When you program an app, you use various components that have already been implemented by other programmers, e.g. in MkChange, we use IO.

These components are abstractions that you can use without knowing how they have been implemented. In Java, abstractions are defined as classes.

The simplest kind of class is called a module, e.g. IO. A module provides:

- **static methods**, i.e., operations that belong to the class and can be called by users (without them knowing their implementation), e.g. `IO.println`;
- **static constants**, whose values can be retrieved by users, e.g. `Integer.MAX_INT`;
- **static variables**, whose values can be retrieved and changed by users, e.g. `IO.fillChar`.

All of these features are static and belong to the class. You refer to them by:

```
ClassName.constantOrVariableName
ClassName.methodName(parameters)
```

The methods and fields of a class are related, e.g. for IO all are for doing input/output and formatting. The class serves to group them together.

The class also hides the details of how the operations are implemented from users of the class. The class may contain other methods and fields that are not public, i.e. not made available to the users. Information on the public methods and fields is collected in the class’s API, which the users can consult. The rest of the class’s definition is private and hidden from users.

There is also another kind of class where the user creates many instances or customized versions of the class template. These instances are called objects. Such classes have non-static methods and fields.

This week we only look at modules. Let’s go over the API of the IO class.
Methods Descriptions/Headers

A method description/header in an API specifies:

- the name of the method,
- the names and types of parameters it takes,
- the type of result it returns — if there is none, void is used,
- whether it is an instance or class (static) method.

E.g.
static void println(double value)
static void println(double value, java.lang.String fd)
static double readDouble()

Parameters

When we call a method, often need to pass some data to it. The method can support this by taking parameters. E.g. we pass the number to be printed to println through its value parameter.

The parameters are declared in the header of the method which appears in the API. Both the parameter name and its type are given.

Method Signature

The signature of a method is the number and types of its parameters and their ordering. E.g.

repeat: (int, char)
1st  IO.println: ()
7th  IO.println: (double)
8th  IO.println: (double, String)
11th  IO.println: (long)
12th  IO.println: (long, String)

A class may provide several methods with the same name if the signatures of the methods are different. This is called overloading.

To decide which overloaded method to call, the compiler looks at the number and types of the arguments.
Formatted Output

The IO class’s `print` and `println` methods allow you to print data in a specified format. The desired format is specified as an additional string argument.

IO.print(x, "w.d") will print x right-justified with d decimal places in a field of w characters; e.g.

```java
double y = 4.3333333;
IO.print(y,"8.3");
```

will print 4.333.

You can leave out the w or .d part of the format, e.g.

```java
IO.print(y,".1");
```

will print 4.3.

You get thousands separators by putting a comma in the format descriptor, e.g.

```java
IO.print(1234567,"12,");
```

will print 1,234,567.

To left-justify the output, use the L format flag, e.g.

```java
IO.print(1234567,"L12,");
```

will print 1,234,567.

```java
IO.print("John Smith","L20");
```

will print John Smith.

See the type package documentation for other features.

Boolean Expressions

Often, our programs will have to perform different actions depending on whether some condition is true or false, e.g.

```java
if (age <= 17)
    fare = 5.0;
else
    fare = 8.0;
```

or verify that a required condition holds at some point in the program, e.g. in input validation.

```java
IO.print("Enter the amount in cents: ");
int amount = IO.readInt();
IO.require(amount < 100, "Amount must be less than 100");
```

The condition may be quite complex. Such conditions are represented by boolean expressions.

Relational Operators

Simple boolean expressions can be obtained by comparing two numerical or char values using a relational operator, e.g.

```java
x < y
x >= 0
age == 17
```

The relational operators are:

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

Note: you cannot compare strings or objects using these.
Comparing Floating-Point Numbers

Note that because floating-point numbers have limited precision, you have to be careful when testing for equality. You probably want to consider two such numbers \( x \) and \( y \) equal if they are close enough, i.e. if

\[
|x - y| \leq \varepsilon.
\]

You may want to divide by the magnitude because precision decreases with it:

\[
\frac{|x - y|}{\max(|x|, |y|)} \leq \varepsilon.
\]

This can be coded as

```java
Math.abs(x-y) <= EPSILON * Math.max(Math.abs(x),Math.abs(y))
```

Logical Operators

More complex boolean expressions can be built using logical operators, e.g.

\[
13 \leq age && age <= 17
\]

\[
(13 \leq age && age <= 17) || age >= 65
\]

\[
!(13 \leq age && age <= 17)
\]

The logical operators are:

- `&&` conjunction - and
- `||` disjunction - or
- `!` negation - not

Note that as in logic, `!` has higher precedence than `&&`, which has higher precedence than `||`. So

\[
p && q || !p && r
\]

is interpreted as

\[
(p && q) || (!p) && r
\]

If you don’t want this interpretation, you must add parentheses.

The `&&` and `||` operators are evaluated left to right and the evaluation stops as soon as the answer can be determined; this is called lazy or short-circuit evaluation, e.g.

```
teen || student
```

It can be used to avoid errors such as division by 0, e.g.

```
d != 0 && n/d > 1
```

There is a primitive type boolean and you can also declare boolean variables (flags), e.g.

```java
boolean senior = age >= 65;
boolean child = age < 13;
boolean discount = senior || child;
```

But avoid the excessive use of boolean variables!