** if the method is included in the same file as your main method
- Second is the **return type**
 - **void** if it returns nothing
 - **int** if it returns an int
 - **String** if it returns a String
- Third is the **name** of the method

Method structure

- The bit in brackets is the **parameter list**
 - This is the list of variables that the method takes in
 - Their types are given as well as the names they will have for the duration of the method
 - Variables can be re-christened with new names when they arrive in a method
 - Parameters are separated by commas
 - If there are no parameters put an empty brackets

```
public void square (int one, int two){...}  
private boolean isPrime (int number){...}  
public static void printList(){...}
```



Method structure

- **public void square (int one, int two){...}**
 - This method is **public**
 - It don't return any thing
 - Its name is square
 - Its **parameter list** is (int one, int two)

Method structure

- **public void square (int one, int two){...}**
 - This method is **public**
 - It don't return any thing
 - Its name is square
 - Its **parameter list** is (int one, int two)
- **private boolean isPrime (int number){...}**
 - This method is private
 - It returns a boolean value
 - Its name is isPrime
 - Its **parameter list** is (int number)

Method structure

- **public void square (int one, int two){...}**
 - This method is **public**
 - It don't return any thing
 - Its name is square
 - Its **parameter list** is (int one, int two)
- **private boolean isPrime (int number){...}**
 - This method is private
 - It returns a boolean value
 - Its name is isPrime
 - Its **parameter list** is (int number)
- **public static void printList(){...}**
 - This method is included in the same file as your **main method**

Method structure

- If your method is supposed to return an answer then a return statement must be included at the end
 - **return** answer;
 - **return** 0;
- The code to be executed by the method is wrapped up in curly brackets **{ }**
- The main method is a special method because it is the method that is always run first by the Java Virtual Machine
 - The main method always takes in an array of Strings as arguments by default
 - `public static void main(String[] args){...}`

A big chunk of code

//Calculating the area of a circle

import java.util.Scanner;

// It don't return any thing

public class MyProgram {

public static void main(String args[]) {

final double PI = 3.1414;

Scanner in = new Scanner(System.in);

System.out.print("Enter radius");

double radius = in.nextDouble();

double circumference = radius*2*PI;

System.out.println("Circumference="+circumference);

double area = PI*radius*radius;

System.out.println("Area="+area);

}

}

**// The code to be executed by the method
is wrapped up in curly brackets { }**

Calling a method

- You can call a method as **many times** as you want, sending in any variable(s) matching its parameters

```
int area1 = getArea(2);  
double area2 = getArea(3);  
double area3 = getArea(5.6);  
double area4 = getArea("hello"); // compile error!
```

- "hello" is a string, and it is not a double value

```
int area5 = getArea(2.5); // compile error!
```

- 2.5 is a float value, and it is not an integer value

- Inputs go in, result comes out

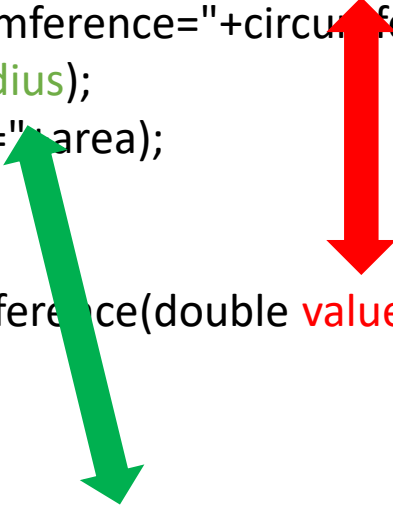
Example with methods

```
public class Test {  
    public static final double PI = 3.1414;  
  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        double circumference = getCircumference(radius);  
        System.out.println("Circumference="+circumference);  
        double area = getArea(radius);  
        System.out.println("Area="+area);  
    }  
  
    public static double getCircumference(double valuein) {  
        return PI*2*valuein;  
    }  
  
    public static double getArea(double valuein) {  
        return PI*valuein*valuein;  
    }  
  
}
```



Example with methods

```
public class Test {  
    public static final double PI = 3.1414;  
  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        double circumference = getCircumference(radius);  
        System.out.println("Circumference="+circumference);  
        double area = getArea(radius);  
        System.out.println("Area="+area);  
    }  
  
    public static double getCircumference(double valuein) {  
        return PI*2*valuein;  
    }  
  
    public static double getArea(double valuein) {  
        return PI*valuein*valuein;  
    }  
}
```



Advantages



One method called **getCircumference** takes in a radius and calculates the circumference



Another method called **getArea** takes in a radius and calculates the area



We've separated the code into distinct parts, making it **easy to identify** and **re-use** individual components

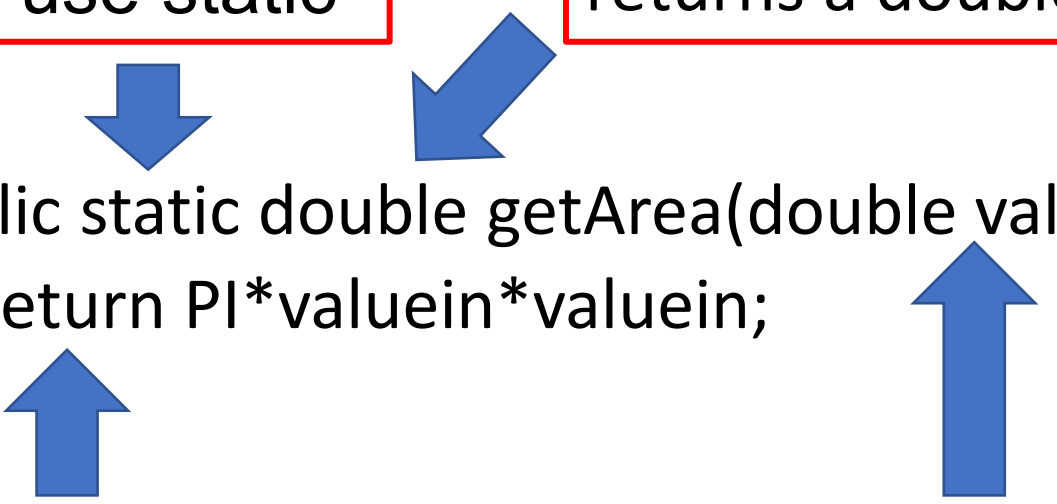


We can run a method as many times as we want with different numbers

Method features

It's in the main file so use static

This method returns a double



```
public static double getArea(double valuein) {  
    return PI*valuein*valuein;  
}
```

It returns the result of this calculation

Method takes in an double
That double will be referred to as 'valuein' for the duration of the method

Variable scope

- A variable defined inside a method will only exist for the **duration of that method**
- When the method returns a result, the variable will be disposed of by Java's garbage collector
- If you want a variable to be available to all methods you need to define it outside those methods
 - **PI** is declared **outside** the methods of the class
 - Therefore it is a **class variable** rather than a **method variable**
- The area of the program in which a variable is available is called the scope of that variable

Class variable scope

```
public class MyProgram {  
    public static final double PI = 3.1414;  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        double circumference = getCircumference(radius);  
        System.out.println("Circumference="+circumference);  
        double area = getArea(radius);  
        System.out.println("Area="+area);  
    }  
    public static double getCircumference(double valuein) {  
        return PI*2*valuein;  
    }  
    public static double getArea(double valuein) {  
        return PI*valuein*valuein;  
    }  
}
```

PI is declared **outside** the methods of the class
It is a **class variable** rather than a **method variable**

circumference is a **method variable**

Class variable scope

```
public class MyProgram {  
    public static final double PI = 3.1414;  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        double circumference = getCircumference(radius);  
        System.out.println("Circumference="+circumference);  
        double area = getArea(radius);  
        System.out.println("Area="+area);  
    }  
    public static double getCircumference(double valuein) {  
        return PI*2*valuein;  
    }  
    public static double getArea(double valuein) {  
        return PI*valuein*valuein;  
    }  
}
```

The diagram illustrates the scope of variables in the provided Java code. A red arrow originates from the `PI` variable declaration at the top and points to a red-bordered box at the bottom containing the text "This is the scope of PI". A blue arrow originates from the `getCircumference` method call in the `main` method and points to a blue-bordered box on the right containing the text "This is the scope of circumference".

This is the scope of
circumference

This is the scope of PI

Class variable scope

```
public class MyProgram {  
    public static final double PI = 3.1414;  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        double circumference = getCircumference(radius);  
        System.out.println("Circumference="+circumference);  
        double area = getArea(radius);  
        System.out.println("Area="+area);  
    }  
    public static double getCircumference(double valuein) {  
        return PI*2*valuein;  
    }  
    public static double getArea(double valuein) {  
        return PI*valuein*valuein;  
    }  
}
```

This is the scope of **valuein** which is the parameter of **getCircumference**

Classes and objects

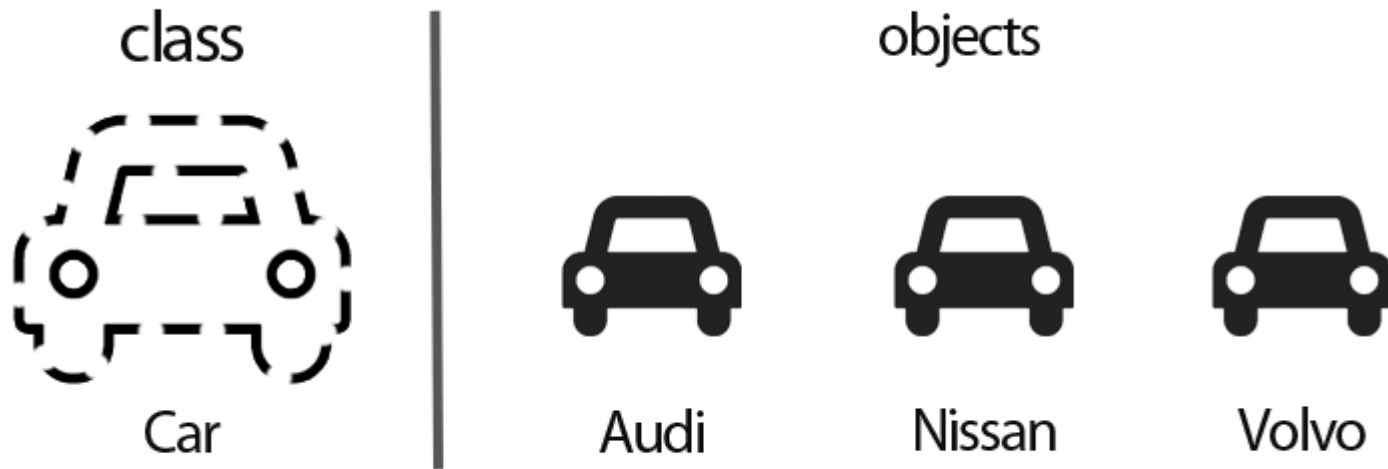
- Say we want to re-use a method like **getArea()** or **getCircumference()** in a new program
- Do we need to copy and paste these methods into our new program? That would be quite a waste of space
- The **object-oriented** approach allows us to use the methods defined in other files (which are called **classes**)
- First we create an instance of that class called an **object**, sending in the values required by the constructor of that class
- Then, we can call methods on that object and the results are returned in the same way as before

Classes

- A **class** is a collection of **variables** and **methods** that operate on those variables

Class	MyProgram
variables	<ul style="list-style-type: none">• radius• PI
methods	<ul style="list-style-type: none">• getCircumference()• getArea()

Objects



Objects

- Creating an object looks like this:

new is the reserved word for creating objects

Circle myCircle = new Circle(double radius);

This can have any name

This is the value sent into the **constructor** of the Circle class

- The program looks for the file called **Circle.java** and runs the constructor of that class

Objects

- This is how to call a method on an object:

double area = **myCircle**.getArea();



Result saved in
this variable

This variable is sent
into the method
Here, we sent nothing

The **dot** is the reserved
symbol which is used to
run a method on an object


Full example

```
public class MyProgram {  
    public static void main(String args[]) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Enter radius");  
        double radius = in.nextDouble();  
        Circle myCircle = new Circle(radius);  
        double circumference = myCircle.getCircumference();  
        System.out.println("Circumference="+circumference);  
        double area = myCircle.getArea();  
        System.out.println("Area="+area);  
    }  
}
```

**// Run constructor
in Circle.java**



**// Run method in
Circle.java and return
result**



Circle.java

```
public class Circle {  
    public final double PI = 3.1414;  
    public double radius;  
    public Circle(double valuein) {  
        radius = valuein;  
    }  
    public double getArea() {  
        return PI*radius*radius;  
    }  
    public double getCircumference() {  
        return PI*2*radius;  
    }  
}
```



Circle.java

// constructor
code is run

```
public class Circle {  
    public final double PI = 3.1414;  
    public double radius;  
    public Circle(double valuein) {  
        radius = valuein;  
    }  
    public double getArea() {  
        return PI*radius*radius;  
    }  
    public double getCircumference() {  
        return PI*2*radius;  
    }  
}
```

// a variable stored
by Circle objects

// method called

// answer returned

// Note see FULL EXAMPLE

// "double circumference = myCircle.getCircumference();" 33

Constructor

- When you create a new object from a class, the constructor in that class is **automatically run**

```
public Circle(double valuein) {  
    radius = valuein;  
}
```



// It puts its **variable** radius equal to the value sent in: this variable is accessible by all methods in the class

// This constructor must be sent an **double** which is referred to as 'valuein' for the duration of the constructor

Multiple objects

- You can **create as many objects** as you want, using different values sent into the constructor
- You can then call methods on particular objects

```
Circle myCircle1 = new Circle(5);  
Circle myCircle2 = new Circle(3.6);  
Circle myCircle3 = new Circle(2.9);
```

```
double area = myCircle2.getArea();
```

- `// area = 40.7`

```
double anotherArea = myCircle3.getArea();
```

- `// anotherArea = 26.4`

Note.

=====

```
public double getArea() {  
    return PI*radius*radius;  
}
```


Advantages of objects



The program is split into several components, each stored in a separate **class** file



Each class has its own **variables** and **methods**



You can make as many objects of a class as you want



You can make an object of a class from any other program, promoting code re-use



If anyone else writes a program that involves circles, they will be able to use your Circle class so long as they have a copy of **Circle.java**

Advantages of objects

- Related pieces of information can be tied together in a single data structure
 - Student number
 - Student name
 - Date of Birth



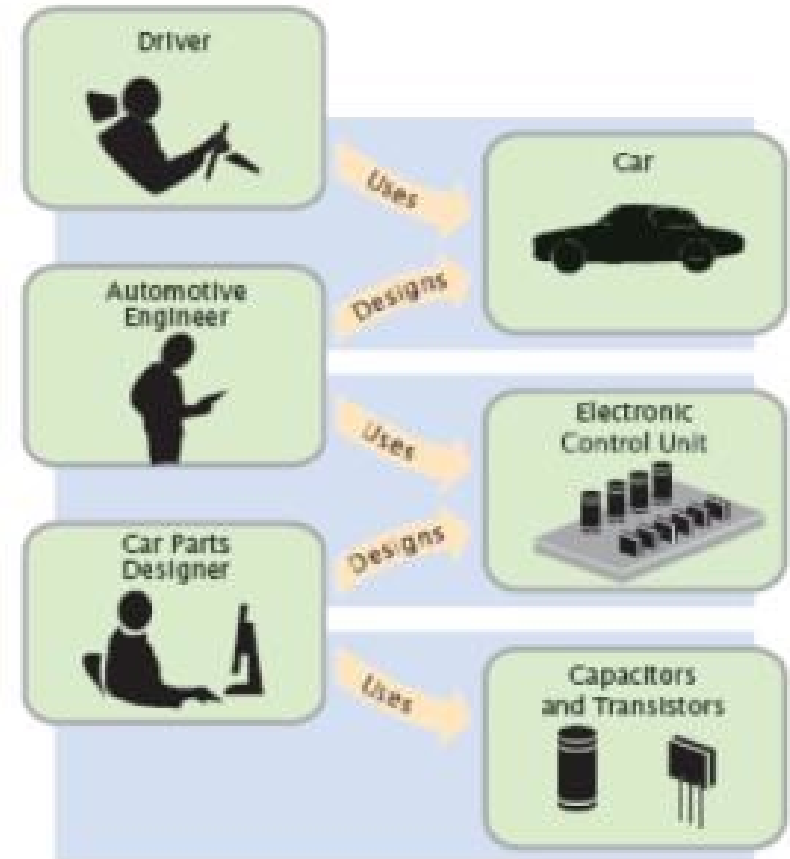
- You can write lots of methods that go with your class
- People calling these methods just need to know the input parameters and the return type
 - **They don't need to know how your methods work**
 - This is called **encapsulation**

Encapsulation

- Encapsulation hides all the unimportant details
- In *object-oriented programming* objects are like a black box because you don't need to know how they're implemented
 - You know what information to use to construct an object
 - You know what methods you can call on the object
 - You don't need to know anything else
- This makes it easy to use other people's code
- It also makes it easy to recycle individual components of a program

Encapsulation example

- Black box systems in a car:
 - The **driver** doesn't need to know how the car works to use it
 - The **mechanic** doesn't need to know how the electronic components of the car work to put them together
 - The **electronic engineer** doesn't need to know how transistors work to put together an electronic control unit



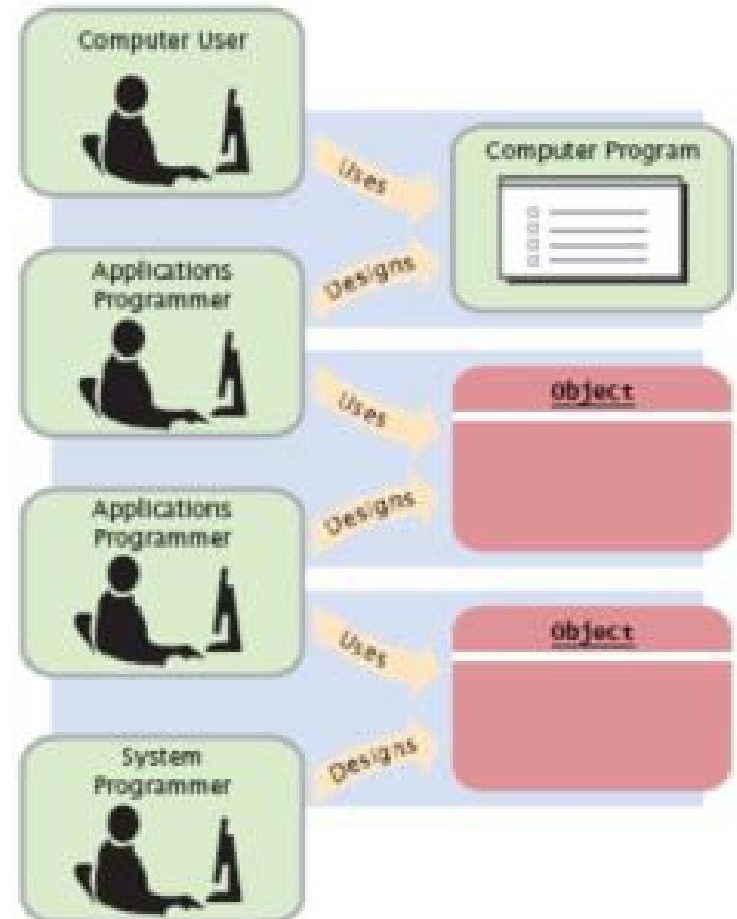
Encapsulation example

- **Encapsulation** leads to efficiency
 - The mechanic deals only with car engine components, not with capacitors and transistors
 - The driver worries only about interaction with car (e.g. putting petrol in the tank), not about how the engine works
 - All they need to know is how to use the system, not how the lower levels actually work



Software encapsulation

- Software engineers adhere to the same principles:
 - The **end user** uses a web application, without needing to know how it works
 - The **programmer** uses an application to help create the web application, without needing to know how it works
 - The **application programmer** uses an operating system to help create applications, without needing to know how it works



Java Application Programming Interface (API)

- The Java API describes all of the methods that go with a class
 - Names of the methods
 - Parameters they take in
 - A brief description of what they do
- The full Java API is available at
<https://docs.oracle.com/javase/7/docs/api/index.html>
- This contains a full description of all the classes and methods that you have already been using
 - String, Math...



String API

Modifier and Type	Method and Description
char	<code>charAt</code> (int index)Returns the char value at the specified index.
int	<code>compareTo</code> (<code>String</code> anotherString)Compares two strings lexicographically.
int	<code>compareToIgnoreCase</code> (<code>String</code> str)Compares two strings lexicographically, ignoring case differences.
<code>String</code>	<code>concat</code> (<code>String</code> str)Concatenates the specified string to the end of this string.
boolean	<code>endsWith</code> (<code>String</code> suffix)Tests if this string ends with the specified suffix.
boolean	<code>equals</code> (<code>Object</code> anObject)Compares this string to the specified object.
boolean	<code>equalsIgnoreCase</code> (<code>String</code> anotherString)Compares this String to another String, ignoring case considerations.
static <code>String</code>	<code>format</code> (<code>Locale</code> l, <code>String</code> format, <code>Object</code> ... args)Returns a formatted string using the specified locale, format string, and arguments.
static <code>String</code>	<code>format</code> (<code>String</code> format, <code>Object</code> ... args)Returns a formatted string using the specified format string and arguments.

Java features

- A java program will consist of a number of classes each written in a separate file with the **same name** as the class
- Each class will have methods and variables declared in it as well as a **constructor** with the **same name** as the class
- One class will be the one which starts the program - this will have a **main()** method which is run automatically
- Curly brackets are used to separate methods within classes **{ }**

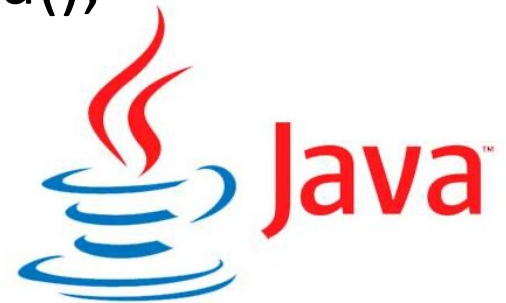


Java features

- Every time you see the word **new**, an object of a class is being created
- Every time you see the dot **.** a method in a class is being called on an object of that class

```
Circle myCircle3 = new Circle(2.9);  
double anotherArea = myCircle3.getArea();
```

- Naming convention:
 - classes start with an uppercase letter
 - objects start with a lowercase letter



```
Typewriter myTypewriter = new Typewriter( );
```

Java program structure

// comments about the class

public class Universe {



// class header

// class body

// Comments can be placed almost anywhere

}

// This class is written in a file named: Universe.java

Running a program

- To run a java program you must run the **main()** method
 - It is **public** which means that it can be run from anywhere
 - It is **static** which means you don't need to have created an object to use it
 - It is **void** meaning it doesn't return a result
 - It can take in an array of Strings as an input parameter (**String[] args**)

```
public static void main(String[] args){
```

Java program structure

// comments about the class

public class Universe {

// comments about the method



public static void main(String[] args) {

} // method body

}

Java program structure

// comments about the class

public class Universe {

// This method doesn't
return anything

// Anyone
can run this
method

// comments about the method

public static void main(String[] args) {

// method
body

// Name of method

// Parameters passed
to this method, in
this case an array of
strings

// This method can be run
without creating an object

Java **class** modifiers

- A class can be:
 - **abstract** is a class with abstract methods which are empty (there is no code associated with the methods)
 - it cannot be instantiated
 - **final** describes a class that can have no subclasses
 - **public** describes a class that can be instantiated or extended by any other package
 - No modifier, then the class is **friendly** (can be instantiated by classes in the same package only)

Java **method** modifiers

- A method can be:
 - **public** is a method that can be called by any class anywhere
 - **protected** is a method that can only be called from inside the class or any of its subclasses in the same package
 - **private** is a method that can only be called from inside the class

Static methods

- Ordinary methods are known as **instance methods** because they operate on a particular instance of an object
- To use these methods, you need to create an object of the class first and then call the method on that object
- A **static method** can be run on its own without creating an object

```
public static void main(String[] args){
```

Variables

- A **local variable** is created inside a method or loop and cannot be accessed outside that method or loop

```
for(int i = 0; i < 10; i++){
```

- A **parameter variable** is one that arrives into a method

```
public Contact(int number, String name){
```

- A **class variable** is a variable in a class for which a single copy is shared by all objects of that class

```
public static final PI = 3.14
```

Variables

- An **instance variable** is one that is defined inside a class
- Every time an object of that class is created, it gets its own unique instance of that variable
- These variables are usually declared as **private** so they can only be manipulated within the class in which they are created

```
public class Card {  
    . . .  
    private String suit;  
}
```

Instance variables

- An instance variable declaration has the following parts:
 - **access specifier**
 - private or public
 - **type of variable**
 - int, double, String etc.
 - **name** of variable

```
public class Card {  
    . . .  
    private String suit;  
}
```

Example

```
public class Card {  
    private String suit;  
    private int value;  
    public Card (String suit_in, int value_in) {  
        suit = suit_in;  
        value = value_in;  
    }  
    public String checksuit ( ) {  
        return suit;  
    }  
}
```



Example

// Can be instantiated by any other class

public class Card {

private String suit;

private int value;

// These instance variables are private
so can only be accessed within the class

public Card (String suit_in, int value_in) {

suit = suit_in;

value = value_in;

// The constructor sets the
instance variables equal to the
parameter variables

}

public String checksuit () {

return suit;

}

// A method which returns a string

}

// Can be accessed from any other class

Accessing instance variables

- If the instance variables are **private** then this means they cannot be directly accessed from other classes
- In this case, methods need to be provided for manipulating and accessing these variables
- The checksuit() method of the Card class can access the private instance variable suit

```
public String checksuit( ) {  
    return suit;  
}
```

Accessing instance variables

- Other classes cannot access or manipulate these private variables directly
- Encapsulation involves hiding data and providing access through methods instead

```
public class AnotherClass {  
    public static void main(String[] args) {  
        Card myCard = new Card("Spades", 6);  
        ...  
        myCard.suit = "Hearts"; // ERROR!!!  
    }  
}
```