## Thread Termination

A thread terminates when `run` terminates.

Sometimes we want it to keep going until another thread tells it to stop.

Call the `void interrupt()` method of a thread to deliver an *interrupt* to it. Consequences:

- If the thread is blocked by `sleep`, it throws an `InterruptedException`.
- If the thread is blocked by an I/O method, it throws an `InterruptedIOException` (subclass of `IOException`).
- If the thread is not blocked, we can call `isInterrupted` or `interrupted` to find out if someone has tried to interrupt it.

The code of a thread should be designed to check these conditions and be ready to terminate *cleanly* and *promptly*.



## Interrupt Example 1

Example: reading from a stream.

```
try {
  while (!isInterrupted()) {
    r = input.read(buf);
    // ...
  }
}
catch (InterruptedIOException e) {}
catch (IOException e) {
  System.err.println(e.toString);
}
```

Another way to put it:

```
while (!isInterrupted()) {
  try { r = input.read(buf); }
  catch (InterruptedIOException e) { break; }
  catch (IOException e) {
    System.err.println(e.toString);
    break;
  }
  // ...
}
```



## Interrupt Example 2

If a method does not throw `InterruptedException`:

```
void tick() {
  try {
    sleep(10);
    System.out.println("ping");
  }
  catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}

void run() {
  while (!isInterrupted()) { tick(); }
}
```

Alternatively, `tick` could throw `InterruptedException` instead.



## Interrupt Subtleties

- `interrupted` is a static method. It is about the *current* thread. Also, it clears the interrupt.
- `isInterrupted` is a non-static method. It is about the `Thread` object it is called on. Also, it does not clear the interrupt.
- When `InterruptedException` is thrown, the interrupt is also cleared.

  So the handler should either terminate the thread itself or re-establish the interrupt.

## Race Conditions

If two threads modify the same variable at the same time, we get a *race condition*.

E.g., consumer thread:

```
while (!q.empty()) {
  q.unqueue();
}
```

producer thread:

```
for (;;) {
  q.queue(10);
}
```

The internal state of the queue may become inconsistent because the queue operations are not *atomic*.



## Mutual Exclusion

Every object is associated with a unique lock.

The *synchronized statement*:

```
synchronized (ref) {
  //...
}
```

- Acquires the lock on the object `ref` before executing the body.
- Blocks if another thread has the lock until it is released.
- Releases the lock afterwards.

Any object can be chosen for the locking.

- Usually the same object to be manipulated.
- But can also be an unrelated object.



## Mutual Exclusion Example

Consumer thread:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
  }
}
```

Producer thread:

```
for (;;) {
  synchronized (q) {
    q.queue(10);
  }
}
```

Exercise: why not use

```
synchronized (q) {
  while (!q.empty()) q.unqueue();
}
```

in the consumer thread?



## Mutual Exclusion Sweetened

Manually writing synchronized statements is mundane and error-prone.

Would be nice to have *synchronized methods*.

```
public synchronized void queue(int x) ...
public synchronized int unqueue() ...
public synchronized boolean empty() ...
```

Calling a synchronized method of an object locks it.

Consumer thread:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
  }
}
```

Producer thread:

```
for (;;) { q.queue(10); }
```

Exercise: why still synchronize in the consumer thread?