Wireless Online Game Development for Mobile Device

Lesson 5

What have we learnt last week?

- Introduction to Low-level API
- Display Image in Canvas
- Configure color, line style, font style for Drawing
- Drawing Line, Arc, Sector, Rectangle, Triangle and String by using the coordinate system.
- Deployment of your MIDP application
- How to clear the screen

Introduction to Thread

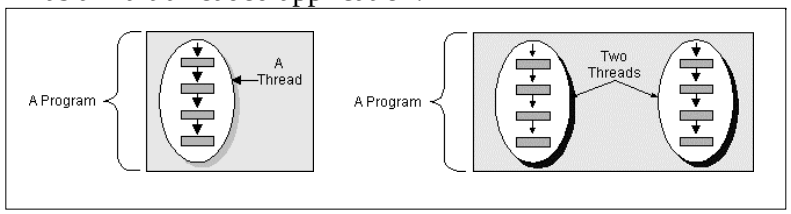
- One of the defining features of the Java environment is its built-in support for threads.
- Threads let an application perform multiple activities simultaneously.
- When used properly, threads let the application's user interface remain responsive while it performs lengthy operations like network communication or very complicated computations.
- While user interface responsiveness is important no matter what the platform, it's even more so on the handheld and consumer-oriented devices that the J2ME platform targets.

What is Thread?

- For a sequential programs,
  - It has a beginning, an execution sequence, and an end.
  - At any given time during the runtime of the program, there is a single point of execution.
- A thread is similar to the sequential programs.
  - A single thread also has a beginning, a sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution.
  - A thread itself is not a program; it cannot run on its own. Rather, it runs within a program.

What is Thread?

- A thread is a single sequential flow of control within a program
- Threads are the fundamental units of program execution. Every running application has at least one thread.
- An application consisting of two or more threads is known as a multithreaded application.



Context of Thread

- Each thread has a context associated with it.
- The context holds information about the thread, such as the address of the currently executing instruction and storage for local variables.
- The context is updated as the thread executes.
- The context also stores the thread's state.

State of Thread

- A thread can be in any of the following states:
  - A **Running Thread** is executing code.
  - A **Ready Thread** is ready to execute code.
  - A **Suspended Thread** is waiting on an external event (For example, it may be waiting for data to arrive from another device. After the event occurs, the thread transitions back to the ready state).
  - A **Terminated Thread** has finished executing code.
- A thread that is running, ready, or suspended is a live thread. A terminated thread is also known as a dead thread.

What is Thread Scheduling?

- Although an application may have many threads, the underlying device has a small, fixed number of processors available for code execution.
- Threads share these processors by taking turns being in the running state.
- This arrangement is called thread scheduling.

Thread Scheduling

- Thread scheduling is a complicated affair over which the application has little control.
- The system gives each ready thread a chance to run for a short time (the suspended threads are ignored), switching rapidly from one thread to another.
- This context switching can occur at any time.
- The only real influence the application has over thread scheduling is via the thread's priority level: higher-priority threads execute more often than those with lower priority.

Implementation of Thread

- There are two ways to create a new thread of execution.
  - One is to declare a class to be a subclass of Thread.
    - This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started.
  - The other way to create a thread is to declare a class that implements the Runnable interface.
    - That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

1) Subclass of Thread

- For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

- The following code create a thread and start it running

```
PrimeThread p = new PrimeThread(143);
p.start();
```


2) Implementation of Runnable

- For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

- The following code create a thread and start it running

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

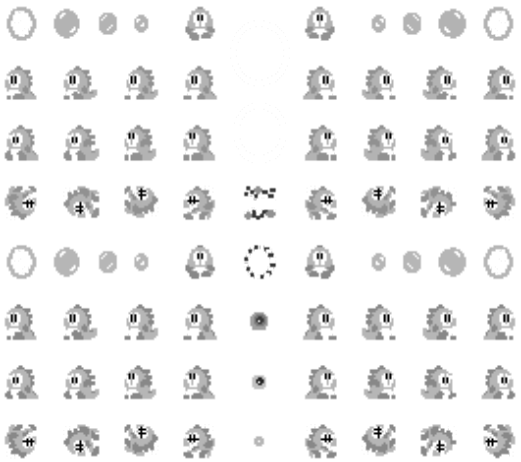
Animation? Moving Pictures?

- The idea of moving images developed in 65 B.C. with the theory of persistence of motion, but it took more than another 1900 years before someone photographed a moving scene.
- Animation is the simulation of movement through a series of pictures that have objects in slightly different positions.

How to Display Animation?

- For animation to work, the pictures making up the animation must replace one another quickly enough to trick the human eye into believing there is movement.
- A replacement rate of at least 14 frames per second or faster accomplishes this sense of movement

Source of Animation



Implementation of Animation

- When you perform animations, you can first calculate the display content and then call ***repaint*()** to paint the new frame.
- In order to control the repaint time slot, we can use ***Sleep()***.

Event Handling

- When an event occurs on a mobile device, a Command object holds information about that event.
- This information includes the type of command executed, the label of the command, and its priority.
- In J2ME, commands are commonly represented with soft-buttons on the device.

Command Events

- We process Canvas Commands the same way we would those on any other component:
  - Create an instance of a Command object.
  - Add the Command to Canvas
  - Create a Command listener to capture events.
  - Write a commandAction() method to process events.

Implement CommandListener

```
class ImageCanvas extends Canvas implements CommandListener {
    private Command cmdExit;
    ...
    display = Display.getDisplay(this);
    cmdExit = new Command("Exit", Command.EXIT, 0);
    addCommand(cmdExit);
    setCommandListener(this);
    ...
    protected void paint(Graphics g) {
        // Draw onto the canvas
        ...
    }
    public void commandAction(Command c, Displayable d) {
        if (c == cmdExit)
        ...
    }
}
```


Capture Key Press

- In addition to soft-keys for processing commands, a Canvas object has access to 12 keycodes.
- It can handle ITU-T codes for 0..9, * and # only
- The codes that are guaranteed to be available on any MIDP device are shown below:

KEY_NUM0	KEY_NUM1	KEY_NUM2
KEY_NUM3	KEY_NUM4	KEY_NUM5
KEY_NUM6	KEY_NUM7	KEY_NUM8
KEY_NUM9	KEY_STAR	KEY_POUND

Key Codes Handling

- The five methods for working with keycodes are as follows:
  - void keyPressed (int keyCode)
  - void keyReleased (int keyCode)
  - void keyRepeated (int keyCode)
  - boolean hasRepeatEvents( )
  - String getKeyName (int keyCode)

Game Key Capture

- LEFT, RIGHT, UP, DOWN and FIRE are game key codes, that need to use ***getGameAction*** to handle
- Movement should be done in a thread
- Experience from J2ME™ MIDP for Palm:
  - Don't implement touchable key-emulation because it is already implemented.
  - Holding the "key" does not have the same impact on a palm devices as in mobile phones

Game Actions Handling

- The values of the game action are mapped to the device's directional arrows, but not all mobile devices have these.
- If a device lacks directional arrows, game actions will be mapped to the keypad (e.g., LEFT mapped to 2, RIGHT mapped to 5, etc…).
- The following constants have been defined for handling game-related events in MIDP:

UP	GAME_A
DOWN	GAME_B
LEFT	GAME_C
RIGHT	GAME_D
FIRE	

Detecting Game Actions

- The following code sample shows one way to detect various game actions and call the appropriate methods based on the action selected.

```
protected void keyPressed(int keyCode) {
   switch (getGameAction(keyCode)) {
      case Canvas.FIRE:
        shoot();
        break;
      case Canvas.RIGHT:
        goRight();
        break;
      ...
   }
}
```

Detecting Game Actions

- Another option would be to create a reference for each game action during initialization, then look for the appropriate key at runtime, as shown below:

```
// Initialization
keyFire = getKeyCode(FIRE);
keyRight = getKeyCode(RIGHT);
keyLeft = getKeyCode(LEFT);
...
// Runtime
protected void keyPressed(int keyCode) {
   if (keyCode == keyFire)
      shoot();
   else if (keyCode == keyRight)
      goRight()
      ...
}
```


Game Flow Control

- We can declare a variable to control such as "GameScreen" the game flow.
  - 0 – Opening
  - 1 – Game Playing
  - 2 – Drop to the Hole
  - …
  - 9 – Game Over

Add an Opening Screen

- To add an opening screen for the shooting game, you must load the image and display it when the game start. The simple method is adding the following code in the "Paint" class.

```
public void paint(Graphics g) {
  try {
    imgOpening = Image.createImage("/opening.png");
    g.drawImage(imgOpening, 0, 0, g.TOP|g.LEFT);
  }
}
```


Start the Game

- After adding the opening screen, then need to start the thread to handle the event of "Start Game" when user press the [Start] button.

```
public void commandAction(Command c, Displayable s) {
  if (c == mStartCommand)
    MyThread.start();
}
```

Game Over Screen

- As mentioned above, “GameScreen” variable can be used to control the game flow. Therefore, we can add our event in the logic condition “GameScreen == 9”

Movement of the Ground

- To simulate the movement, you can set the ground move from right to left.