Learning Aims

- Understand the nature of and need for abstraction
- Describe the differences between an abstraction and reality
- Devise an abstract model for a variety of situations

Computational Thinking

Think how you can decompose the problem

Think about how the solution can be simplified and removing unnecessary details.

Think about inputs and outputs

Think logically Identify when decisions will need to be made

Think about how to make the program more **robust** by using a test table to check a range of user inputs

Think about which parts of the program can be run concurrently.

Think Procedurally how to break up the code using a function/procedure?

Think about how to make the program more **efficient** (execution time required or the space used)

Computational Thinker

- Computational thinkers aim for solutions that are both **correct and efficient**, not just any solution.
- Programmers must prove their solutions work using **logic, test data, and user feedback**.
- Computational thinking is essential for programming and problem-solving.
- It involves **thinking logically** and using compu**ting techniques to tackle challenges**.

Computational Thinking

- **Abstraction** - Hiding or removing irrelevant details from a problem to reduce the complexity.

- **Decomposition** -breaking down a complex problem or system into smaller, more manageable steps

- **Algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem using pseudocode to highlight problems

The difference between abstraction and reality

- Abstraction is a simplified representation of reality
- Abstraction is hiding unnecessary detail and showing details that are important.

Examples of abstraction:

- Creating simple abstract models of real world objects
- Designing interfaces with icons, symbols and colour coding
- Real-world values can be stored as variables and constants
- Implementation in a computer may involve abstract data structures: such as arrays, binary trees, graphs

Types of Abstraction

- Representational Abstraction
- Abstraction by generalisation
- Data Abstraction
- Procedural abstraction

Representational abstraction can be defined as a representation arrived at by removing unnecessary details.

Example of Representational abstraction

A map is an abstraction made up of a number of component parts.

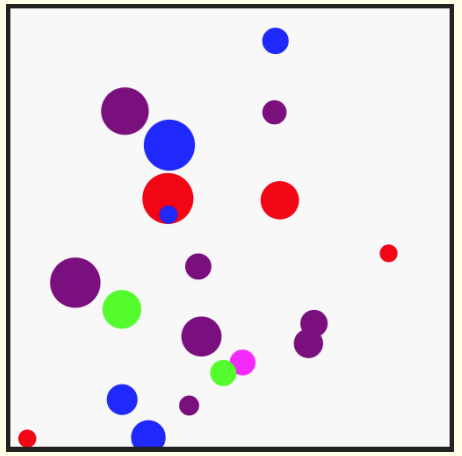
It is a means of hiding detail / only using relevant detail

It is a representation of reality using symbols and labels to show real-life features and irrelevant features, such as buildings and trees are left out.

Examples of Representational abstraction

If you are planning to write a program for a game involving a bouncing ball, you will need to decide:

- What **properties** of the ball to take into account.
- If it's bouncing vertically rather than, say, on a snooker table, gravity needs to be taken into account.
- How elastic is the ball? How far and in what direction will it bounce when it hits an edge?
- What you are required to do is **build an abstract model of a real-world situation**, **which you can simplify**; remembering, however, that the more you simplify, the less likely it becomes that the model will mimic reality.



Simulations and Abstraction

Abstraction in simulations means **simplifying a real-world system by focusing only on the important details** and ignoring unnecessary complexities. This makes it easier to model and understand complex systems.

How Abstraction Helps in Simulations

- **Removes unnecessary details** – Focuses only on key aspects (e.g., in a car simulation, we model speed and fuel but ignore minor engine details).
- **Reduces complexity** – Makes it easier to build and run the simulation.
- **Improves efficiency** – Uses fewer resources, making the simulation run faster.

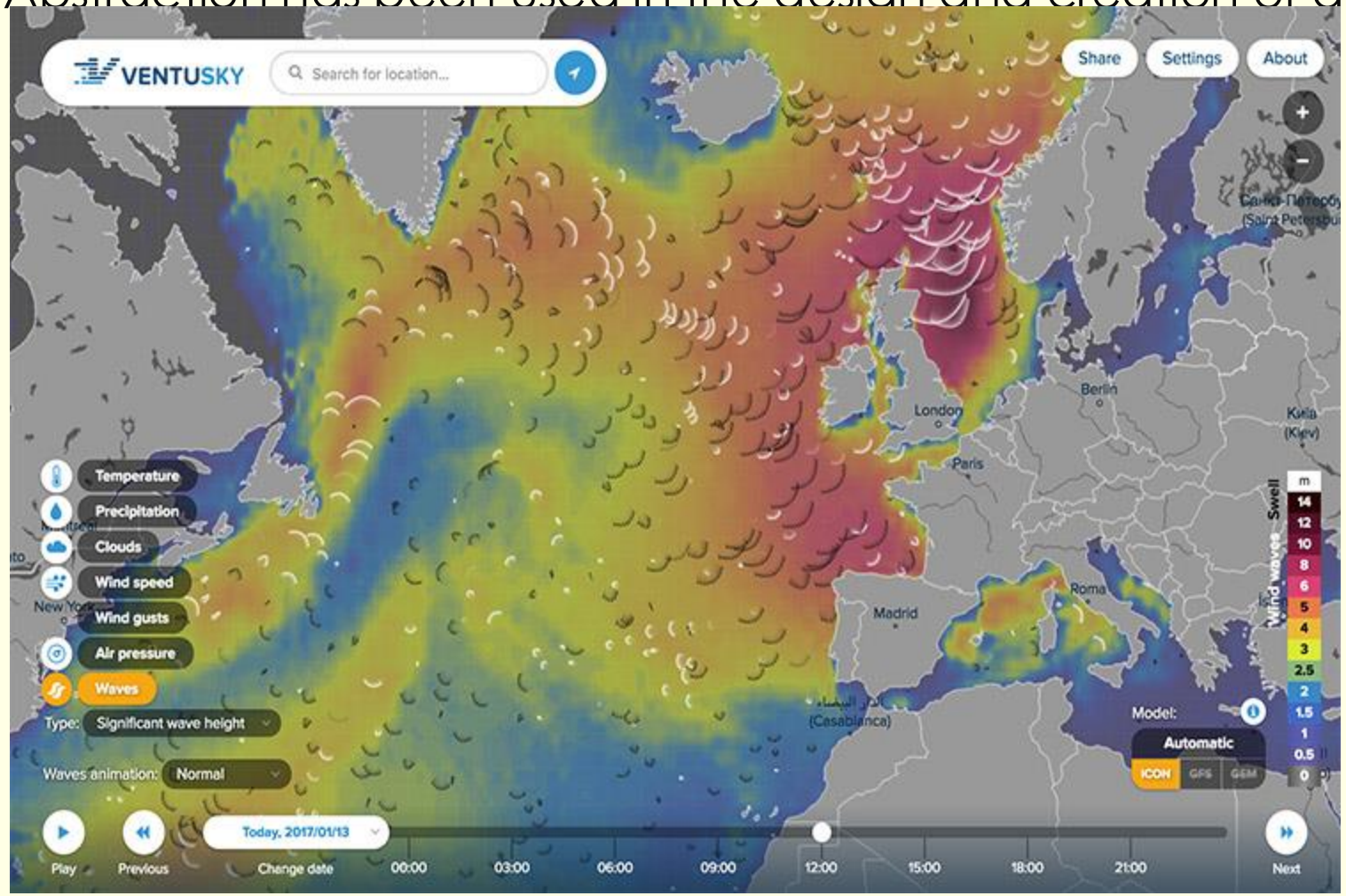
Examples of Simulations Using Abstraction

Example	Real-world system	Abstracted model:
Traffic Simulation	Roads, cars, pedestrians, weather, signals.	Only include **cars, traffic lights, and road rules** while ignoring car brands, engine types, or driver emotions.
Weather Simulation	Includes atmosphere, temperature, wind, humidity, pressure.	Focus only on **temperature, pressure, and wind speed** to predict storms.
Flight Simulator	Airplane mechanics, fuel system, engine power, wind resistance, pilot reactions.	Focus only on **flight controls, aerodynamics, and navigation**, ignoring mechanical details like engine pistons.

Why Use Abstraction in Simulations?

- **Faster and easier to build** – Only key details are included.
- **More understandable** – Helps analyse and predict behaviours efficiently.
- **More flexible** – Can be adjusted based on different needs (e.g., a simple car game vs. a detailed driving simulator).

Abstraction has been used in the design and creation of a weather map



What are the essential features that are included in this abstract model?

What unnecessary detail has been removed?

Abstract models of real-world objects

Car (Vehicle Abstraction)

Real-World Object: A car has many components like an engine, fuel system, transmission, etc.

Abstract Model: A **Car class** in programming might only include essential attributes and methods:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        print("Engine started")


    def drive(self):
        print("Car is moving")
```

This abstracts away the **complexity of how a car actually works** (like fuel injection, combustion, and transmission).

This type of abstraction is very common in object-oriented programming

OCR

A LEVEL

COMPUTER SCIENCE

REMASTERED

2.1.1D

DEVISE AN ABSTRACT MODEL

Abstraction applied to high level programming languages

Abstraction is the most important feature of high level programming languages such as Python, C#, Java and hundreds of other languages written for different purposes.

To understand why, we need to look at different generations of programming language.

The first generation of language was machine code – programmers entered the binary 0s and 1s that the computer understands. Writing a program to solve even a short, simple problem was a tedious, time-consuming task largely unrelated to the algorithm itself.

The second generation was an improvement; mnemonic codes were used to represent instructions. It is still an enormously complex task to write an assembly language program and what's more, if you want to run the program on a different type of computer, it has to be completely rewritten for the new hardware.

The third generation of languages, starting with BASIC and FORTRAN in the 1960s, used statements like X = A + 5, freeing the programmer from all the tedious details of where the variables X and A were stored in memory, and all the other fiddly implementation details of exactly how the computer was going to carry out the instruction.

Finally, programmers could focus on the problem in hand rather than worrying about irrelevant technological details, and that is a good example of what abstraction is all about.

Abstraction by generalisation

- Groups similar problems together to form a **general model** or **pattern**.

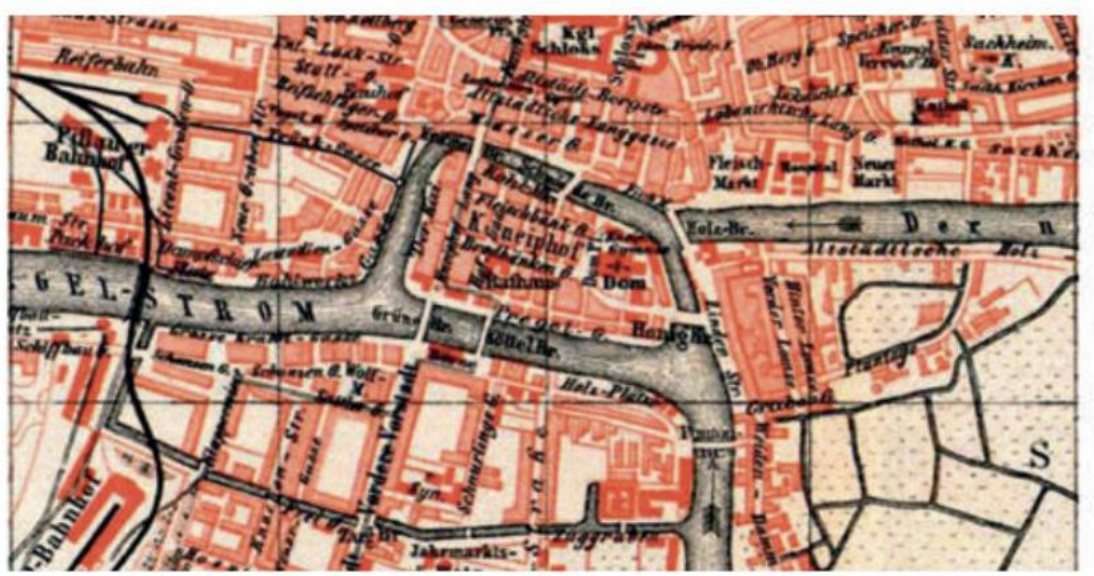
- Reduces problem-specific details to create a **reusable solution**.

Abstraction by generalisation

There is a famous problem dating back more than 200 years to the old Prussian city of Königsberg.

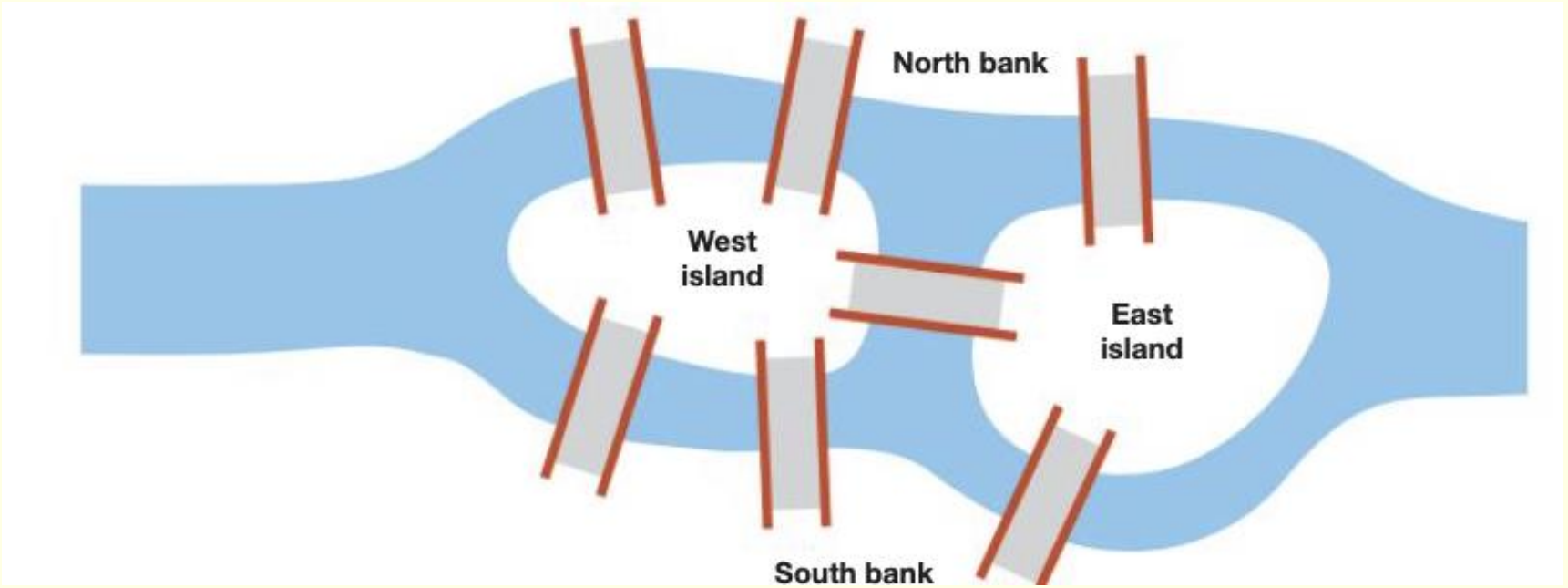
This beautiful city had seven bridges, and the inhabitants liked to stroll around the city on a Sunday afternoon, making sure to cross every bridge at least once.

Nobody could figure out how to cross each bridge once and once only, or alternatively prove that this was impossible, and eventually the Mayor turned to the local mathematical genius **Leonhard Euler.**



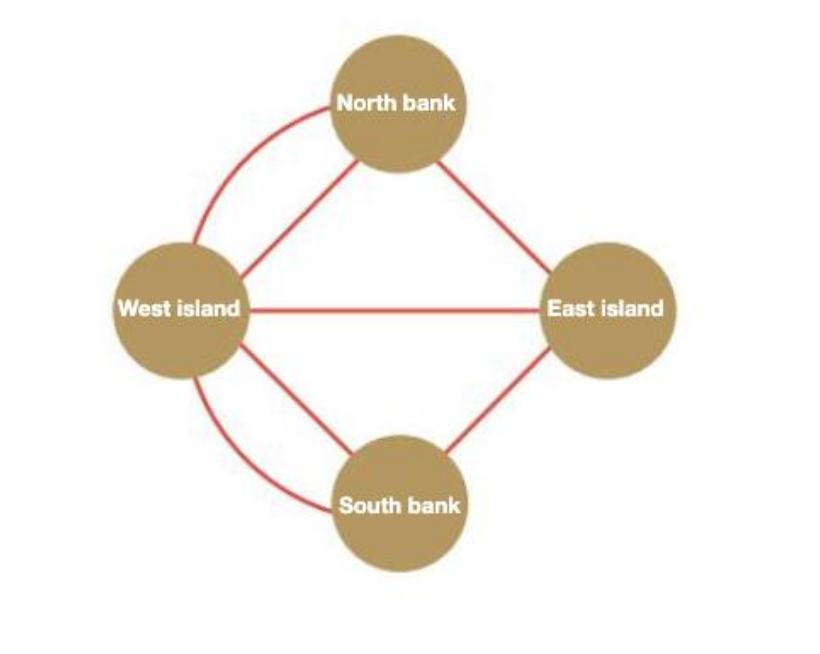
Solving the seven bridges

- Euler's first step was to remove all irrelevant details from the map, and come up with an abstraction:



Solving the seven bridges

- To really simplify it, Euler represented each piece of land as a circle and each bridge as a line between them.



What he now had was a graph, with nodes representing land masses and edges (lines connecting the nodes) representing the bridges.

Now that Euler had his graph, how could he solve the problem?

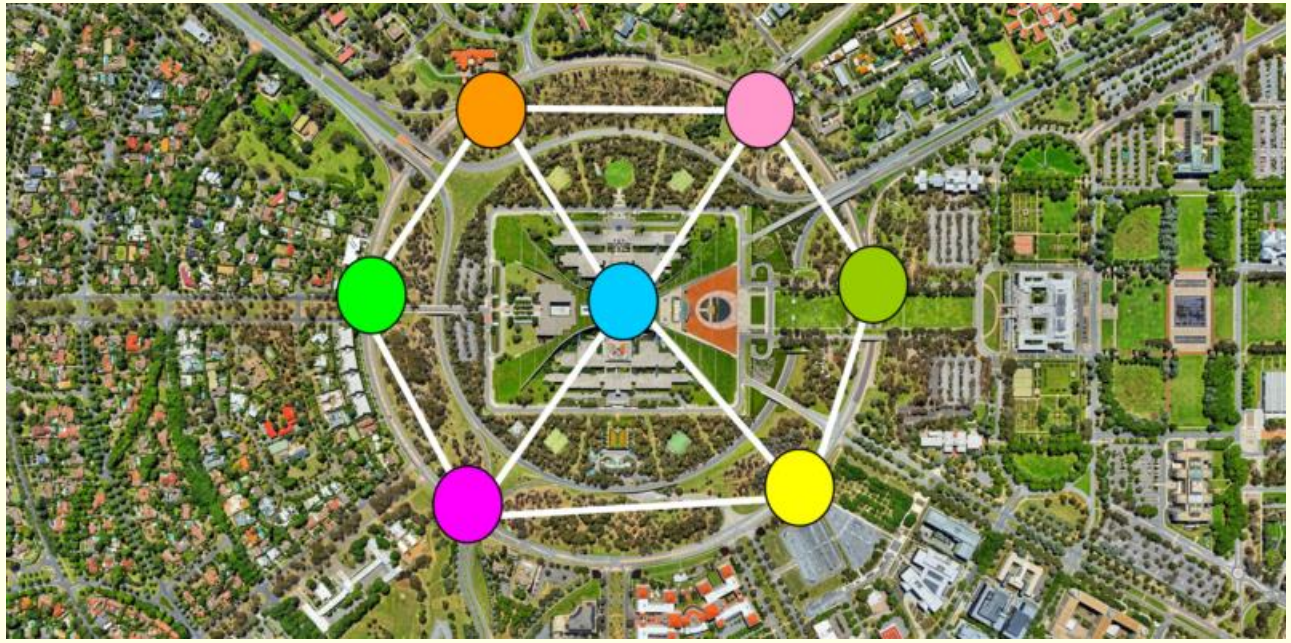
Solving the seven bridges

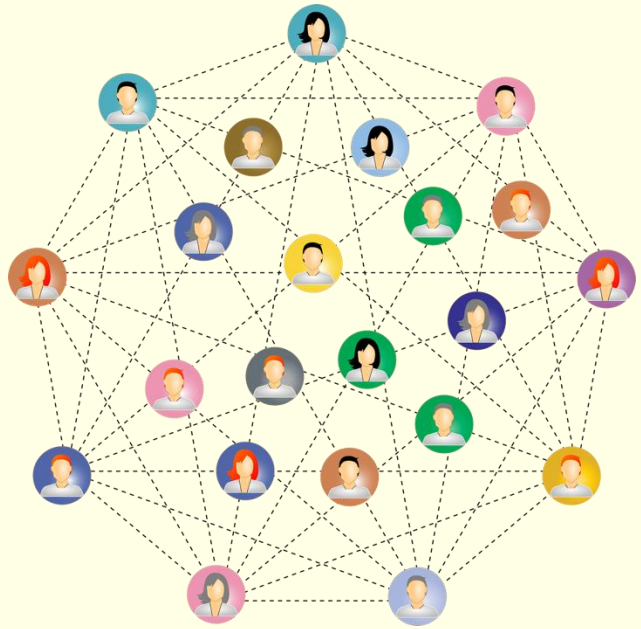
- Euler realised he didn't need to try every solution; instead, he focused on finding a general solution for similar problems.
- He noticed that to cross each bridge only once, each point (node) must have an even number of connections, except for the start and end points.
- Since all the points in the puzzle had an odd number of connections, **solving it was impossible!**
- Euler's work formed the basis of graph theory, allowing similar problems to be solved through abstraction.
- **By generalizing the problem**, Euler showed how it applies not just to cities with bridges but to **many other problems** with similar rules.

Graph data structure is used with maps but can also be used on **social media** sites.

Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them.

Facebook's Friend suggestion algorithm uses graph theory.





Abstraction by generalisation

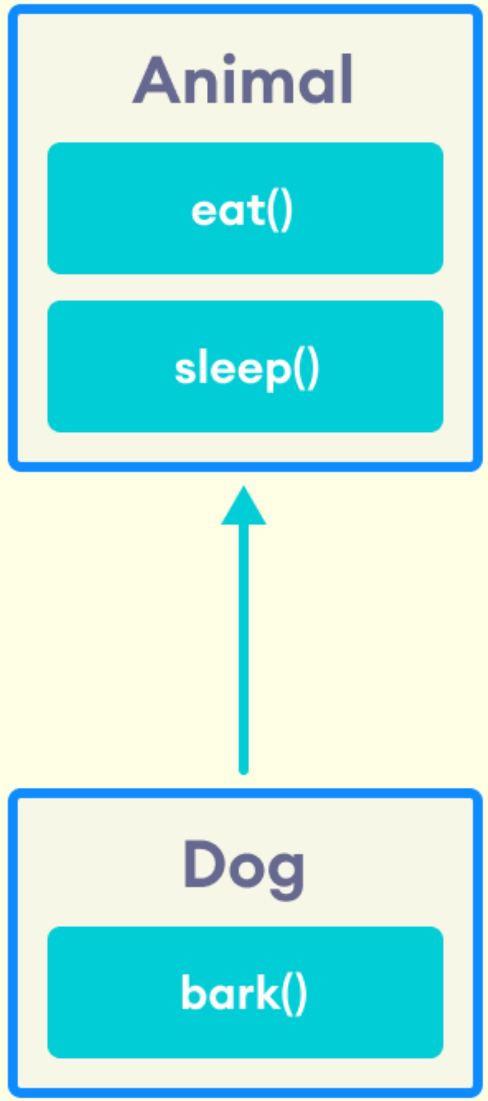
- Also means **grouping similar things into a more general category** by identifying what they have in common.

- Instead of focusing on specific details, we create a broad category that applies to all similar objects.

Example:

dog, cat, and bird → Animal

- A **dog, cat, and bird** are all animals.
- Instead of treating them separately, we group them under **"Animal"**, since all animals **eat, sleep, and move**.

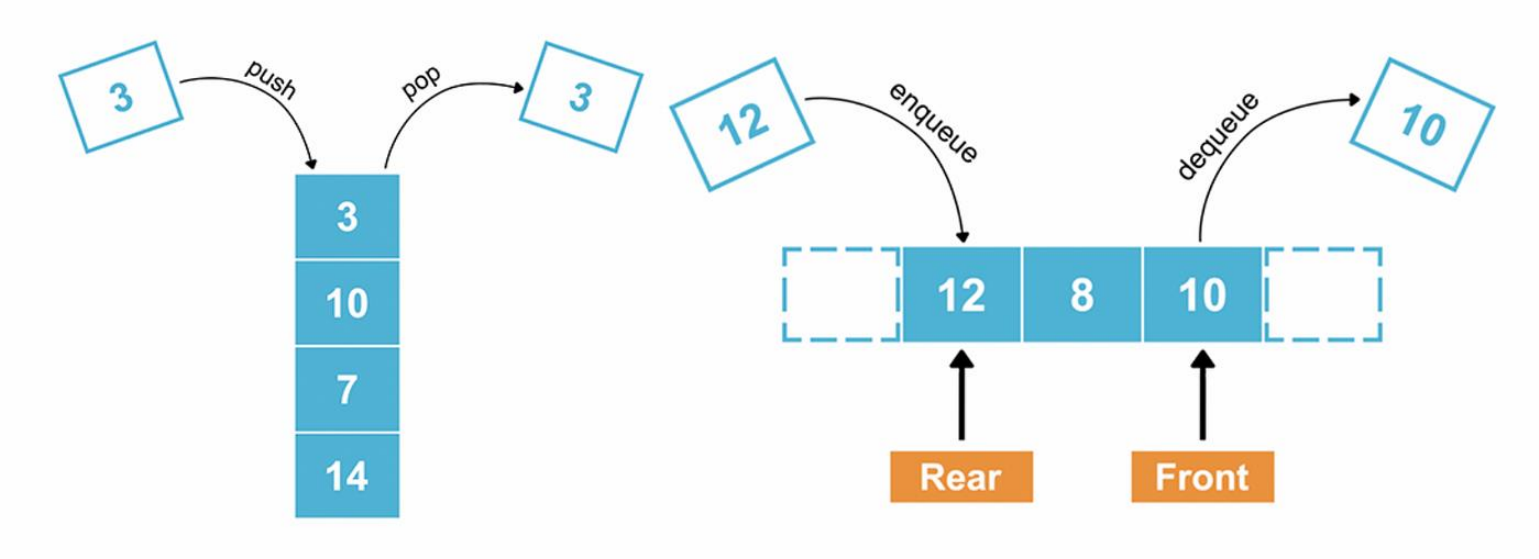


Data abstraction

- Hides the way **data is stored** and manipulated, providing only necessary details.
- Uses **abstract data types** such as stacks, queues, and lists.

Example:

- Using a stack.push(x) and stack.pop() without knowing the internal array or linked list implementation.



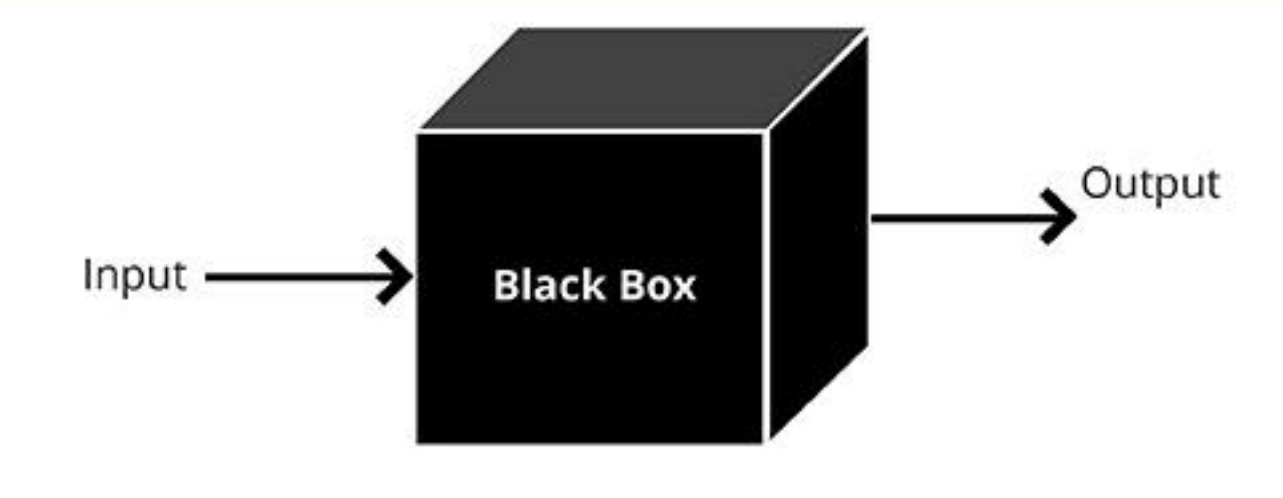
This is an abstract model for explaining how to stack and queue work. How stacks and queues are actually stored in memory and implemented is hidden.

Procedural abstraction

- Focuses on **breaking down** a problem into smaller procedures (subroutines or functions).
- Hides **implementation details**, allowing users to call a procedure without knowing how it works internally.

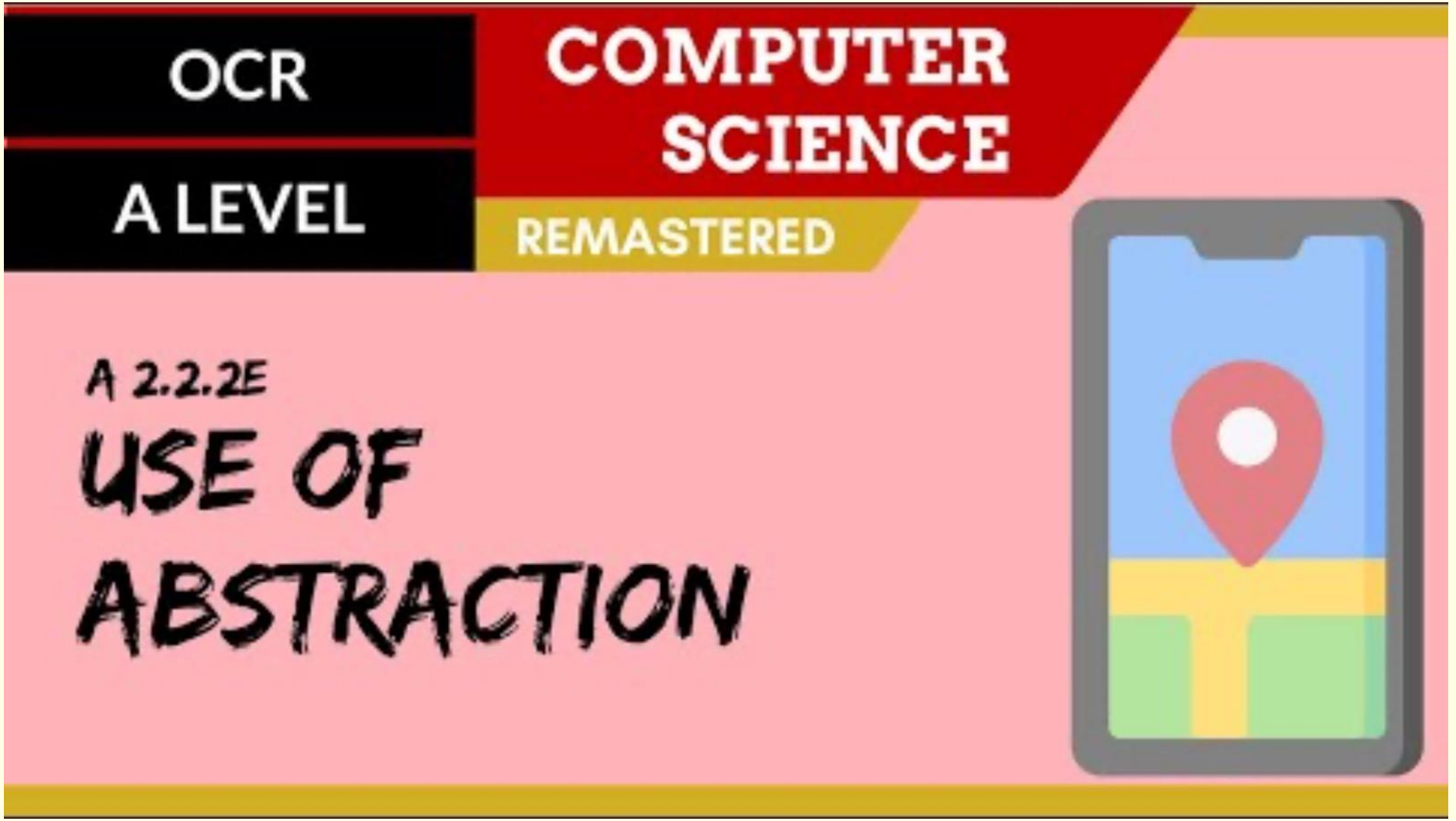
Example:

- A function calculateInterest(amount, rate, years) computes interest without revealing its internal formula.

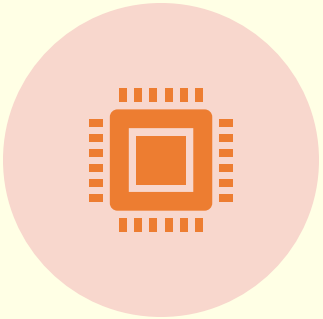


Video

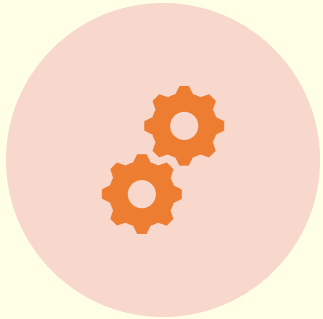
- https://youtu.be/xNHqa5oGCnM?feature=shared



Why abstraction ...



Reduces programming time and factors that can detract from the program - programmers can focus on core elements rather than unnecessary data



Reduces complexity of programming

Focus is on the core aspects of the program rather than the extras

Too much abstraction can detract from the appeal of the application I.e. a game, may be too simplistic/not realistic enough, may not have enough scope to engage users