1) C# types:
   Value types (memory contains data)
      - Built-in
      - User-defined
      - Enumeration
   Reference types (reference to memory where data stored)
      - Pointers
      - Interface
      - Self-describing (delegates)
   Struct/Classes: classes are reference type
      - classes can be extended
      - classes can have parameterless constructor
      - Child of Object
   Struct are value types
      - structs cannot be extended
      - struct cannot initialize fields
      - struct cannot have parameterless constructor
      - Child of Object

   Boxing/unboxing using Object (data loss)
   Nullable types: int?
   Arrays/Jagged Arrays (rows may have different length)
   switch (must have break)
   while/for/foreach
   All objects/arrays/strings are passed by reference

2) Objects have fields, attributes, properties, actions
   Object is an instance of a class
   Objects created from one class are not identical
   Anonymous types: var circle = new {radius=3};
   Access modifiers (public/private)
      - static fields are shared by all objects of that class
      - protected: can be accesses by derived classes only
   overloading (same name, different parameter lists)
   class implementing interface, must implement all its methods

3 pillars of OO:
   - Encapsulation
   - Specialisation
   - Polymorphism

3) Stream is an object that represents a sequence of bytes
   Any data marked as Serializable can be transformed to a stream (serialisation)
   Streams can be used to read/write data, move data between machines
   ToString() is an example function of serialisation
   To serialize an object: serialize all values by writing them into result buffer
   Naive serialization - ToString()
   DB access:
      - ADO.NET
      - LINQ (DB&XML)

4) LINQ: SQL queries like commands rather than explicit SQL Strings (like in ADO)
   Lambda notation: can be used for anonymous functions:
      Thread t = new Thread(
         () => ....anonymous function....
      );

5)
Indexers (Treats class as if it is an array)

```
public string this[int index] { get { .. } set { .. } }
```

Generics (polymorphism). Pre-defined classes (List, Stack, Queue, Dictionary)

Collections (framework for putting objects of the same type together. Can make a collection by implementing certain interfaces [IEnumerable, ICollection])

Exceptions (help to deal with foreseen errors)

Delegates (defines higher-order function [function that takes other functions as arguments]. Delegate refers to a method) Example: in GUI when handling events

Anonymous methods: better to use instead of delegates

6) Threads share memory

Processes do not share memory
Every process has different address space
Use threads to maximise the concurrency of execution

Thread creation - mutual exclusion - event waiting - waking up a thread - lock
Thread creation: new thread object -> pass it to ThreadStart delegate -> call start method

Forking: creating and starting thread

join() used to join threads (one is waiting for another to complete)

Critical region

Monitor / lock(obj)
Thread is blocked until thread is released from lock
Monitor: Wait, Pulse, PulseAll (Pulse* - awakens queued thread waiting for locked object)

Race condition

Deadlocks (avoid using locks)

Starvation (opposite of deadlocks)

7) strong typing

heap
Managed code (runs within .NET CLR)
Unmanaged code (code from outside)
safe code
unsafe code (involves unsafe operations - pointer operations) [direct memory access]. Direct access must be marked with FIXED. Must be marked with UNSAFE

&x - address

*x - value

external function must be declared in C# code

8) Moores law - no more free lunch

shared-memory model
parallel programming

Imperative programming languages (functional) / OO

2 types:

- Data parallelism
- Task parallelism

Parallel.For(..)
loop.[Break()] || [Stop()]
Parallel aggregate (MapReduce)
Parallel.ForEach() <- aggregate
Partitioner to split entire range into sub-ranges (Partitioner can be self-defined)
Parallel.Invoke(..) <- task parallelism
WaitAll, WaitAny
Parallel.SpeculativeInvoke <- start many, need one result only [ex. finding element]

Task<int> FutureVar (lazy) = ....
Divide-And-Conquer

Pipeline (sequence of operations, where the output of i-th stage becomes input to i+1 stage) - parallelism can be achieved by overlapping the computations (pipelining)