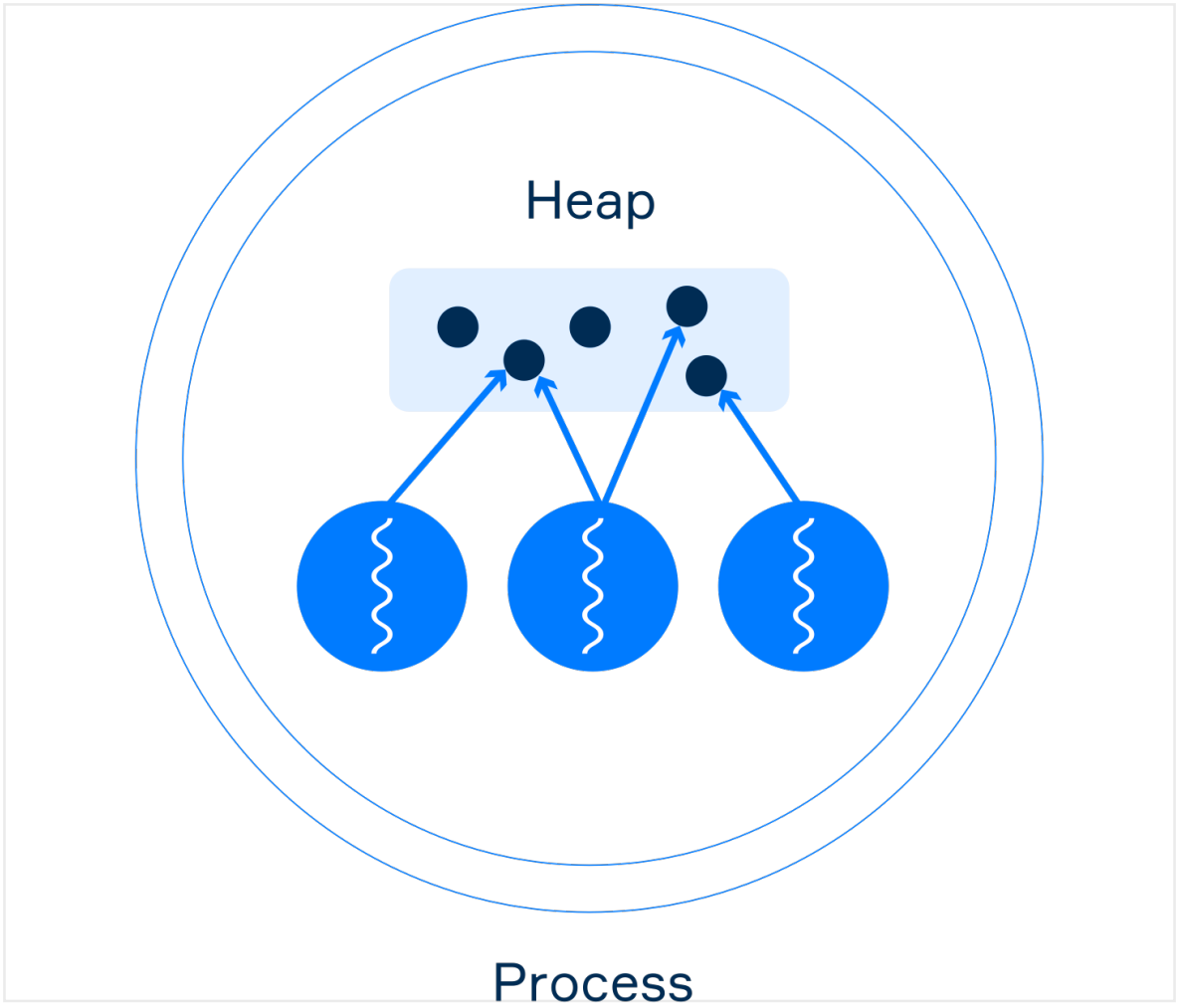
Shared Data

Sharing data between threads

Threads that belong to the same process share the common memory (that is called **Heap**). They may communicate by using shared data in memory. To be able to access the same data from multiple threads, each thread must have a reference to this data (by an object).



Multiple threads of a single process have references to objects in the Heap.

```
class Counter {

   private int value = 0;

   public void increment() {
      value++;
   }
```

```
    public int getValue() {
        return value;
    }
}

class MyThread extends Thread {

    private final Counter counter;

    public MyThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        counter.increment();
    }
}

Counter counter = new Counter();

MyThread thread1 = new MyThread(counter);
MyThread thread2 = new MyThread(counter);

thread1.start(); // start the first thread
thread1.join();  // wait for the first thread

thread2.start(); // start the second thread
thread2.join();  // wait for the second thread

System.out.println(counter.getValue()); // it prints 2
```

As you can see if you try it by yourself the result is 2, because both threads work with the same data by using a reference.

In this example, we started the first thread and waited until it has completed its work (by this time an increment happened), then we started the second thread and waited till it also has completed its work (increment's happened again). The result is exactly as we would've expected.

When you write your code in different threads that work with the same data concurrently, it is important to understand a few things:

- some operations are non-atomic;
- changes of a variable performed by one thread may be invisible to the other threads;
- if changes are visible, their order might not be (reordering).

Thread interference

A non-atomic operation is an operation that consists of multiple steps. A thread may operate on an intermediate value of non-atomic operation performed by another thread. This leads to a problem called **thread interference:** the sequences of steps of non-atomic operations performed by several threads may overlap.

Note: in the previous example, the two threads did not work with the data at the same time. Before the start of the second thread, the first has already terminated.

The operation value++ can be decomposed into three steps:

1. read the current value;
2. increment the value by 1;
3. write the incremented value back in the field;

Since the increment operation is non-atomic and takes 3 steps to work, the **thread interference** may occur in case two threads call the method increment of the same instance of Counter.

In the same way, the operation value-- may be decomposed into three steps.

Now if Thread A invokes the method increment of this instance and Thread B also invokes the method at the same time, the following happens:

1. **Thread A:** read value from the variable.
2. **Thread A:** increment the read value by 1.
3. **Thread B:** read value from the variable (it reads an intermediate value 0).
4. **Thread A:** write the result in the variable (now, the current value of the field is 1).
5. **Thread B:** increment the read value by 1.
6. **Thread B:** write the result in the variable (now, the current value of the field is 1).

The result of Thread A was lost, overwritten by Thread B. Although sometimes the result may be correct, this particular interleaving is possible.

Let's consider another case: an assignment of 64-bit values. It may be surprising, but even the reading and writing fields of double and long types (64-bits) may not be atomic on some platforms.

```
class MyClass {
```

```
    long longVal; // reading and writing may be not atomic

    double doubleVal; // reading and writing may be not atomic
}
```

It means while a thread writes a value to a variable, another thread can access an intermediate result (for example, only 32 written bits).
To make these operations atomic, fields should be declared using the volatile keyword.

```
class MyClass {

    volatile long longVal; // reading and writing are atomic now

    volatile double doubleVal; // reading and writing are atomic now
}
```

The reading of and writing to the fields of other primitive types (boolean, byte, short, int, char, float) are guaranteed to be **atomic**.

Visibility between threads

Sometimes, when a thread changes shared data, another thread may not notice these changes or obtain them in a different order. It means different threads may have inconsistent views of the same data.
The reasons are different, including caching values for threads, compiler optimisation, and more.

Example.
Here's an int field, defined and initialised:
int number = 0;
The field is shared between two threads: Thread A and Thread B.

Thread A increments the number by 5.
number += 5;

Right after it, Thread B prints number in the standard output:
System.out.println(number);

The output may be either 0 or 5, because there is no guarantee that the change performed by Thread A is visible to Thread B**.**

As we've already mentioned, the keyword volatile is used for **visibility**. To make visible changes of a value made by one thread to other threads, we should declare the field with the keyword volatile.

volatile int number = 0;

When the field is declared as **volatile** all changes made to this field by a thread are guaranteed to be visible for another thread when it's reading the value from this field.
The volatile keyword may be written in an instance and static fields declaration.

Other cases of visibility

Sometimes we don't need to write the volatile keyword. The following procedures will also guarantee visibility:

- changes of variables performed by a thread **before starting** a new thread are always visible to the new thread;
- changes of variables inside a thread are always visible to any other threads after it successfully returns from a join on the thread (we used this one at the beginning of this topic).

They are formalized using a special relationship named **"Happens-before"**.

More on volatile keyword

Again, the volatile keyword allows us to make visible changes of a field made by one thread to other threads. This keyword also makes writing
to double and long fields atomic. But the keyword doesn't make the increment/decrement and similar operations atomic.

The volatile keyword in Java is used to mark a Java variable as “being stored in main memory”. Every thread that accesses a volatile variable will read it from main memory, and not from the CPU cache. This way, all threads see the same value for the volatile variable.

Check more on:

https://medium.com/codex/volatile-keyword-in-java-9e8792b4e6ba#:~:text=The%20volatile%20keyword%20in%20Java,value%20for%20the%20volatile%20variable

https://www.geeksforgeeks.org/volatile-keyword-in-java/

Primitive types other than long and double are guaranteed to be atomic only for read and write operations. For operations like increment-decrement they are not thread-safe. Since the question doesn't specify what kind of changes, it

can be assumed all types possible changes can be done including increment and decrement.