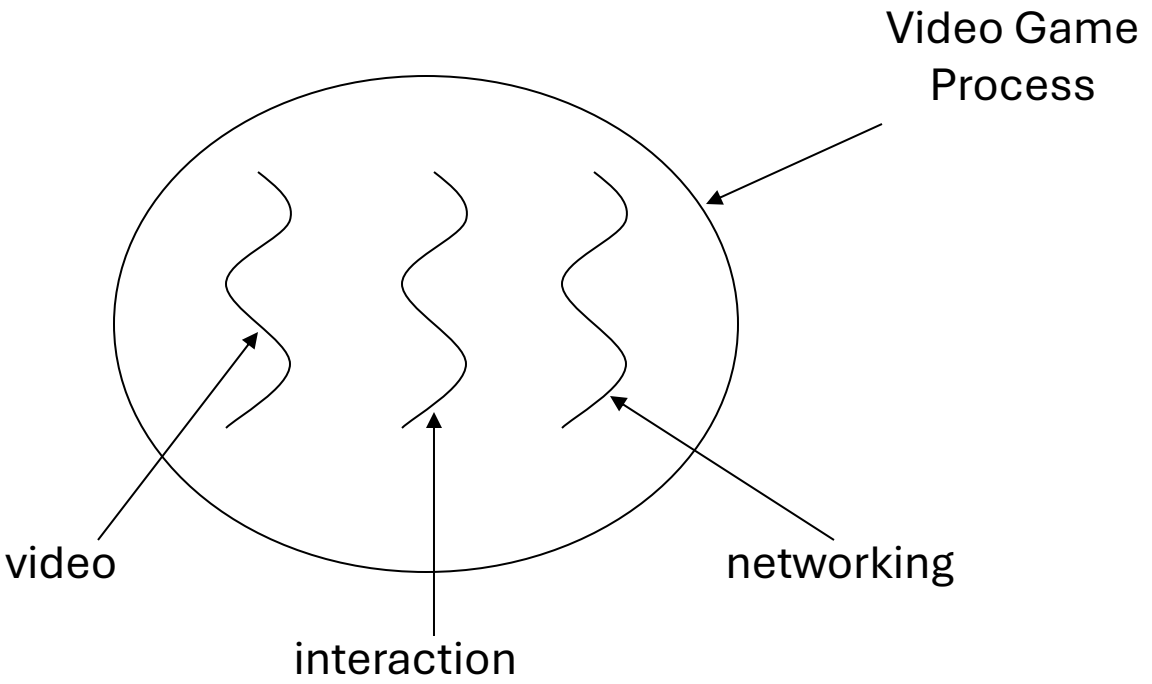
Threads

The slides adapted from different resources including:
Shivaram Venkataraman, CS 537 - Introduction to Operating Systems

What is a Thread?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking

What is a Thread?



Advantages

- easier to program
  - 1 thread per task
- can provide better performance
  - thread only runs when needed
  - no polling to decide what to do
- multiple threads can share resources
- utilize multiple processors if available

Disadvantage

- multiple threads can lead to deadlock
- overhead of switching between threads

Creating Threads (method 1)

- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call .*start()* to get the thread ready to run

Creating Threads Example 1

```
class Output extends Thread {
    private String toSay;
    public Output(String st) {
            toSay = st;
    }
    public void run() {
            try {
                    for(;;) {
                            System.out.println(toSay);
                            sleep(1000);
                    }
            } catch(InterruptedException e) {
                    System.out.println(e);
            }
    }
}
```

Example 1 (continued)

```
class Program {
    public static void main(String [] args) {
            Output thr1 = new Output(“Hello”);
            Output thr2 = new Output(“There”);
            thr1.start();
            thr2.start();
    }
}
```

- main thread is just another thread (happens to start first)
- main thread can end before the others do
- any thread can spawn more threads

Creating Threads (method 2)

- implementing Runnable interface
  - virtually identical to extending Thread class
  - must still define the *run()*method
  - setting up the threads is slightly different

Creating Threads Example 2

```
class Output implements Runnable {
    private String toSay;
    public Output(String st) {
            toSay = st;
    }
    public void run() {
            try {
                    for(;;) {
                            System.out.println(toSay);
                            Thread.sleep(1000);
                    }
            } catch(InterruptedException e) {
                    System.out.println(e);
            }
    }
}
```

Example 2 (continued)

```
class Program {
    public static void main(String [] args) {
            Output out1 = new Output(“Hello”);
            Output out2 = new Output(“There”);
            Thread thr1 = new Thread(out1);
            Thread thr2 = new Thread(out2);
            thr1.start();
            thr2.start();
    }
}
```

- main is a bit more complex
- everything else identical for the most part

Advantage of Using Runnable

- remember - can only extend one class
- implementing runnable allows class to extend something else

Controlling Java Threads

- _.*start()*: begins a thread running
- *wait()* and *notify()*: for synchronization
- _.*stop()*: kills a specific thread (deprecated)
- _.*suspend()* and *resume()*: deprecated
- _.*join()*: wait for specific thread to finish
- _.*setPriority()*: 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

GUI programming is multithreaded

Event-driven programming

- Event dispatch thread (EDT) handles all GUI events
  - Mouse events, keyboard events, timer events, etc.
- Program registers callbacks (“listeners”)
  - Function objects invoked in response to events

Ground rules for GUI programming

- All GUI activity is on event dispatch thread
- No other time-consuming activity on this thread
  - Blocking calls (e.g., I/O) absolutely forbidden
- Many GUI programs violate these rules
  - They are broken
- Violating rule 1 can cause safety failures
- Violating rule 2 can cause liveness failures

Ensuring all GUI activity is on EDT

- Never make a Swing call from any other thread
  - Swing calls includes Swing constructors
    - If not on EDT, make Swing calls with invokeLater:

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(() -> new Test().setVisible(true));
}
public void actionPerformed(ActionEvent e)
{
        new Thread(new Runnable()
        {
            final String text = readHugeFile();
            SwingUtilities.invokeLater(new Runnable()
            {
                public void run()
                {
                  textArea.setText(text);
                }
            });
      }).start();
  }
```

Callbacks execute on the EDT

- The SwingUtilities class has a static method available to use to put references to blocks of code onto the event queue:

```
public static void invokeAndWait(Runnable target)
throws InterruptedException, InvocationTargetException
```

- The parameter target is a reference to an instance of Runnable.
  - In this case, the Runnable will not be passed to the constructor of Thread.
  - The Runnable interface is simply being used as a means to identify the entry point for the event thread. Just as a newly spawned thread will invoke run(), the event thread will invoke run() when it has processed all the other events pending in the queue.
- An InterruptedException is thrown if the thread that called invokeAndWait() is interrupted before the block of code referred to by target completes.
- An InvocationTargetException (a class in the java.lang.reflect package) is thrown if an uncaught exception is thrown by the code inside run()

```
public static void invokeLater(Runnable target)
```

- it puts the request on the event queue and returns right away.
- The invokeLater() method does not wait for the block of code inside the Runnable referred to by target to execute.
  - This allows the thread that posted the request to move on to other activities.
- a new thread is not created when Runnable is used with SwingUtilities.invokeLater() and SwingUtilities.invokeAndWait()
  - The event thread will end up calling the `run()` method of the `Runnable` when its turn comes up on the event queue.

inovkeAndWait Example

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import javax.swing.*;

public class InvokeAndWaitDemo extends Object {
   private static void print(String msg) {
   String name = Thread.currentThread().getName();
   System.out.println(name + ": " + msg);
   }

   public static void main(String[] args) {
        final JLabel label = new JLabel("————");

        JPanel panel = new JPanel(new FlowLayout());
        panel.add(label);

        JFrame f = new JFrame("InvokeAndWaitDemo");
        f.setContentPane(panel);
        f.setSize(300, 100);
        f.setVisible(true);

try {
            print("sleeping for 3 seconds");
            Thread.sleep(3000);

            print("creating code block for event thread");
            Runnable setTextRun = new Runnable() {
                    public void run() {
                        print("about to do setText()");
                        label.setText("New text!");
                    }
                };

            print("about to invokeAndWait()");
            SwingUtilities.invokeAndWait(setTextRun);
            print("back from invokeAndWait()");
          } catch ( InterruptedException ix ) {
             print("interrupted while waiting on invokeAndWait()")
          } catch ( InvocationTargetException x ) {
             print("exception thrown from run()");
          }
        }
}
```

inovkeLater Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class InvokeLaterDemo extends Object {
    private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + msg);
    }

    public static void main(String[] args) {
        final JLabel label = new JLabel("————");
          JPanel panel = new JPanel(new FlowLayout());
          panel.add(label);
          JFrame f = new JFrame("InvokeLaterDemo");
          f.setContentPane(panel);
          f.setSize(300, 100);
          f.setVisible(true);
          try {
            print("sleeping for 3 seconds");
              Thread.sleep(3000);
        } catch ( InterruptedException ix ) {
              print("interrupted while sleeping");
        }
```

```
print("creating code block for event thread");
          Runnable setTextRun = new Runnable() {
                public void run() {
                    try {
                          Thread.sleep(100); // for emphasis
                          print("about to do setText()");
                          label.setText("New text!");
                    } catch ( Exception x ) {
                         x.printStackTrace();
                  }
                }
           };

          print("about to invokeLater()");
          SwingUtilities.invokeLater(setTextRun);
          print("back from invokeLater()");
    }
}
```

When invokeLater() Is Not Needed

- It is not always necessary to use invokeLater() to interact with Swing components.
- Any thread can safely interact with the components before they have been added to a visible container.
  - After the components have been drawn to the screen, only the event thread should make further changes to their appearance.
- There are a couple of exceptions to this restriction.
  - The adding and removing of event listeners can safely be done by any thread at any time.
  - any thread can invoke the repaint() method.
  - any method that explicitly indicates that it does not have to be called by the event thread is safe.
    - The API documentation for the setText() method of JTextComponent explicitly states that setText() can be safely called by any thread. The setText() method is inherited by JTextField (a subclass of JTextComponent), so any thread can safely invoke setText() on a JTextField component at any time. Threads and Swing
- If you aren't sure whether a particular method on a Swing component can be invoked by any thread, use the invokeLater() mechanism to be safe.

Callbacks execute on the EDT

- You are a guest on the Event Dispatch Thread!
- Don’t abuse the privilege
- If you do, liveness will suffer
  - Your program will become non-responsive
  - Your users will become angry
- If > a few ms of work to do, do it off the EDT
  - javax.swing.SwingWorker designed for this purpose

- Typical scenario:
  - long running task in a background thread
  - provide updates to the UI either when done, or **while processing**.

The following example illustrates the simplest use case. Some processing is done in the background and when done you update a Swing component.

Say we want to find the "Meaning of Life" and display the result in a JLabel.

```
final JLabel label;
class MeaningOfLifeFinder extends SwingWorker<String, Object> {
    @Override
    public String doInBackground() {
        return findTheMeaningOfLife();
    }

    @Override
    protected void done() {
        try {
            label.setText(get());
        } catch (Exception ignore) {
        }
    }
}

(new MeaningOfLifeFinder()).execute();
```