JAVA PROGRAM STRUCTURE

- Java programs are built up of multiple files called….?
  - classes
- What are the advantages of splitting a program into distinct files?
  - It allows us to reuse components easily, so the same piece of code can be run without retyping it
- Classes are made up of variables and methods
  - Variables store information
  - Methods are chunks of code that do a specific job, and return some result

A BIG CHUNK OF CODE

```
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

As our code gets longer and more complex it becomes more important to break it into **distinct components**

A method is a piece of code that takes in some values (or none) and sends back a result

METHOD STRUCTURE

```
public static void printSomething()
```

- 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

```
public static void printSomething()
```

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

- 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 inside 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){...}
```

CALLING A METHOD

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

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

- Inputs go in, result comes out

EXAMPLE WITH METHODS

```
public class MyProgram{
    public 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;
     }
}
```

ADVANTAGE?

- 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 (inputs)

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 a *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, we need to define it outside of those methods
  - In the previous examples, PI was declared outside the methods if the class. We spoke about this last week – 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 the variable

CLASS VARIABLE SCOPE

```
public class MyProgram{

    public 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 double getCircumference(double valuein){
         return PI*2*valuein;
      }

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

Where does the scope of PI start and end?

Where does the scope of valuein start and end?

CLASSES AND OBJECTS

Let’s 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?

CLASSES AND OBJECTS

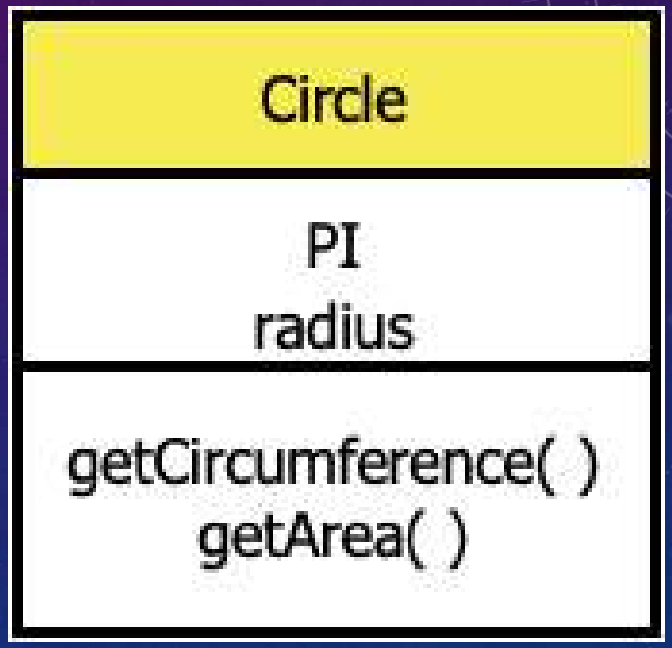
Instead we want to use an **object oriented** approach, which allows us to use the methods defined in other files (which are called **classes**)

First we create an instance of our class, called an object, and send in the values required by the constructor of that class

Now 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



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

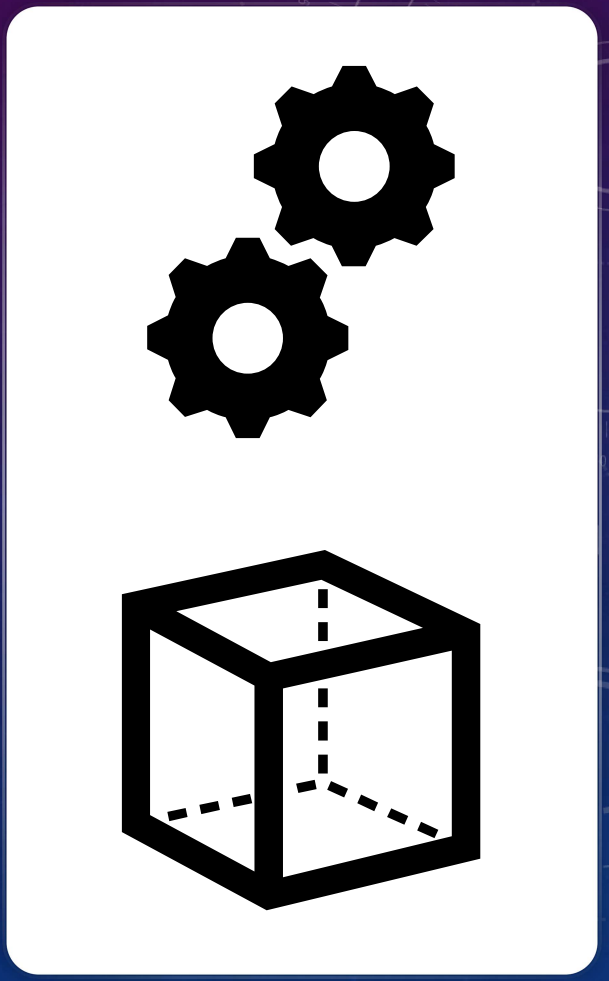
This is the value sent into the **constructor** of the Circle class

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

OBJECTS

How do we call a method on an object?

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



OBJECTS

This variable is sent into the method

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

Result saved in this variable

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

FULL EXAMPLE

Run constructor in Circle.java

```
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 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;
    }
}
```

a variable stored by Circle objects

constructor code is run

method called

answer returned

CONSTRUCTOR

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

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

```
Circle myCircle = new Circle(radius);
```

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

It puts its variable **radius** equal to the value sent in

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();

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

area = 40.7

anotherArea = 26.4

ADVANTAGES OF OBJECTS?

- Our program is now split into a number of 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**

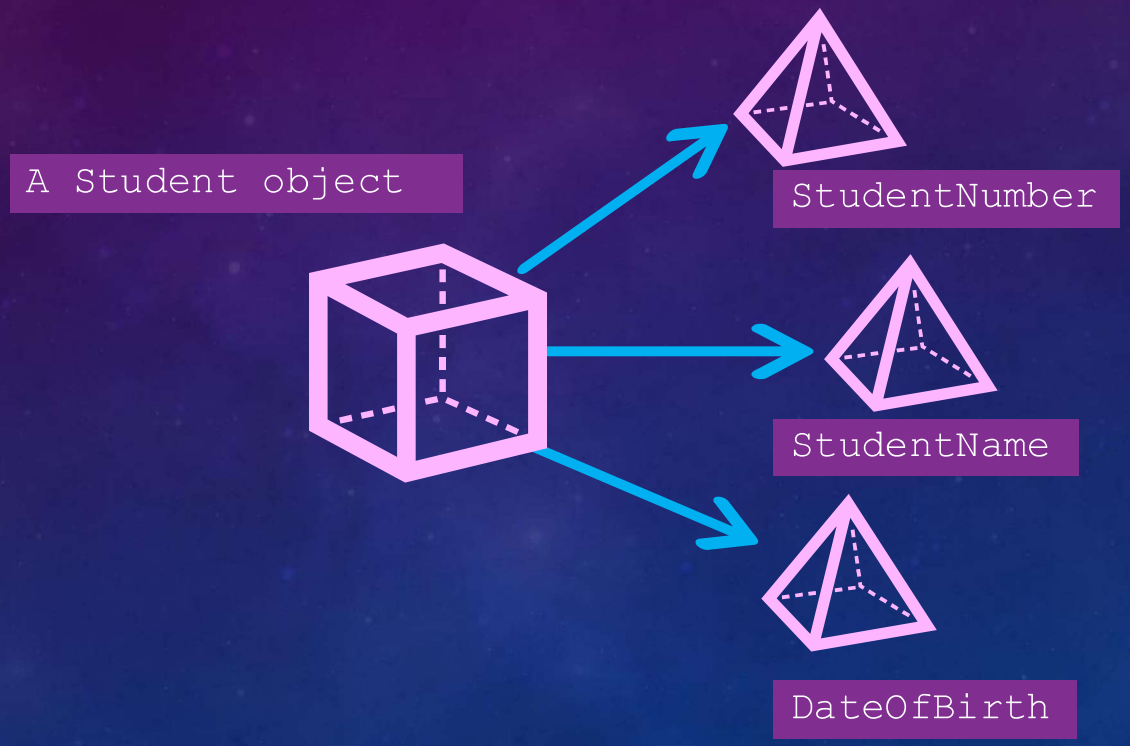
ADVANTAGES OF OBJECTS

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

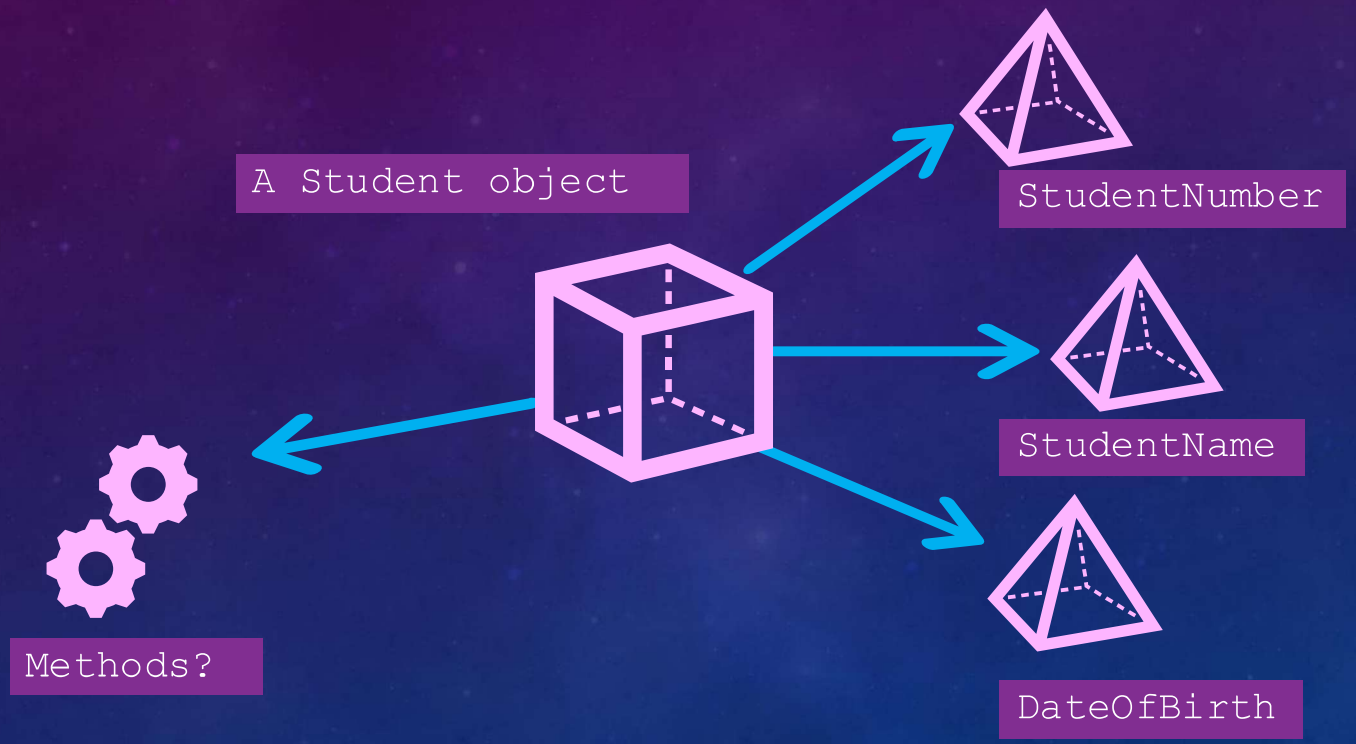
That's the name of the module!

- 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**

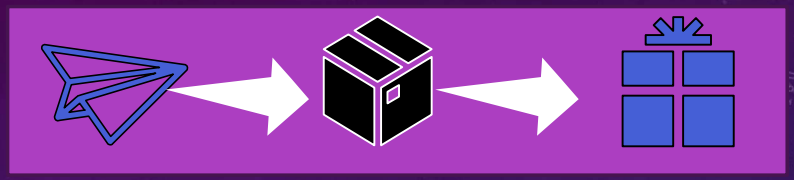
STUDENT.JAVA



STUDENT.JAVA



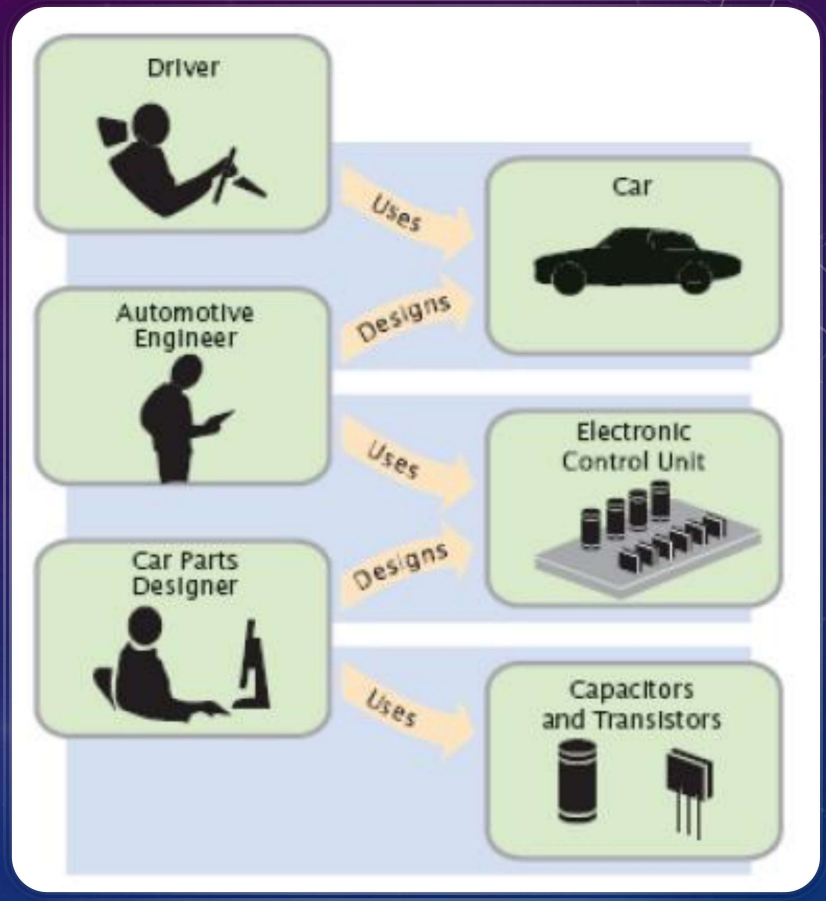
ENCAPSULATION



- 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

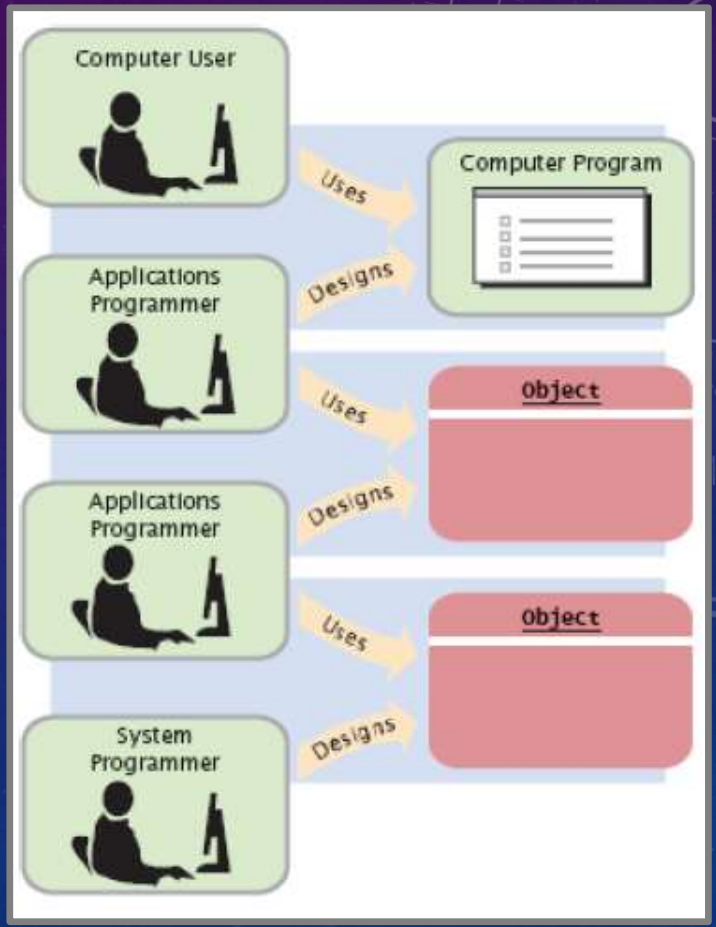


ENCAPSULATION EXAMPLE

- 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 **http://java.sun.com/j2se/1.4.2/docs/api/**

- This contains a full description of all the classes and methods that you have already been using
  - String, Scanner, Math…

STRING API

Method Summary

char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case considerations.
String	**concat**(String str) Concatenates the specified string to the end of this string.
static String	**copyValueOf**(char[] data) Returns a String that is equivalent to the specified character array.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that is equivalent to the specified character array.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.

JAVA FEATURES

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

- A java program will consist of a number of classes each written in a separate file with the **same name** as the class

```
Circle class -> Circle.java
```

- 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 Cirle(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

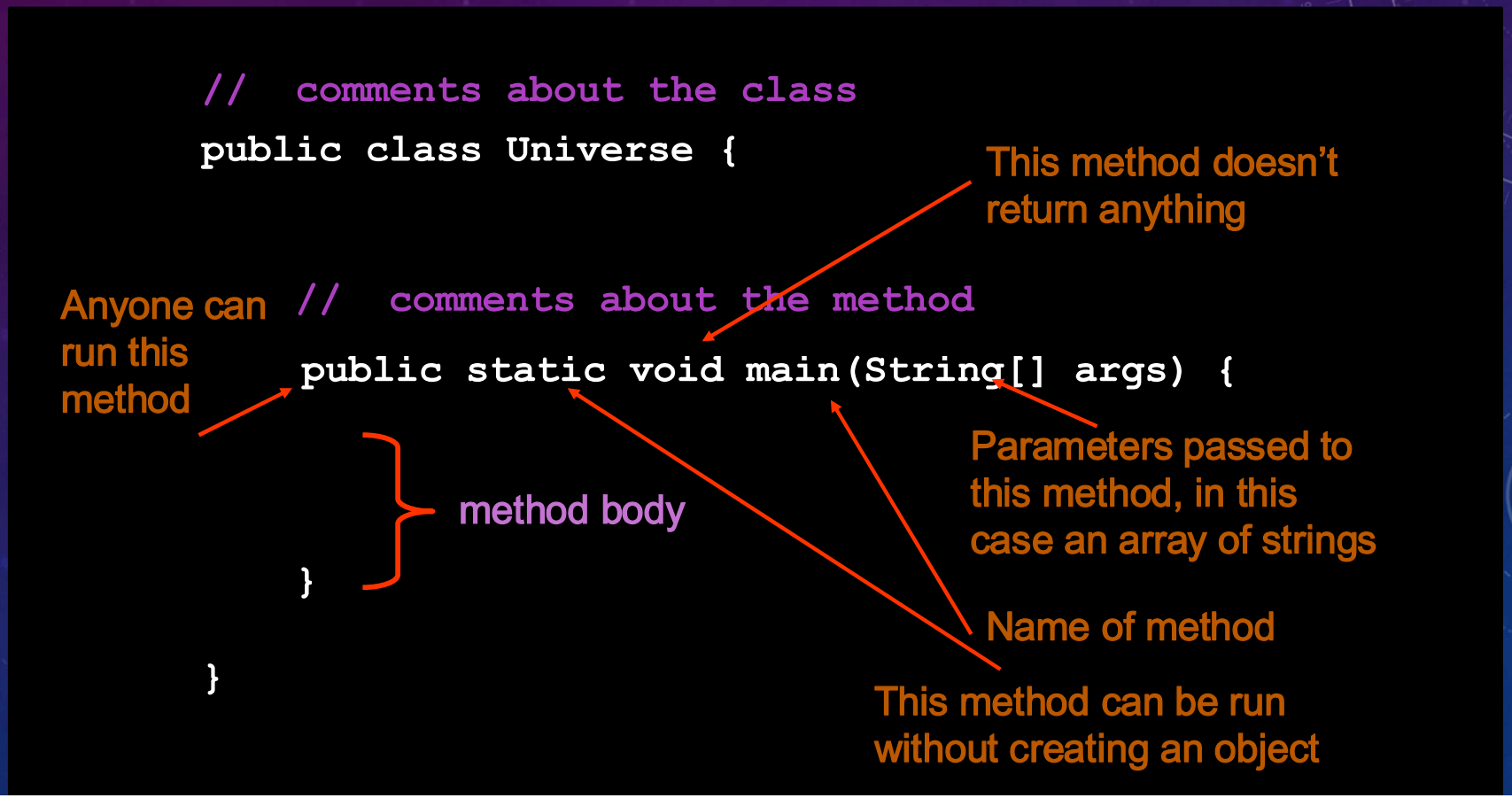
This class is written in a file named: Universe.java

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



JAVA CLASS MODIFIERS

- A class can be:
  - **`abstract`** - 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 consists of the following parts:
  - access specifier (`private, public`)
  - type of variable (`int, double, String etc.`)
  - name of variable

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

EXAMPLE

Can be instantiated by any other class

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

```
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;
    }
}
```

The constructor sets the instance variables equal to the parameter variables

Can be accessed from any other class

A method which returns an `int`

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

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

- Encapsulation involves hiding data and providing access through methods instead

LAB 1 - LUHN'S ALGORITHM

Problem Statement

The task is to take in a credit card number and find out if it is a valid credit card number or not. Credit card numbers follow an algorithm called "Luhn's algorithm".

The formula verifies a number against its check digit, which is the last digit. This number must pass the following test:

1. From the rightmost digit, which is the check digit, and moving left, double the value of every second digit. If the result of this doubling operation is greater than 9 (e.g., 8 × 2 = 16), then add the digits of the product (e.g., 16: 1 + 6 = 7, 18: 1 + 8 = 9) or, alternatively, the same result can be found by subtracting 9 from the product (e.g., 16: 16 - 9 = 7, 18: 18 - 9 = 9).
2. Take the sum of all the digits.
3. If the total modulo 10 is equal to 0 (if the total ends in zero) then the number is valid according to the Luhn formula; else it is not valid.

Assume an example of an account number "7992739871" that will have a check digit added, making it of the form 7992739871x. The sum of all the digits, processed as per steps 1 and 2, is **67+x**. Thus, x must be 3 to bring the total to be modulo 10 = 0. If x is not 3, then this is not a valid credit card number.

Input Format

- An n-digit credit card number, where the last digit is the check digit.

Output Format

- Output "VALID" if it is a valid credit card number and "INVALID" if it is not.

Constraints

- 4<=n<=30

Sample Input

- 4539682995824395

Sample Output

- VALID

Explanation

Make sure to read from right to left.

Double every second digit from the second last digit and subtract 9 if needed.

4539682995824395

2 * 9 = 18; 18 - 9 = 9

2 * 4 = 8

2 * 8 = 16; 16 - 9 = 7

2 * 9 = 18; 18 - 9 = 9

2 * 2 = 4

2 * 6 = 12; 12 - 9 = 3

2 * 3 = 6

2 * 4 = 8

9 + 8 + 7 + 9 + 4 + 3 + 6 + 8 = 54

Adding these together

46 + 54 = 100

100 % 10 == 0 So the card is VALID