

Object-oriented programming (OOP)

- Objects (or more precisely the classes objects come from), are reusable software components.
- There are time objects, video objects, etc....
- Almost any noun can be reasonably represented as a software object in terms of:
 - attributes (e.g. name, colour, size, etc...)
 - behaviours (e.g. calculating, moving, communicating, etc..)
- Software developers (particularly groups) can be more productive and programs are often easier to understand, correct and modify.

Car Example

- *Your goal:* drive a car and make it go faster by pressing the accelerator pedal.
- *To achieve this:* Someone must **design** it (typically starts with engineering drawings or **blueprints**).
- The drawing will include the design for the accelerator pedal.
- The actual pedal, however, **hides** the complex mechanisms that actually make the car go faster from the driver.
- This enables anyone with little or no knowledge of how engines work to easily drive a car.

Car Example

- However, you cannot drive a cars blueprints!!
- The car must be *built*, using the blueprints that define it, before it can be driven.
- The completed car has an *actual* accelerator pedal, which makes the car go faster.
- But even this is not enough – it won't accelerate on its own.
- It requires a driver to *press* the pedal to get the car to accelerate.

Lets use the car example to introduce some OOP concepts

Car Example

Methods and Classes

- Performing a task in a program requires a **method**.
- The method houses the program statements that actually perform its tasks.
- The method *hides* these statements from its user, just as the car hides from the driver the mechanisms that it employs to make the car go faster.
- In Java, we create a program unit called a **class** to house the set of methods that perform the class's tasks.
- A class is similar in concept to a car's blueprint, which contain the design of the accelerator pedal, steering wheel, etc...

Car Example

Some other class/methods examples

- A *class* that represents a **bank account**, might have *methods* to *withdraw* and *deposit* money, also to check the *balance*.
- A *class* that represents a **door**, might have *methods* to *lock*, *unlock*, *open* and *close* it.
- A *class* that represents a **phone directory**, might have *methods* to *insert*, *delete*, *edit*, *retrieve* and *search* entries within it.

Car Example

Instantiation

- We stated earlier that we need to **build** the car from its blueprints, before it could actually be driven.
- Before a program is able to perform the tasks that the class's methods define, we must **build** an object to represent that class.
- This process is known as **instantiation**
- The object is then referred to as an **instance** of its class.

Car Example

Reuse

- The blueprints for the car can be **reused** many times to build many cars.
- Similarly, you can **reuse** a class many times to build many objects.
- Reuse of existing classes when building new classes, saves time and effort.
- It also helps to build more reliable and effective programs, as existing classes will have gone through extensive *testing, debugging* and *performance tuning*.
- Reusable classes are a crucial part of software development.

Car Example

Messages and Method Calls

- When you drive a car, pressing the accelerator pedal sends it a **message** to perform a task – accelerate.
- In a similar fashion, you can **send messages to objects**.
- Each message is implemented as a **method call** that tells a method of the object to perform its task.

For example: a program might call a particular bank account's *deposit* method to increase that account's balance.

Car Example

Attributes and Instance Variables

- In addition to having the ability to accomplish tasks, a car also has **attributes** (e.g. colour, number of doors, current speed, etc....)
- Like its capabilities, the car's attributes are represented as part of its design in its blueprint.
- As you drive an actual car, these attributes are carried along with the car.
- Every car maintains its **own** attributes.
- E.g. each car knows how much fuel is in its tank, but **not** how much is in the tanks of **other** cars.

Car Example

Attributes and Instance Variables

- Similarly, an object has attributes that it carries along as its used in a program.
- These attributes are defined as part of the object's class.
- Attributes are defined by the class's **instance variables**.
- E.g. a bank account object has a **balance attribute** to represent the amount of money in the account.
- Each bank account object knows the balance in the account it represents, but **not** the balances of **other** accounts in the bank.

Car Example

Encapsulation

- Classes **encapsulate** (i.e. wrap) attributes and methods into objects
 - An object's attributes and methods are intimately related.
- Objects may communicate with one another, but they're normally not allowed to know how other objects have been implemented.
 - Implementation details are **hidden** within the objects themselves.
- This **information hiding**, as we'll see later one, is crucial to good software engineering.

Car Example

Inheritance

- A new class of objects can be created quickly and conveniently by **inheritance**.
 - The new class **absorbs** the characteristics of the existing class, possibly customising them or adding unique characteristics of its own.
- Consider our car, an object of class ‘convertible’ **is certainly an** object of the more **general** class ‘automobile’, but more **specifically**, the roof can be lowered and raised.

We will come back to look at OOP later

Objects

Constructing objects

- As outlined in the car analogy, objects must be **declared** before they can be used.
- Objects are initialised using the **new** operator.
- The new operator calls the **appropriate constructor** for the class. (**NOTE:** We'll come back to cover constructors in a little while).

Constructing objects

```
Date d = new Date();
System.out.println( "The current date: " );
System.out.println( d );

d = new Date( -100000 );
System.out.println( "Another date: " + d );

System.out.println( "The current date: " +
new Date() );
```

Accessing fields and methods

- Fields and methods are accessed via the **dot operator**

```
Point p = new Point( 10, 20 );  
System.out.println(p.y);  
p.x = 20;  
p.translate(20,10);
```

NOTE: For this code to work you would need to import java.awt.Point;

- In the last line, p is called the **implicit parameter** of the method call
- 20, 10 are the **explicit parameters** of the method call

References

- **Object variables** do not store the actual value or instance directly in their memory location (unlike **primitive type variables** such as **int**).
- Instead they store a reference to the memory location of the object.
- There can be **more than one reference** to a single object.

Multiple References

EXAMPLE: ReferenceVariableExample.java

Multiple References:

```
Point a = new Point( 10, 20 );  
  
Point b = a;  
  
System.out.println( "a:" + a + " b:" + b );  
  
System.out.println( "Excecuting a.translate(" +  
10, 10 );" );  
  
a.translate( 10, 10 );  
  
System.out.println( "a:" + a + " b:" + b );  
  
System.out.println( "Excecuting b.translate(" +  
10, 10 );" );  
  
b.translate( 10, 10 );  
  
System.out.println( "a:" + a + " b:" + b );
```

Wrapper classes

Wrapper classes

- **Wrapper classes** allow objects to be created from primitive types e.g:

```
Integer i = new Integer(57);
```

- These classes also contain some useful methods for working with primitive type variables
- Placing a primitive type within an object is referred to as **boxing**
- Java 1.5 added a feature called **autoboxing** where primitive type variables are **automatically** converted to an object of the equivalent type-wrapper class.

Autoboxing

EXAMPLE: AutoboxingExample.java

Autoboxing:

```
int x = 3;  
Integer y = x + 5;  
System.out.println("y = " + y);  
  
System.out.println("y == 8 : " + (y == 8));  
  
Integer z = new Integer(8);  
System.out.println("y == z : " + (y == z));  
  
// Use the intValue() method of class Integer to obtain  
the int value in the Integer object  
  
int v = z.intValue();  
System.out.println("y == v : " + (y == v));
```

Comparing objects

EXAMPLE: EqualsTest.java

Comparing objects

```
Integer a = new Integer( 57 );
Integer b = new Integer( 57 );
Integer c = a;

System.out.println( (a == b) );
System.out.println( a.equals(b) );
System.out.println( (a == c) );
System.out.println( a.equals(c) );
System.out.println( (b == c) );

System.out.println( b.equals(c) );
```

NOTE:

- `==` on primitive types tests for equivalence of *values*
- `==` on “object variables” tests for equivalence of *references*
- `.equals` on “object variables” tests for equivalence of *values*

Adding Two Numbers

EXAMPLE: EasyAdd.java

```
// EasyAdd.java
// Adds two specified integers.
import java.util.Scanner;

public class EasyAdd {
    public static void main( String args[] ) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int x = in.nextInt();
        System.out.print("Enter second number: ");
        int y = in.nextInt();
        int sum = x + y;
        System.out.println( "Their sum is " + sum );
    }
}
```

Application to add together 2 integers, requested from the user by using the Scanner class. **TASK:** Lets now rewrite our application to create a method that will get integer values using the Scanner Class.

Methods

Definition

A set of statements to solve some subproblem should be coded as a ***method***.

```
// Example method to count sheep
public static int countSheep( int n ) {
    int sheepCount = n * n;
    return sheepCount;
}
```

Static methods

Example – Method No Return [**void**]

```
public static void outputInt( String s, int n ) {  
    System.out.println(s + " " + n);  
}
```

Example – Method With Return [**int**]

```
public static int squareNumber( int n ) {  
    int result = n * n;  
    return result;  
}
```

```
public class Foo {  
    public static void main(...) {  
        bar();  
    }  
    public static void bar() { ... }  
}
```

```
public class Bob {  
    public static void main(...) {  
        Foo.bar();  
    }  
}
```

Static methods

- At first, the methods we write will be static.
- Static methods are methods that do not operate on objects.
- For example, the `pow` method of the `Math` class is a static method. The expression:

```
Math.pow(x, a);
```

- computes the power x^a , but you do not need to instantiate a `Math` object to use the method. i.e.

```
Math myMath = new Math();  
myMath.pow(x, a);
```

]

NOT NEEDED

static method – general form:

```
public static returnType methodName( parameter_list ) {  
    // Method body  
}
```

Parameter list (comma separated)

- Each parameter consists of a **type** and **identifier**.

Definition

- ***explicit parameters***: parameters in the method declaration.
- ***arguments***: parameters supplied when the method is called.

Parameters and scope

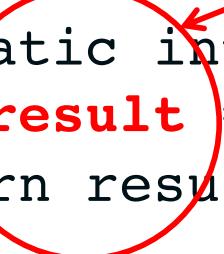
EXAMPLE: MethodScope.java

Remember -> Each parameter is only in scope in the corresponding method body.

```
public static void main( String[ ] args ) {  
    squareNumber(5);  
    System.out.println(result);  
}
```

OUT OF
SCOPE

```
public static int squareNumber( int n ) {  
    int result = n * n;  
    return result;  
}
```



Adding Two Numbers

EXAMPLE: SimpleAddWithMethod.java

```
// SimpleAddWithMethod.java
// Adds two specified integers using a getInt()
method.

import java.util.Scanner;

public class SimpleAddWithMethod {
    public static void main( String args[] ) {
        int x = getInt("Enter first integer");      METHOD
        int y = getInt("Enter second integer");      CALLS
        int sum = x + y;
        System.out.println( "Their sum is " + sum );
    }

    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
}
```

NOTE: Same functionality, but this time using a **method**

Passing arguments

EXAMPLE: PrimitiveArguments.java

Passing arguments (primitive types)

Primitive type variables are passed using *call by value* – a copy of the variable is passed to the method.

```
public static void main( String[] args ) {
    int i = 0;
    System.out.println( "Before someMethod i = " + i );
    someMethod( i );
    System.out.println( "After someMethod i = " + i );
}

static void someMethod( int a ) {
    System.out.println( " In someMethod a = " + a );
    i++;
    System.out.println( " In someMethod a = " + a );
}
```

Passing arguments

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

- **Objects** are **never passed as arguments** to **methods**
- References to objects are passed using *call by reference value*

```
public static void main( String[ ] args ) {  
    java.awt.Point p = new java.awt.Point( 20, 20 );  
    System.out.println( "Before someMethod p = " + p );  
    someMethod( p );  
    System.out.println( "After someMethod p = " + p );  
    someOtherMethod( p );  
    System.out.println( "After someOtherMethod p = " + p );  
}  
  
static void someMethod( java.awt.Point q ) {  
    System.out.println( " On entering someMethod q = " + q );  
    q.translate( 10, 10 );  
    System.out.println( " On leaving someMethod q = " + q );  
}
```

Passing arguments

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

```
public static void main( String[ ] args ) {
    java.awt.Point p = new java.awt.Point( 20, 20 );
    System.out.println( "Before someMethod p = " + p );
    someMethod( p );
    System.out.println( "After someMethod p = " + p );
    someOtherMethod( p );
    System.out.println( "After someOtherMethod p = " + p );
}
```

```
static void someOtherMethod( java.awt.Point q ) {
    System.out.println( " On entering someOtherMethod q = " + q );
    q = new java.awt.Point( 50, 50 );
    System.out.println( " In someOtherMethod q = " + q );
    q.translate( 10, 10 );
    System.out.println( " On leaving someOtherMethod q = " + q );
}
```

Group related methods

Lets go back to our adding application

```
public static void main( String args[] ) {  
    int x = getInt("Enter first integer");  
    int y = getInt("Enter second integer");  
    int sum = x + y;  
    System.out.println( "Their sum is " + sum);  
}  
  
public static int getInt( String prompt ) {  
    Scanner in = new Scanner(System.in);  
    System.out.print( prompt + " : " );  
    int result = in.nextInt();  
    return result;  
}
```

- Why copy and paste this **method** into every class that needs it!!
- Lets create a **class** that can be easily reused.

Input Class

```
import java.util.Scanner;

public class Input {
    // Prompted input of an int
    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
    // Prompted input of an float
    public static float getFloat( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        float result = in.nextFloat();
        return result;
    }
    // Prompted input of an word/token
    public static String getToken( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        String result = in.next();
        return result;
    }
}
```

Using our new class

Lets now write an application to make use of your new class:

```
public class SimplerAdd {  
    public static void main( String args[] ) {  
        int x = Input.getInt("Enter first integer");  
        int y = Input.getInt("Enter second integer");  
        int sum = x + y;  
        System.out.println( "Their sum is " + sum);  
    }  
}
```

But the compiler and JVM must be able to find the **Input** class (i.e. the .class file)

(Note: it will automatically find files that are in the **same folder** as the **.java file**)

NOTE: We will explore classpath and packages later.

static vs instance methods

As outlined, a **static** method is special, you can call it without first creating an object of the class in which the method is declared.

The **sqrt** method in the class **Math** is **static**. This method does not need to be invoked on a **Math** object, instead we just write:

```
double d = Math.sqrt(25);
```

static
method

The method **translate** in the **Point** class is **not static** – it must be invoked on a **Point** object

```
Point p = new Point(0,0);
```

```
p.translate(10,33);
```

instance
method

Input class

Improving our Input class

- Each method call creates a new **Scanner** object
- **Adding** a **field** and a **constructor** to the class, we can avoid this and **remove** the Scanner from each method call:

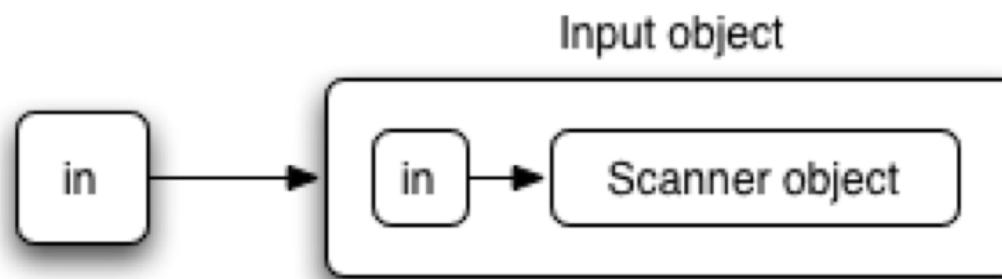
```
public class Input {  
    Scanner in; // Instance Variable  
  
    // Constructor  
    public Input() {  
        in = new Scanner(System.in);  
    }  
  
    public static int getInt( String prompt ) {  
        Scanner in = new Scanner(System.in);  
        System.out.print( prompt + " : " );  
        int result = in.nextInt();  
        return result; }  
}
```

Remove **static** and becomes **instance** method

Input class

Improving our Input class

- The methods of the class are no longer **static**.
- Each object of our **Input** class is *composed* of a single **Scanner** object.



New Input class

```
import java.util.Scanner;

public class Input {
    Scanner in;

    // Constructor
    public Input() {
        in = new Scanner(System.in);
    }

    // Prompted input of an int
    public int getInt( String prompt ) {
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }

    // Prompted input of a float
    public float getFloat( String prompt ) {
        System.out.print( prompt + " : " );
        float result = in.nextFloat();
        return result;
    }

    // Prompted input of a word/token
    public String getToken( String prompt ) {
        System.out.print( prompt + " : " );
        String result = in.next();
        return result;
    }
}
```

EXAMPLE: Input.java

Method Overloading

EXAMPLES: Output.java

```
public class Output {  
    public static void printArray( int[] a ) {  
        for ( int i = 0; i < a.length; ++i ) {  
            System.out.print( a[i] );  
            if ( i < a.length - 1 )  
                System.out.print( ", " );  
        }  
        System.out.println(); }  
  
    public static void printArray( String[] a )  
    {  
        for ( int i = 0; i < a.length; ++i ) {  
            System.out.print( a[i] );  
            if ( i < a.length - 1 )  
                System.out.print( ", " );  
        }  
        System.out.println(); }  
}
```

```
public class SimplerAdd {  
    public static void main( String args[ ] ) {  
        Input in = new Input();  
        int x = in.getInt("Enter first integer");  
        int y = in.getInt("Enter second integer");  
        int sum = x + y;  
        System.out.println( "Their sum is " + sum);  
    }  
}
```

Using the new Input class

Method Overloading

- Note that both methods on the previous page have **identical names** but **differ** in their **arguments**:

```
public static void printArray( int[] a )
```

```
public static void printArray( String[] a )
```

- A method is **overloaded** if the class containing it has at least **one other method** with the **same name** (but with different parameter types).
- Any number of methods can be defined with the same name as long as each overloaded method takes a **unique list of argument** types.