Programming Foundations FIT9131

Basic java constructs: selection, operators, variables

Week 3

Lecture outline

- selection – if construct
- relational operators
- boolean expressions
- compound statements
- nested ifs
- logical operators
- switch statement
- shorthand arithmetic operators
- local variables
- basic input

Reflecting on the naive-ticket-machine

The *naive-ticket-machine* is inadequate in several ways, for instance :

- no check to ensure that the customer has entered enough money for the ticket;
- no refund given if too much money is entered;
- no check to ensure that the customer enters a sensible amount of money;
- no check that the ticket price is set to a sensible amount.

Selection construct

We often want a program to do either one thing or another, depending on whether something is true or not.

We use a *selection construct* to do this.

A selection construct contains 3 components:

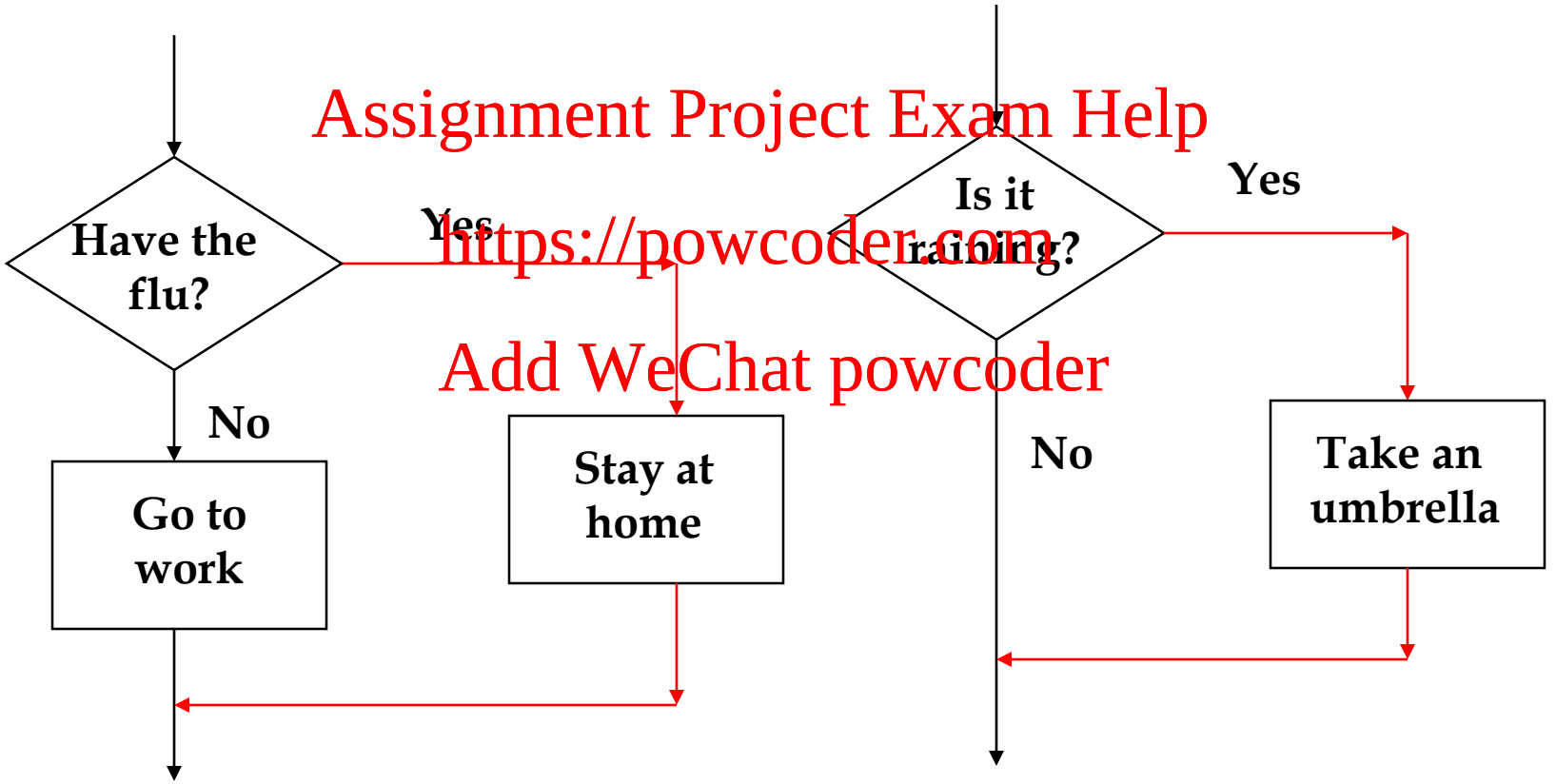
- a *condition* to be evaluated to true or false
- a *consequent* (what to do if the condition is true)
- an *alternative* (what to do if the condition is false). This component is optional.

Examples in English:

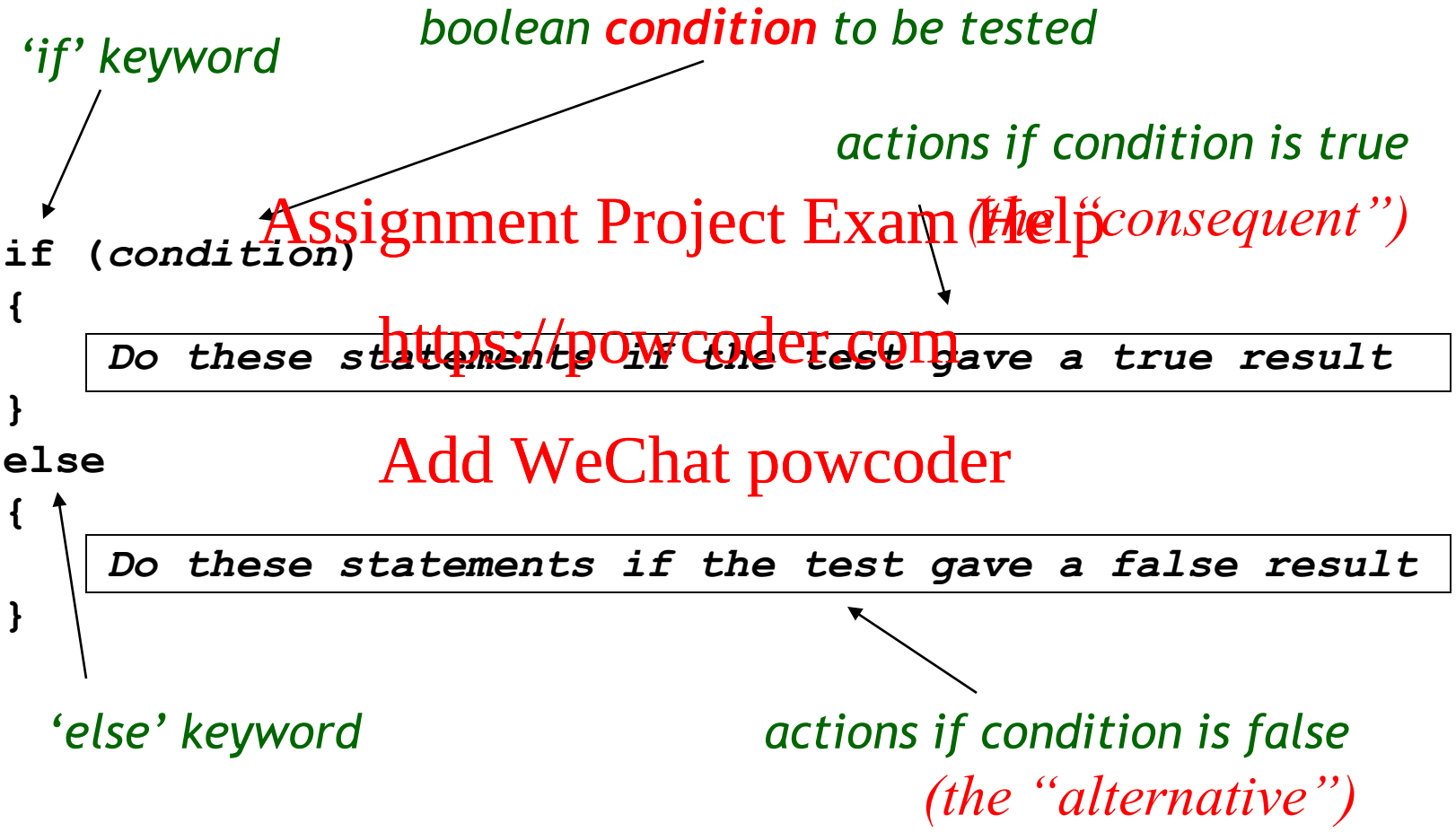
- *If you have the flu, stay at home, otherwise go to work.*
- *If it is raining, take an umbrella.*

Flow of control (making choices)

A selection construct can change the *flow of control* of a program.



Making choices – the IF statement



Relational operators

Expressions can be compared using *relational* operators. The result is a *boolean* value (i.e. `true` or `false`).

These are relational operators:

`==`	equal to
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

A *boolean expression* is an expression that returns a boolean value. These are often used as the *conditions* in selection constructs.

Some boolean expressions

Given the definition:

```
int age = 20;
```

What will the following expressions evaluate to?

```
age == 20
age != 20
age >  20
age >= 20
age <  20
age <= 20
```


Another example. Given the definition:

```
boolean isStudent = false;
```

The following is also a boolean expression:

```
isStudent
```

Some boolean expressions

Given the definition:

```
int age = 20;
```

What will the following expressions evaluate to:

```
age == 20   → true
age != 20   → false
age >  20   → false
age >= 20   → true
age <  20   → false
age <= 20   → true
```


Another example. Given the definition:

```
boolean isStudent = false;
```

The following is also a boolean expression:

```
isStudent
```

Some boolean expressions

Given the definition:

```
int age = 20;
```

What will the following expressions evaluate to:

```
age == 20   → true
age != 20   → false
age >  20   → false
age >= 20   → true
age <  20   → false
age <= 20   → true
```


Another example. Given the definition:

```
boolean isStudent = false;
```

The following is also a boolean expression:

```
isStudent   → false
```

Use of relational operators

```
if (age >= 18)   …

if (response == 'Y')…

if (price <= 0)   …

if (isStudent)   …

if (postcode.length() < 4)   …
```

Analysing the `insertMoney` *method*

Currently the `insertMoney` method of the `TicketMachine` class adds the amount of money entered to the balance without checking that it is a sensible value.


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

We would like to check that a positive amount is entered before we add it to the balance. If a negative amount is entered then we will display an *ERROR* message.

Digression – using a Java method

When we want to use a Java method, we typically use one of these following terms :

- *invoke* the method
- *call* the method
- *execute* the method

They all mean the same thing!!

In *BlueJ*, we would perform this by right-clicking the object, then selecting the method we wish to invoke/call/execute. In later weeks, we will also learn how to do this by writing the code directly.

Improving the `insertMoney` method

Here is a selection construct that would do this:


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


Note : the `if` statement itself does not return a value.

The `if` statement

The alternative for the `if` statement is optional.

```
public void insertMoney(int amount)  // another version
{
     if (amount > 0)
          balance = balance + amount;
     else
          System.out.println("Use a positive amount: " +
                             amount);
     if (balance >= price)
          System.out.println("Thanks. Enough money entered");
}
```


Example of an **"if"** statement without an "**else**" part

Compound statements

Sometimes a consequent or an alternative of an `if` has more than one statement in it. For instance, in English, we may say :

if you feel sick

- *go to the doctor (and)*

- *notify your employer that you will be away*

Both these statements are part of the consequent. There is no alternative in this example.

In Java, we would "group" statements together into a *compound statement*, delimited by braces.

Compound statements in Java

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

    ...

}
```


Example of a ***compound statement*** , consisting of 2 simple statements (note the use of the braces to enclose the statements)

Also note the way the code has been formatted (via the ***indentations***)

Nested `if` statements

Eg :

```
public void showCategory()
{
    if (age >= 14)
        if (isStudent)
            System.out.println("Student");
        else
            System.out.println("Adult");
    else
        System.out.println("Child");
}
```

The consequent of an `if`, like the alternative, may be a simple statement, or a compound statement, or another `if` statement.

Again note the way the code is ***indented***)

Note: In this example we assume that age has been defined as type int and isStudent has been defined as type boolean.

Bad formatting

```
public void showCategory()
{
if (age >= 14)
if (isStudent)
System.out.println("Student");
else
System.out.println("Adult");
else
System.out.println("Child");
}
```

This is the exact same code from the previous slide. Which one is more readable?

Make sure you read (and follow) the FIT9131 Java Coding Standards!!

Another cascaded `if` statement example

```
public void showCategory()
{
    if (age < 14)
        System.out.println("Child");
    else
        if (isStudent)
            System.out.println("Student");
        else
            System.out.println("Adult");
}
```

Logical operators

Boolean expressions can be combined using the *logical* operators *not* (!), *or* (||) and *and* (&&).

```
if (!isStudent) …
if (age >= 20 && !isStudent) …
if (age < 15 || isStudent) …
if ((age == 18 && isStudent) || isPensioner) …
```


The result of an *and* is true only if both *operands* are true.

The result of an *or* is false only if both operands are false.

A *not* results in the logical opposite of its operand.

Truth Tables

AND	true	false
true	true	false
false	false	false

OR	true	false
true	true	true
false	true	false

These 2 tables show the results when 2 ***boolean*** values are operated upon by the ***AND/OR*** operators. In programming, we typically call these values "**operands**".

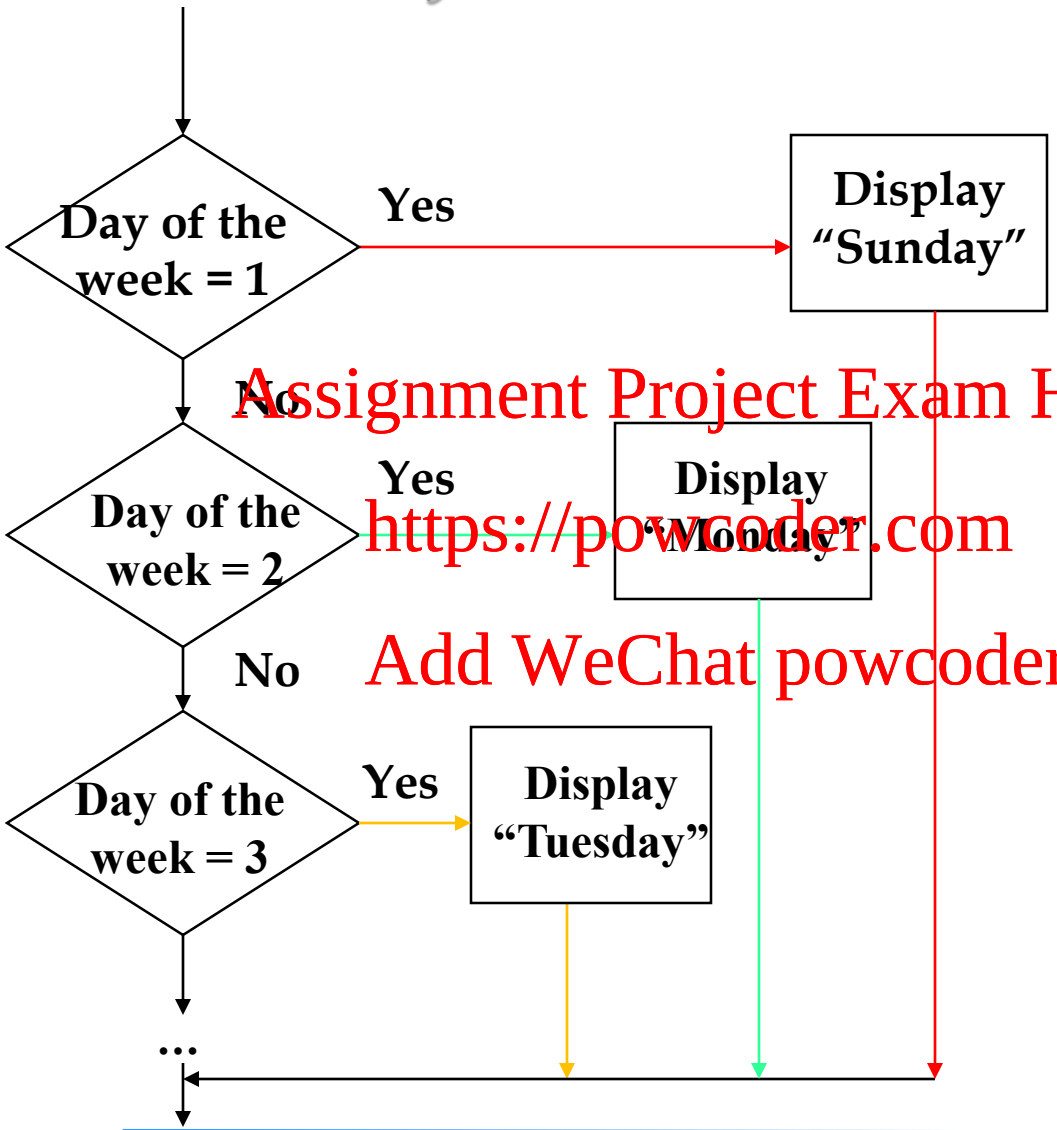
Another selection construct

Java has another selection construct called a `switch` statement.

The `switch` statement is a shorthand way of writing a `nested if`.

`if` and `switch` statements do not return values.

Flow of control (switch)



Example : Printing the name of a day

```
public void printDayName(int dayOfWeek)
{
    if (dayOfWeek == 1)
        System.out.println("Sunday");
    else
        if (dayOfWeek == 2)
            System.out.println("Monday");
        else
            if (dayOfWeek == 3)
                System.out.println("Tuesday");
            else
                ...
                    System.out.println("Invalid day");
}
```

This example uses a nested if statement. The result is not very tidy (it does work!).

Tidier : switch statement

The same example using a switch statement; makes the code much tidier.

```
public void printDayName(int dayOfWeek)
{
     switch (dayOfWeek)
     {
          case 1: System.out.println("Sunday"); break;
          case 2: System.out.println("Monday"); break;
          case 3: System.out.println("Tuesday"); break;
          case 4: System.out.println("Wednesday"); break;
          case 5: System.out.println("Thursday"); break;
          case 6: System.out.println("Friday"); break;
          case 7: System.out.println("Saturday"); break;
          default:
             System.out.println("Invalid day value"); break;
     }
}
```


What happens if we leave out the break statement(s) ?

When to use a `switch` statement?

A `switch` statement can be used only when:

- the expression in the condition is being compared for *equality* with the values in the case statements

Note: Prior to Java 7, the expression in the `switch` condition had to contain an ***`int`*** or a ***`char`***. As of Java 7, the `switch` condition can also contain a ***`String`*** type

Multiple `return` statements

A method can *return* only one value. As soon as it has returned the value, the method has finished.

It is possible to write a method with more than one `return` statement in it, but of course only one of them can be executed on each call!

Note that it is good coding practice to only use one `return` statement in a method.

Example : Multiple `return` statements in a method

This method tests to see if a year is a leap year. It returns true if it is.

```
public boolean isLeapYear(int year)
{
    if (year % 4 > 0)
        return false;
    if (year % 400 == 0)
        return true;
    if (year % 100 == 0)
        return false;
    return true;
}
```


Note that only ***ONE*** of these **`return`** statements will be executed at any one time.

Alternative code using single `return` statement

```
public boolean isLeapYear(int year)
{
  boolean result = true;
  if (year % 4 > 0)
     result = false;
  else
     if (year % 400 == 0)
        result = true;
     else
        if (year % 100 == 0)
           result = false;
  return result;
}
```

```
public boolean isLeapYear(int year)
{
     if (year % 4 > 0)
        return false;
     if (year % 400 == 0)
        return true;
     if (year % 100 == 0)
        return false;
     return true;
}
```


Which version is clearer?

Shorthand operators for integers

The arithmetic operators have *shorthand* equivalents:

```
int number = 0;
```



```
number += 1   is equivalent to   number = number + 1
number -= 1   is equivalent to   number = number - 1
number *= 2   is equivalent to   number = number * 2
number /= 3   is equivalent to   number = number / 3
number %= 3   is equivalent to   number = number % 3
```

The ++ operator

Java has another shortcut for the statement:

```
numberOfLines = numberOfLines + 1;
```

which uses the ++ (increment) operator:

```
numberOfLines++;
```


There is also a -- (decrement) operator.

```
numberOfLines--;
```


which is a shortcut for:

```
numberOfLines = numberOfLines - 1;
```

It is best to use the ++ and - - operators in statements by themselves, not as part of expressions.

Placement of ++ and --

The ++ operator increments a variable, but *when* this happens depends on the position of the ++.

- `number`**++** increments `number` *after* its original value is used in an expression.
- **++**`number` increments `number` *before* its original value is used in an expression.

The same applies to the placement of the **--** operator, e.g. **--**number and number**--**

Placement of ++ and --

What will the following code display?

```
int number = 3;
System.out.println(++number);
number = 3;
System.out.println(number++);
System.out.println(number);
```

Placement of ++ and --

What will the following code display?

```
int number = 3;
System.out.println(++number);
number = 3;
System.out.println(number++);
System.out.println(number);
```

4

3

4

Precedence rules for evaluating expressions

Highest precedence

Parentheses (brackets)

Unary operators `++ -- !`

Binary arithmetic operators `* / %`

Binary arithmetic operators `+ -`

Relational operators `< > <= >=`

Relational operators `== !=`

Logical operator `&&`

Logical operator `||`

Lowest precedence

Local variables

Methods can include *local variables* which exist only as long as the method is being executed.

- **they are only accessible from within the method.**

A local variable

No visibility modifier

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

Scope and lifetime

The *scope* of a variable defines the section of the source code from which the variable can be accessed:

- A formal parameter is available only within the body of the constructor or method that declares it.
- A field can be accessed from anywhere in the same class.
- The scope of a field can be modified using an *access modifier* (also called a *visibility modifier*). More about this later …

Scope and lifetime

The *lifetime* of a variable describes how long a variable can exist before it is destroyed:

- A formal parameter exists for the duration of the execution of the constructor (or method).

- A field exists while the object to which it belongs exists.

Fields, parameters and local variables

	Definition	Scope	Lifetime
fields	Defined outside constructors and methods	Within any methods or constructors of the class (if private) in which they are defined	Exist while their object exists
parameters	Defined in the header of the method or constructor	Within the body of the method in which they are defined	Exist while the method or constructor executes
local variables	Defined within the body of a method or constructor	Within the block in which they are defined, starting from the line where they are defined	Exist while the method or constructor executes

The `System` *class*

The `System` class contains some "constants" which represent the standard *input device* and standard *output device* (typically keyboard and screen) and methods to enable input and output.

The System class is part of the `java.lang` package.

Output to the screen

The standard output device is represented in the `System` class as '`out`'. The methods `print()` and `println()` are used to display string arguments to the standard output device.

Note : the "**.print(…)**" here means "call the **print** method of the **System.out** class.


```
//Display a string literal
System.out.print("Enter cat name");
```



```
//Display a string literal and then a carriage return
System.out.println("Enter cat name");
```

Basic input

In the code that we have written so far we have assigned values to variables in the code.

But how do we read in values from the user?

This can be a complex process, one that is difficult for novice Java programmers.

Java, however, has some built-in ways that it can help us get input from the user … we will use the `Scanner` class.

Basic input

The `Scanner` is a pre-defined Java class that we can use to obtain user input from the keyboard.

This requires the following steps:

1. import the `Scanner` class into our program
2. create a new `Scanner` object

3. invoke methods to read values from the `Scanner` object

Digression : creating an object without using BlueJ

In Java,we use the **new** keyword to create an object with our code, for example :


```
Scanner scannerName = new Scanner(System.in);
```


class name — object name — creating the object

```
Student newStudent = new Student("andy", "1234");
```

Step One: Import the Scanner

In order to use the `Scanner` class, we need to explicitly tell Java to "import" it into our program.

We do this by including an `import` statement at the very top of our code, above our class, Eg :


```
import java.util.*;

public class Welcome
{
    ...
}
```

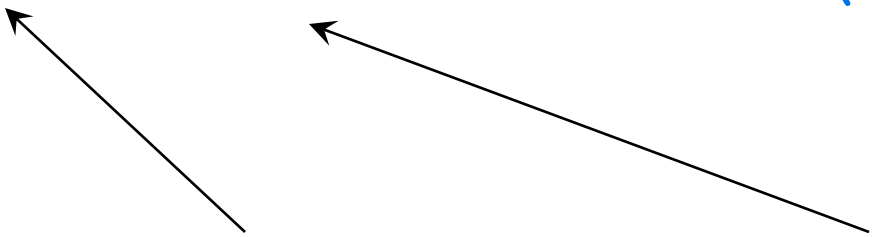
Step Two: Create a Scanner object

We then need to create a `Scanner` object, such as :

```
Scanner scannerName = new Scanner(System.in);
```


Eg

```
import java.util.*;
public class Welcome
{
    Scanner console = new Scanner(System.in);
}
```

Note:

1. in this case the `Scanner` object is named "`console`".
2. `System.in` is the standard input source (keyboard)

Step Three: Reading in data

We can now use methods from the `Scanner` object to read data in from the terminal. These allow us to specify the type of data to be read in:

`scannerName.nextLine()` ... for Strings including whitespace characters

`scannerName.next()` ... for Strings up to the first or next whitespace character encountered

`scannerName.nextInt()` ... for ints

`scannerName.nextDouble()` ... for doubles

`scannerName.nextLine().charAt(0)` ... for chars

Most of these commands will read the next piece of data, up until a white space character.

Step Three: Reading in data

Typically we will assign what we read in from the user into a variable (or series of variables)

For example:


```
Scanner console = new Scanner(System.in);
int age;

System.out.println("Please enter your age");
age = console.nextInt();
```

creating the scanner object

using the ***scanner object*** to read in an integer, and then assigns it to a variable named **age**

Basic input example

So putting it all together :

```
import java.util.*;
public class InputTest
{
    public void displayInput()
    {
        Scanner console = new Scanner(System.in);

        int number1 = 0;
        char letter1 = 'a';
        System.out.println("Please enter an int and a char " +
                           "separated by a space");
        number1 = console.nextInt();
        letter1 = console.next().charAt(0);
        System.out.println("You entered " + number1 +
                           " and " + letter1);
    }
}
```

Assignment 1 Released

Assignment 1 will be released this week

Progress by next week:

From last week's material

- **All class skeletons can be completed**

From this week's material

- **Basic input structures can be completed**

Remember to work logically in stages each week.