03 Problem Solving with Computers

[⛛ Close all sections] Only visible sections will be included when printing this document.

References: L&L 1.6

▽ What is a problem?

For our purposes, a "problem" is a situation where we have a known starting point (input data, a physical location, an existing application, etc.) and a known *goal* (result we wish to calculate, a different location to reach, an additional function in an application, etc.).

Problem *Solving*

The general steps involved in problem solving, regardless of the setting (real-world or computer) are:

1. Understand the problem
2. Dissect the problem into manageable pieces
3. Design a solution
4. Consider alternatives to the solution and refine it
5. Implement the solution
6. Test the solution and fix any problems that exist

Steps 5 and 6 are often done together, especially when building software: implement a little, test it, repeat.

Algorithms

An algorithm is a set of instructions for performing a task (i.e., solving a problem). An algorithm is a...

- **Sequence** of
  - order is important; must be able to determine the next step
- **Steps** that eventually
  - each instruction must be clear and executable
- **Stops** and which
  - the process must terminate
- **Succeeds**
  - the goal must be achieved on termination

There are many examples of algorithms: recipes, musical notation, knitting patterns, programs, and more.

▽ Program Development

The creation of software involves four basic activities:

- establishing the requirements
  - WHAT the program must do
- creating a design (HOW to solve the problem)
  - The ALGORITHM
- implementing the code
  - Use INCREMENTAL development
- testing the implementation
  - Incremental testing. The longer an error stays in a program the more expensive it is to fix

About implementation

Implementation—translating a design into source code—should be the *least* creative step (although we know that, early on, you will find this part challenging and need to do some experimentation to get things right)

- Important decisions are made during requirements analysis and design, and during algorithm development, *not* during coding!
  - Many software projects fail because the developer didn't understand the problem to be solved
- During implementation should focus on coding details, including style guidelines and documentation (see Appendix F of L&L)
- Testing: A program should be executed multiple times with various input in an attempt to find errors.

▽ 'High-level' algorithms

- Use existing libraries of code
  - Many Java libraries
  - These are organised into *packages* containing many different classes of objects
  - To use a class *C* from a package *p*, must import it: `import p.C;`
- Create objects that will be used to solve the problem
  - Interact with them via their methods
- Documentation is important
  - What classes of objects exist?
  - What methods do they have?
  - How do the methods work?
- Some sources of documentation
  - Lewis & Loftus
  - unit MyLO site
  - API docs, other books & web sites
  - The full Java API is far too big look at now; just be aware that it's there when you *do* need to look something up

An example

Imagine that we'd like to draw a square on screen and have decided to use a turtle graphics object (see Pass Task 2.1) to do that. The 'turtle' represents a drawing device that can be told to raise or lower its pen, rotate on the spot and move a given distance in the direction it is facing. Given that the turtle starts at the lower left of its window we could write the following algorithm to draw a square with sides of 100 pixels, 50 pixels away from the left and bottom edges of its window:

```
Variables:
Turtle turtle, the turtle graphics object

Steps:
create turtle
turtle pen up
turtle move(50)
turtle turn(90)
turtle move(50)
turtle pen down
turtle move(100)
turtle turn(90)
turtle move(100)
turtle turn(90)
turtle move(100)
turtle turn(90)
turtle move(100)
```

Which we could then translate into the following Java program (which combines the *declaration* of the Turtle object with its *creation* [also called *instantiation*]):

```
import kit101.turtle.Turtle;

public class HighLevel {

    public static void main(String[] args) {
        Turtle turtle = new Turtle(); //the turtle graphics object

        turtle.penUp();
        turtle.move(50);
        turtle.turn(90);
        turtle.move(50);
        turtle.penDown();
        turtle.move(100);
        turtle.turn(90);
        turtle.move(100);
        turtle.turn(90);
        turtle.move(100);
        turtle.turn(90);
        turtle.move(100);
    }
}
```

Note: To run this code you'll need to:

- Save it in a file called `HighLevel.java`
- Have the kit101 folder (containing the turtle subfolder and its .java files) in the same folder as the program. It's available as part of the starter code for Pass Task 2.1.

You will learn over time what abilities (known as *methods*) different kinds of objects have.

▽ 'Low-level' algorithms

'Low-level' algorithms use the constructs of the programming language:

- Set aside storage area
- Store value
- Overwrite value
- Arithmetic calculations
- Branch (depending on some property)
- Repeat (depending on some property)

An example

Note: This example uses many language features that have not yet been introduced, but it will run! You will learn these over time.

Imagine we'd like to calculate the area of a circle, which is given by $\pi \times r^2$, where r is the radius of the circle. For simplicity, in this example we won't get the values from the user, but build them into the program, and we will assume a circle radius of 100 units.

```
Variables:
double r, the radius of the circle
double area, the calculated result

Steps:
r = 100
area = π × r × r
display area
```

Which can be translated into the following short program:

```
public class LowLevel {

    public static void main(String[] args) {
        double r;   //circle radius
        double area; //calculated area

        r = 100;
        area = Math.PI * r * r;
        System.out.println("Area is " + area);
    }

}
```

Note: There is actually no hard distinction between 'high' and 'low' level algorithms. At various times you will think about the programs you write at both levels.

▽ ☑ Test Yourself

Activity: Read some code and spot the mistakes or predict the output.

Here is a sample program that has some syntax errors. *What are they?* Once you've had a guess, paste it into a new .java file in DrJava (saved it as `Reading1.java`) to see what the compiler says.

```
public class Reading1
    public static void main(String[] args)
        System.out.println("What's missing?");
```

Show Solution

Here is another sample program that has some syntax errors. *What are they?*

```
public class Reading2 {
    public static void main(String[] args) {
        System.out.println("Something is")
        System.out.println("not quite")
        System.out.println("right")
    }
}
```

Show Solution

Finally, what's the output from running this program?

```
public class Reading3 {
    public static void main(String[] args) {
        System.out.println("Strings are versatile.");
        System.out.println("You can append to their end: " + 10);
        System.out.print("println adds a newline, while ");
        System.out.println("print does not.");
    }
}
```

Show Solution