An A to Z preliminary guide for a game developer

Deprecated: This article contains content which comes from an old version of the developer's library. See instead:
- Game API
- Game Design
- UI Graphics and Interaction

Introduction

This document covers:

- Goals of J2ME Games
- Recommended practices for developing games.
- Game API in MIDP 2.0
- Essential classes required for MIDP 1.0 and CLDC 1.0
- Code Optimization

Goals for J2ME Games

Let us review general goals that are necessary to be understood while developing J2ME Games.

- Must support a broad variety of devices
- Must ensure that content is lightweight, well tested on most devices
- Must ensure that the games are easy to play and have a user friendly interface
- Additional effects such as sound and vibration can be added based on the gaming requirement Multi-player games should provide smooth join and exit or pauses of other players during the game-play and hence should not affect the game of any player.
- Develop the game in such a way that porting to be easy.

Recommendations

Full Screen - Most screens are best played when in full screen. It is suggested that the game be initiated and played in full screen through out the game play.

Sound - All games are recommended to have some sound at least using tones. Specialized games developed exclusively for phones such as N-Gage and N-Gage QD can have .wav based sounds.

Game Pre-loading: It is suggested that the heavy classes, images, and entire layout be pre-loaded with a loading screen as soon as the user clicks on the game option from the games menu. Also the loading screen can have a status bar that shows how much has been loaded.

Game Menu: It is recommended that the game include a proper gaming menu which can allow users to choose various options such as New Game, Save Game etc.

Use Double Linked Lists instead of vectors: It is recommended that the developer use double linked lists instead of vectors to manage objects instead of vectors as they are fast and easy to navigate. Please refer to book "J2ME Game Programming" as to how we can use double linked lists.

Draw state-ful objects that are only within the visible context of the layout: It is recommended that the objects within the layer-manager be drawn only when they fall within the visible context of the screen. By state-ful, these objects could be something that needs itself to remember its state or some data through out the game. For example could be several computer racers within the game, these objects need to remember their X,Y positions and hence they generally need to be kept alive i.e. not destroyed. But they can be drawn on the Layout-Manager (Just in Time)(JIT).

Create Objects JIT: If conditions permit, some state-ful objects can be destroyed with their state being maintained by other means. Then these objects can also be created Just in Time (JIT) when they begin to appear within the visible context of the screen.

Pre-load Images, Sounds: Loading sounds or images from files generally takes a long time. Hence it is suggested that all images and sounds be preloaded within the constructor.

Follow Coding Conventions such as a) Naming Conventions - Hungarian naming conventions b) Code comments including IPR Statement c) Indent your code

Follow good programming practices a) Create objects JIT / variables JIT b) Reduce the number of variables that are publicly defined and have a global scope

c) Do not use cyclic functions

d) Reduce redundant code by introducing functions e) Remove unnecessary code & variables

Follow good re-usability practices a) Create Classes that can enable you easy re-use. They should have minimum customization required b) If many of the libraries are commonly used, create them as runtime objects and re-use c) Functionality specific classes can be made generic during game development and later re-used. Like for e.g. car movements made generic and car class is made. Later when any car game is developed that class can be re-used.

Follow the Game Development Life-cycle (GDLC)

The game development life-cycle forms an important part of game development as it defines the process well and introduces a better and refined game development approach also called as the game development process.

Additions to the Game Development Recommendation Guidelines

These additions are added with the experience gained from the past games.

- Splash screen - Must have a clear Logo of the game, distinct from the splash screen. Must occupy full screen and must include a startup tone based music
- Text box for name - Please remove this as it is quite troublesome. To save the game, request for text box after the game completes for high scores.
- Quick restart - Show a “Restart” button when you show the game over screen. This will enable a quicker restart. Also show a “Quit” button on right to enable the users to leave the game.
- Game loading - Center this screen, and really load the game midlet / canvas class within it. Also load any other objects class within it. Then just hide their screen.
- Sound - Sound must be a compulsory item to all games, tone based or midi based.
- Versioning -Version logging of games should begin from now in order to keep track of the game. All games will have a “strVersion” string which will be updated every time a game is “Released”. Versioning should be maintained as given below.

<GAME MAJOR VERSION NO> <RELEASE NO> . <COMPILATION VERSION NO> . <MIDP VER IF APPLICABLE>

where • GAME MAJOR VERSION relates to the storyline of the game. If the storyline has changed and if there has been a previous game then there must be an increment in major version. • RELEASE NO is the number to be allocated by Ali for each game released for publishing. For every release (for publishing) this must be incremented. • COMPILATION VERSION NO is the no to be incremented for each compilation. • MIDP VER is the midp version if applicable.

Sample

My Game 1.1.24.2

This indicates,

First version First release 24th compilation MIDP 2.0

This information is for statistical purposes only and is not a performance indicator.

- Centralized storage of all games: The game team lead or senior members of the team must have access to a repository within the office and store all game source code / binary code. For each game (complete / incomplete) must be stored within a respectively named folder with different versions within it. New backup must be taken within the same folder for every new “RELEASE NO” only.
- In case of troublesome bugs refer to the centralized bug repository.
- Have colorful, graphical menus instead of text based menus.
- Have a dynamic splash screen.

The Game API

The Game API package provides a series of classes that enable the development of rich gaming content for wireless devices. Wireless devices have minimal processing power, so much of the API is intended to improve performance by minimizing the amount of work done in Java; this approach also has the added benefit of reducing application size. The API's are structured to provide considerable freedom when implementing them, thereby permitting the extensive use of native code, hardware acceleration and device-specific image data formats as needed. The API uses the standard low-level graphics classes from MIDP (Graphics, Image, etc.) so that the high-level Game API classes can be used in conjunction with graphics primitives. For example, it would be possible to render a complex background using the Game API and then render something on top of it using graphics primitives such as drawLine, etc. The Game API can be found in javax.microedition.lcdui.game. There are five new classes in the API and they are GameCanvas, Layer, LayerManager, Sprite, and TiledLayer. The GameCanvas is an abstract class that provides the basis for the game user interface. The class has two benefits over the Canvas class; it has an off-screen buffer and it has functionality to get the states of the physical keys of the device. The Layer is an abstract class that represents an element in the game. Sprite and TiledLayer are inherited from Layer. Layer is mostly used by its subclasses. The LayerManager manages several Layer objects and draws them on the screen in specific order. The Sprite is a Layer that may contain several frames stored in an Image. One Image contains e.g. four images of walking dog. With Sprite we can use parts of the image as frames and make a sequence of frames to create motion etc. The Sprite also has functionality for checking collisions between other Sprites or TiledLayers. The TiledLayer is a little like Sprite but it is mainly meant for backgrounds, roads or other larger areas. TiledLayer consists of a grid of cells, which can be filled with images or tiles. So the background or scenery is built of small images. Handling user input With MIDP 2.0, handling the user input can be done a bit differently than in MIDP 1.0. In 1.0 you had to use the getGameAction() method of Canvas to get the game action. In 2.0 you can get the state of the keys by using the getKeyStates() method of GameCanvas. The following code shows how it can be done. First we get the key states into keyState and then we check (with bitwise operation) all four keys (up, down, left, and right) and do what we want to do.

```
protected void keyPressed(int keyCode) {
 int move = 0;

  int keyState = getKeyStates();
  if ((keyState &amp;amp; LEFT_PRESSED)&amp;nbsp;!= 0) {
  // do something
  }

  if ((keyState &amp;amp; RIGHT_PRESSED)&amp;nbsp;!= 0) {
  // do something
  }

  if ((keyState &amp;amp; UP_PRESSED)&amp;nbsp;!= 0) {
  // do something
  }
```

```
  if ((keyState &amp;amp; DOWN_PRESSED)&amp;nbsp;!= 0) {
  // do something
  }

}
```

Using the off-screen buffer The off-screen buffer makes it easier to create non-flickering animation (or movement) and developers don't have to use an extra Graphics object to create double buffering. The idea of the off-screen buffer in GameCanvas is to provide an off-screen Graphics object that can be used to draw pixels (or layers or sprites) to the canvas. When everything is drawn the whole thing can be flushed at once. In the following code the getGraphics() method of GameCanvas is used to get the off-screen buffer. In the while loop the buffer is used to paint the contents of the LayerManager (layers object). Then the buffer is flushed with flushGraphics() method. There is also flushGraphics(int x, int y, int width, int height) method, which only flushes the specified region of the off-screen buffer to the display.

```
public void run() {

 Graphics g = getGraphics();

  while (play) {
  try {
  // First draw all the layers
  layers.paint(g, 0, 0);

  // If the game is on, flush the graphics
  if (play) {
  flushGraphics();
  }

  try {
  mythread.sleep(sleepTime);
  } catch (java.lang.InterruptedException e) {}

  } catch (Exception e) {
  e.printStackTrace();
  }
  }

}
```

Using layers In a game (or in any graphical application) the screen usually consists of different things, which either are connected to each other or not. For example a bee could fly over the forest, land, water and air, but in a puzzle the little guy shouldn't be able to walk through walls. The MIDP 2.0 Game API introduces layers, which provide a way to manage objects or contexts on the screen. Layers can be either TiledLayer (e.g. background, walls), Sprite (e.g. battleship, bee), or self-made class inherited from Layer. The code in the following example is gathered from different parts of an application to illustrate the usage of layers. The most important classes in this example are TiledLayer, LayerManager, and Image. Image is a class that holds several equally sized images, tiles. The TiledLayer uses these images by placing them in a grid.

When an instance of TiledLayer is created, the constructor requires five parameters. These parameters are the number of columns and rows, the image, and width and height of a tile. In this case the grid has 40 columns, 16 rows, the image is Tiles.png and the width and the height of a tile is 7 pixels. The last integers (TILE_GROUND etc.) in the beginning of the code are references to certain tiles. After the instance of the TiledLayer is created, the cells of the grid can be filled with fillCells() (which fills a region) or fillCell() (which fills one cell). On the final line of the code the TiledLayer is added to the LayerManager. The append() method puts the layer furthest from the viewer. With insert() you can insert the layer at any place you want.

```
private TiledLayer tiles;
private LayerManager layers;
private Image tilesImage;

public final int TILE_GROUND = 1; public final int TILE_WATER = 2; public final int TILE_SHORE_LEFT = 3; public final int TILE_FORES

// ...

// Creating an instance of the TiledLayer layers = new LayerManager(); try {

 tilesImage = Image.createImage("/Tiles.png");

} catch (IOException e) {} tiles = new TiledLayer(40, 16, tilesImage, 7, 7);

// ...

// Filling the TiledLayer with tiles tiles.fillCells(0, 0, 40, 16, TILE_AIR); tiles.fillCells(14, 12, 12, 4, TILE_WATER); tiles.fill

layers.append(tiles);
```

Using sprites As said before, sprites represent a single object on screen. The object can be a little man pushing stones, a battleship shooting enemies, etc. The Sprite class works a bit like TilesLayer (after all they are both inherited from Layer). The Sprite also has an image that holds one or several equally sized images. However these images are not tiles, they are frames for animation. The sprite can be an animated sprite and the animation is made by changing the frame. The following code shows how an instance of Sprite is created. The principle is exactly the same as with TiledLayer.

```
try {

 spriteImage = Image.createImage("/Sprite.png");

} catch (IOException e) {}

sprite = new Sprite(spriteImage, 7, 7);
```

The sprite can be moved in the game area with two simple methods in Layer class:

```
move(int dx, int dy)
setPositions(int x, int y)
```

It is possible to let the LayerManager handle the drawing and moving the sprite, and it is possible to store the location of the sprite and set the place in the code. The following code shows how a sprite is moved.

The image of the sprite is 7 pixels in width and height and so the sprite is moved 7 pixels at a time.

```
public static final int UP = 0;
public static final int RIGHT = 1;
public static final int DOWN = 2;
public static final int LEFT = 3;

// ...

switch (direction){

 case UP:
  sprite.move(0, -7);
  break;
  case DOWN:
  sprite.move(0, 7);
  break;
  case RIGHT:
  sprite.move(7, 0);
  break;
  case LEFT:
  sprite.move(-7, 0);
  break;
  default: break;

}
```

Another important thing in game programming is detecting collisions. The sprites might have to stay in the game area or in a specific puzzle or labyrinth, and it's also important to detect if the sprites collide with each other. Collision in another game could just mean change of direction and in another it could mean the end of the game. Sprite has four methods that help developers with collisions:

```
collidesWith(Image image, int x, int y,boolean pixelLevel)
collidesWith(Sprite s, boolean pixelLevel)
collidesWith(TiledLayer t, boolean pixelLevel)
defineCollisionRectangle(int x, int y,int width, int height)
```

It is possible to detect collisions between two instances of Sprite, a Sprite and a TiledLayer, and a Sprite and an Image.

Essential classes required for MIDP 1.0 and CLDC 1.0

The above mentioned GAME API is not present in MIDP 1.0, so we have to write our own Tiled layer and Sprite classes. Also, for CLDC 1.0, Math calculations can be accomplished only if we have our own defined Math class. We can see how these can be achieved. Sprite Inorder to achieve Sprite transition in MIDP 1.0, we have to get a sprite image from the graphic designing team and to write our own code to make the functionality. A sprite image looks as this,

Here below, I include one Sprite class that I used for one of my games,

```
public class Sprite {
//General variables for Sprite attributes

 Image sourceImage;
  int numberOfFrames;
  int srcFrameWidth;
  int srcFrameHeight;
  int frameRows;
  int xRef,yRef;
  int frameColumns;
  private int sequenceIndex;


//Constructor

 public Sprite(Image image) {
  if (image == null) {
  throw new NullPointerException("demo");
  } else {
  sourceImage = image;
  numberOfFrames = 1;
  srcFrameWidth = image.getWidth();
  srcFrameHeight = image.getHeight();
  sequenceIndex = 0;
  frameRows = 1;
  frameColumns = 1;
  }
  }


//Setting the initial pointer to one particular frame of the sprite image

 public void setPosition(int x,int y) {
  xRef=x;
  yRef=y;
  }


//For checking the Sprite frame collision

 public boolean collidesWith(Sprite s) {
  return collide(this.xRef, this.yRef, this.srcFrameWidth, this.srcFrameHeight,
  s.xRef, s.yRef , s.srcFrameWidth, s.srcFrameHeight );
  }
  private static boolean collide(int x1, int y1, int w1, int h1,
  int x2, int y2, int w2, int h2) {
  return hit(x1, w1, x2, w2) &amp;&amp; hit(y1, h1, y2, h2);
  }
  private static boolean hit(int p1, int l1, int p2, int l2) {
  return (p1 &lt;= p2) ? ( (p1 + l1) &gt;= p2) : (p1 &lt;= (p2 + l2));
  }


//Returns the X-value

 public int getX() {
  return xRef;
  }


//Returns the Y-value
```

```
 public int getY() {
  return yRef;
  }

  public Sprite(Image image, int frameWidth, int frameHeight) {
  if (image == null) {
  throw new NullPointerException("Source Image is Null");
  } else if (frameWidth == 0) {
  throw new IllegalArgumentException("Invalid FRAME WIDTH");
  } else if (frameHeight == 0) {
  throw new IllegalArgumentException("Invalid FRAME Height");
  } else if (image.getWidth()&amp;nbsp;% frameWidth&amp;nbsp;!= 0) {
  throw new IllegalArgumentException(
  "IMAGE WIDTH is not an integer multiple of the Frame Width");
  } else if (image.getHeight()&amp;nbsp;% frameHeight&amp;nbsp;!= 0) {
  throw new IllegalArgumentException(
  "IMAGE HIGHT is not an integer multiple of the Frame Height");
  } else {
  sourceImage = image;
  srcFrameWidth = frameWidth;
  srcFrameHeight = frameHeight;
  frameRows = image.getHeight() / frameHeight;
  frameColumns = image.getWidth() / frameWidth;
  sequenceIndex = 0;
  numberOfFrames = frameColumns * frameRows;
  }
  }

  public Sprite(Sprite s) {
  sourceImage = s.sourceImage;
  numberOfFrames = s.numberOfFrames;
  srcFrameWidth = s.srcFrameWidth;
  srcFrameHeight = s.srcFrameHeight;
  sequenceIndex = s.getFrame();
  frameRows = s.frameRows;
  frameColumns = s.frameColumns;
  }


 public void setFrame(int sequenceIndex) {
  if (sequenceIndex &lt; 0 || sequenceIndex &gt;= numberOfFrames) {
  throw new IllegalArgumentException("Invalid Sequence Index");
  } else {
  this.sequenceIndex = sequenceIndex;
  }
  }

  public final int getFrame() {
  return sequenceIndex;
  }

  public int getFrameSequenceLength() {
  return numberOfFrames;
  }

  public void nextFrame() {
  sequenceIndex = (++sequenceIndex)&amp;nbsp;% numberOfFrames;
  }

  public void prevFrame() {
  sequenceIndex = (--sequenceIndex + numberOfFrames)&amp;nbsp;% numberOfFrames;
  }

  public final void paint(Graphics g, int xRef, int yRef) {
  if(yRef&gt;=0 &amp;amp;&amp;amp; yRef&lt;=130) {
  g.setClip(xRef, yRef, srcFrameWidth, srcFrameHeight);
  g.drawImage(sourceImage,
  xRef - (sequenceIndex&amp;nbsp;% frameColumns) * srcFrameWidth,
  yRef - (sequenceIndex / frameColumns) * srcFrameHeight,
  Graphics.TOP | Graphics.LEFT);
  }
  g.setClip(0,0, 128,128);
  g.clipRect(0,0, 128,128);
  }
  public final void paint(Graphics g) {
  g.setClip(xRef, yRef, srcFrameWidth, srcFrameHeight);
  g.drawImage(sourceImage,
  xRef - (sequenceIndex&amp;nbsp;% frameColumns) * srcFrameWidth,
  yRef - (sequenceIndex / frameColumns) * srcFrameHeight,
  Graphics.TOP | Graphics.LEFT);
  g.setClip(0,0, 128,128);
  g.clipRect(0,0, 128,128);
  }
  public final void paint(Graphics g,int Y) { //boolean Move) {
  int temp_y=yRef-Y;

  if(yRef&gt;=0 &amp;amp;&amp;amp; temp_y&lt;=130) {
  g.setClip(xRef, temp_y, srcFrameWidth, srcFrameHeight);
  g.drawImage(sourceImage,
  xRef - (sequenceIndex&amp;nbsp;% frameColumns) * srcFrameWidth,
  temp_y - (sequenceIndex / frameColumns) * srcFrameHeight,
  Graphics.TOP | Graphics.LEFT);
  }
  g.setClip(0,0, 128,128);
  g.clipRect(0,0, 128,128);
  }

}
```

Tiled Layer

Similarly We have to implement the Tiled Layer class also since it is not available with MIDP 1.0. Here I include sample code for Tiled Layer,

```
public class TileLayer {

 private Image source;
  private int cols,rows,tileHeight,tileWidth;

 private int xpos,ypos;
  private int noofcols;

 private byte cellArrray[][];

<br>
```

```
 public TileLayer(int cols,int rows,Image source,int tileWidth,int tileHeight) {
  this.cols=cols;
  this.rows=rows;
  this.source=source;
  this.tileHeight=tileHeight;
  this.tileWidth=tileWidth;
  cellArrray=new byte[cols][rows];
  noofcols=source.getWidth()/tileWidth;
  }

 public int getCell(int col,int row) {
  try {
  return cellArrray[col][row];
  }catch(ArrayIndexOutOfBoundsException e){
  return -1;
  }
  }

 public void setCell(int col,int row,int tileindex) {
  cellArrray[col][row]=(byte)tileindex;
  }

 public void setPosition(int xpos,int ypos) {
  this.xpos=xpos;
  this.ypos=ypos;
  }

  public void paint(Graphics g,int col,int row,int yp) {
  int x,y;
  int endi,endj;
  endi = col+Math.abs(128) / tileWidth;
  endj = row+Math.abs(145) / tileHeight;

 x=0;
  y=yp%29;

  if(y&gt;=2 &amp;&amp; endj&lt;519) {
  endj+=1;
  }
  y=-1*y;

  for(int i=row;i&lt;endj;i++) {
  for(int j=col;j&lt;endi;j++) {
  g.setClip(x,y,tileWidth,tileHeight);
  g.clipRect(x,y,tileWidth,tileHeight);
  g.drawImage(source,x-(cellArrray[j][i]%noofcols)*tileWidth,y-(cellArrray[j][i]/noofcols)*tileHeight, g.TOP|g.LEFT);
  x=x+tileWidth;
  }
  y=y+tileHeight;
  x=0;
  }
  g.setClip(0,0,128,128);
  g.clipRect(0,0,128,128);
  }

}
```

Math Class

While developing the games for CLDC 1.0 devices, we have to write a utility class for performing the math operations such as sin,cos,tan. See the below mentioned class,

```
class Math

 {
  private static final short sinValue[] =
  {
  0, 175, 349, 523, 698, 872, 1045, 1219, 1392, 1564, 1736, 1908, 2079,
  2250, 2419, 2588, 2756, 2924, 3090, 3256, 3420, 3584, 3746, 3907, 4067,
  4226, 4384, 4540, 4695, 4848, 5000, 5150, 5299, 5446, 5592, 5736, 5878,
  6018, 6157, 6293, 6428, 6561, 6691, 6820, 6947, 7071, 7193, 7314, 7431,
  7547, 7660, 7771, 7880, 7986, 8090, 8192, 8290, 8387, 8480, 8572, 8660,
  8746, 8829, 8910, 8988, 9063, 9135, 9205, 9272, 9336, 9397, 9455, 9511,
  9563, 9613, 9659, 9703, 9744, 9781, 9816, 9848, 9877, 9903, 9925, 9945,
  9962, 9976, 9986, 9994, 9998, 10000
  };

 public static short sin(short degree)
  {
  short angle=(short)((360+(degree% 360))%360);
  return (angle&gt;=0 &amp;&amp; angle &lt;= 90)?sinValue[angle]:(angle&gt;90 &amp;&amp; angle&lt;=180)
   ?sinValue[180-angle]:(angle&gt;180 &amp;&amp; angle&lt;=270)?(short)((-1)*sinValue[angle-180]):(short)((-1)*sinValue[360-ang
  }

 public static short cos(short degree)
  {
  return sin((short)((degree % 360) - 90));
  }

}
```

See the first array, I write the the real the sign value of angle 0 to angle 90 as signvalue*1000 to avoid floating point. And when ever I want to calculate sin value, I get the value from this array and divide it by 10000. Eg. If I want to get the sin(30), I get the sinValue[30] from the array and divide it by 1000.

```
sinValue[30] = 5000;
```

If I divide this by 10000, I get the real sign value, i.e 0.5.

Note: here I multiplied by 10000 and retrieved, instead you multiply it by powers of 2, you can perform left shift operation.

If I multiply the sin value by 16384(2^14), I can get the real value by 2^14. It would be more accurate.

Code Optimization

This section I wrote by referring Sun’s code optimization steps and added some more additions for a J2ME game developer. Java 2 Micro Edition (J2ME) is a stripped-down version of Java, suitable for small devices with limited capabilities, such as cell phones and PDAs. J2ME devices have: • limited input capabilities (no keyboard!) • small display sizes • restricted storage memory and heap sizes • slow CPUs Writing fast games for the J2ME platform further challenges developers to write code that will perform on CPUs far slower than those found on desktop computers. When Not to Optimize: J2ME developers are all too familiar with the challenges of keeping their JAR as small as possible. Here are a few more reasons not to optimize: • optimization is a good way to introduce bugs • some techniques decrease the portability of your code • you can expend a lot of effort for little or no results • optimization is hard Use of Optimization: Optimization can of course, mean many things, but in our context we'll define it as designing and developing MIDlets such that they can efficiently exist in the constrained environments of limited devices while placing minimal strain on the available resources. We can distinguish between different types of optimization: • Optimizing for speed • Optimizing for maintainability • Optimizing for size

Some Examples:

1. Unroll Loops to Optimize Your Code

A compiler can automatically optimize the code by unrolling loops. Consider this code:

```
 int buff = new int[3];
  for (int i =0; i&lt;3; i++)
 {
  buff[i] = 0;
 }
```

On every iteration, the loop assigns a value to the next array element. However, precious CPU time is also wasted on testing and incrementing the loop counter, and performing a jump statement. To avoid this overhead, the compiler can unroll the loop into a sequence of three assignment statements:

```
 buff[0] = 0;
  buff[1] = 0;
  buff[2] = 0;
```

This way, you avoid the unnecessary overhead of a loop. Note, however, that the compiler applies this optimization automatically; you shouldn't do it yourself.

2. Use return statements out of switch blocks The break statements are necessary in each branch in a switch statement because otherwise the flow of execution "falls through" to the next case.

```
switch (suit) {

case CLUBS: {

----

----

stmts; }
return "Clubs";

case DIAMONDS: {

----

----

stmts; }
 return "Diamonds";

<br>

<br>

case HEARTS: {

----

 -------------
  stmts;
  }

return "Hearts";

 case SPADES:
  {
  -------------
  -------------
  stmts;
  }
  return "Spades";

 default:
  {
  -------------
  -------------
  stmts;
  }
  return "Not a valid suit";
  }
```

return statement saves 3 bytes. In this case we don't need break statements because the return statements cause the flow of execution to return to the caller instead of falling through to the next case. In general it is good style to include a default case in every switch statement, to handle errors or unexpected values

3. Eliminate Common Sub expressions: Before(Eliminate):

```
double x = d * (lim/max) * sx;
double y = d * (lim/max) * sy;
```

After (Eliminate):

```
double depth = d * (lim/max);
double x = depth * sx;
double y = depth * sy;
```

4. Don't intialize if you don't have to private static final boolean isVisible = false; It will cost 23 bytes. Simply private static final boolean isVisible;

5. Avoid String constants

System.out.println statements as they are bulky and apart from testing purposes aren't going to do much on a cell phone. For Ex:

```
System.out.println(“Collide”);
System.out.println (“intBallX”+ intBallX);
```

6. Division is slower than multiplication, so multiply by the inverse instead of dividing. There are also some optimizations you can perform when using a Fixed Point math library. First, if you're doing a lot of division by a single number, you should instead work out the inverse of that number and perform a multiplication. Multiplication is slightly quicker than division. So instead of...

```
 int fpP = FP.Div( fpX, fpD );
  int fpQ = FP.Div( fpY, fpD );
  int fpR = FP.Div( fpZ, fpD );
```

...you should rewrite it like this:

```
 int fpID = FP.Div( 1, fpD);
  int fpP = FP.Mul( fpX, fpID );
  int fpQ = FP.Mul( fpY, fpID );
  int fpR = FP.Mul( fpZ, fpID );
```

```
If you're performing hundreds of divisions every frame, this will help.
```

7. Try to avoid typecasting. Use type integers instead. For example, you may find that the multiplication method has to cast both ints to longs and then back to an int:

```
Public static final int Mul (int x, int y) {

 long z = (long) x * (long) y;
  return ((int) (z &gt;&gt; 16));
}
```

Those casts take time. Collision detection using bounding circles or spheres involves adding the squares of ints together. That can generate some big numbers that might overflow the upper bound of your int Fixed Point data type. To avoid this, you could write your own square function that returns a long:

```
 public static final long Sqr (int x) {
  long z = (long) x;
  z *= z;
  return (z &gt;&gt; 16);

}
```

This optimized method avoids a couple of casts 8. Carefully evaluate coding alternatives A switch statement is an alternative to a chained conditional that is syntactically prettier and often more efficient. It looks like this:

```
 switch (symbol) {
  case '+':
  perform_addition ();
  break;
  case '*':
  perform_multiplication ();
  break;
 default:
  System.out.println("I only know how to perform addition and multiplication");
  break;
  }
```

This switch statement is equivalent to the following chained conditional:

```
 if (symbol == '+') {
  perform_addition ();
  } else if (symbol == '*') {
  perform_multiplication ();
  } else {
 System.out.println("I only know how to perform addition and multiplication");
 }
```

It can reduce 40 bytes.

9. Pre-Compute Values For Example: Pre-Compute the values like getWidth() of the screen width and getHeight() of the screen height., etc., and examples like SQRT formulaes are useful if it is pre-computed.

10. Strings create garbage and garbage is bad so use StringBuffers instead. One common way to avoid creating unnecessary objects is to use

StringBuffers instead of Strings. For example, instead of:

```
String tempString="";

iceCreamGroup x=new IceCreamGroup();

 for (int i=0;i&lt;someNumber;i++) {

 tempString+=x.getIceCream (i);
  }
```

Use a StringBuffer instead:

```
 StringBuffer buff=new StringBuffer();

 iceCreamGroup x=new IceCreamGroup();

 for (int i=0;i&lt;someNumber;i++) {

 buff.append (x.getIceCream (i));
  }
```

Another way in J2EE to reuse objects is to make use of so-called object pools, when the creation of "expensive" objects is kept to a minimum by storing already-created objects in pools, where they can be reinitialized and reused when needed. Note though, that caching objects when they aren't always needed may actually cause problems in J2ME and lead to the dreaded Out Of memory error. When joining couple of Strings use StringBuffer instead of String.

Instead of writing

```
 String str= "Welcome"+ "to" + "our" + "site";
```

Write it as

```
StringBuffer sb = new StringBuffer(50);
sb.append("Welcome");
sb.append("To");
sb.append("our");
sb.append("site");
```

If you know the Max Length of the String Buffer initially then use the number. For Example in the above scenario, the initial length of the StringBuffer is set to 50. This saves lot of over head since when length of the StringBuffer needs to be increased, then StringBuffer has to allocate new Character array with larger capacity, copy all the old contents into new array and discard the old array (during GC). To avoid this over head declare the approximate size of StringBuffer initially. Also don't over allocate. Memory is precious at runtime.

11. Optimize loops by removing unnecessary evaluations. There are several code optimization techniques that will help speed up your MIDP apps. The following code calls the Vector's size() method everytime it loops. Depending on the number of loops called, this could have an effect on the speed of execution of your app code.

```
for (int i=0; i&lt;theVector.size(); i++) { Object x=theVector.elementAt(i);
 }
```

Instead, calculate the size of the Vector before the loop.

```
int theSize=theVector.size();

 for (int i=0;i&lt;theSize;i++) {
  Object x=theVector.elementAt(i);
  }
</code java>

12. Use Simple Containers Containers, such as Vector and Map are great, but they’re also slow. If possible just use an array which i

Use Strongly Typed Arrays Use strongly typed arrays where possible, rather than using object arrays to store types. This avoids type

To avoid the boxing overhead declare a strongly typed int array, as follows:
<code java>
int [] arrIn = new int [10]; arrIn[0] = 2 +3;
```

Storing reference types, such as string or custom classes in the array of objects, involves the typecasting overhead. Therefore, you should use strongly typed arrays to store your reference types to, as shown in the following code sample.

```
string[10] arrStr = new string[10];
arrStr[0] = new string(“abc”);
```

13. Use local variables instead of instance variables if possible It takes slightly longer for Midlets to access instance variables than local variables. For example, instead of accessing an instance variable multiple times from inside a loop, set a local variable to the instance variable, then use the local variable in the loop.

```
int x=object.iVariable;

for (int i=0;i&lt;someNumber;i++) {

 temp=i+x;
```

```
}
```

14. Less class the better. Minimize the number of classes in your jar file: (a) Only include the classes you need in your final jar, (b) consider removing "bells and whistles" functionality, and (c) refrain from using inner classes.

Break up extra packages that you had simply dumped into your directory during the developmental stage, such as packages for XML parsing. The extra time it takes to do this might make the difference between a viable MIDlet and one that cannot be commercially deployed. Prune unnecessary functionality from your app by leaving out some classes. Try to get rid of "bells and whistles" functionality that may not add anything essential to the whole. Refrain from using inner classes, and especially anonymous inner classes. In addition to incurring a certain amount of overhead due to the additional classes, the compiler generates additional methods and variables that allow the inner classes access to the private variables of the class enclosing 1

5. Reduce the Image Size.

```
Images can add a lot of to the total MIDlet size, and some time spent looking over the images you want to include in the MIDlet suit
```

Only include images that you feel are necessary to the app, and keep the number of colors down to reduce the file size of each image. Since many of the target devices can display only a few colors per pixel, creating images with many more colors per pixel may not add anything of value to your app. Finally, reducing the actual dimensions of each image will also reduce its file size. Another way to get around the image size problem is to request the image from the server when necessary. Of course, this is probably not advisable because (1) it may create new problems of long downloading times for the end user and more importantly (2) it necessitates that the end user be able to connect to the server, something that is probably not the case for the majority of end users.

16. Use Fast Methods

```
• synchronized methods are the slowest, since an object lock has to be obtained
• interface methods are the next slowest
```

• instance methods are in the middle • final methods are faster • static methods are fastest So we definitely shouldn't be making anything synchronized, and it looks like we can even mark work() and workMore() as final static methods. Doing this cut 1% from the time the emulator spent inside run(). Another factor that affects the performance of method calls is the number of parameters passed into the method. We are calling workMore() 51712 times, passing an int array and two ints into the method and returning an int each time. try to reduce the number of parameters you're passing into it. The more parameters, the greater the overhead.

```
public final static int work (int[] n ) {

 divisor = 1;
  r = 0;
  for ( int j = 0; j &lt; DIVISOR_COUNT&amp;nbsp;; j++ ) {
  for ( int i = 0&amp;nbsp;; i &lt; n.length&amp;nbsp;; i++ ) {
  r += n[i] * n[i] / divisor + n[i];
  }
  divisor *= 2;
  }
  return r;
```

17. Declare Variables as Static

```
 static int employee;
```

Do u know an int declared as static saves 3 bytes.

18. Use Validation Code to Reduce Unnecessary Exceptions If you know that a specific avoidable condition can happen, proactively write code to avoid it. For example, adding validation checks such as checking for null before using an item from the cache can significantly increase performance by avoiding exceptions.

The following code uses a try/catch block to handle divide by zero.

```
int result = 0;
try {
result = numerator/divisor;
} catch(Exception e)
{ System.out.println (“exception caught here”); }
```

The following rewritten code avoids the exception, and as a result is more efficient.

```
int result = 0;
if(divisor != 0)
 result = numerator/divisor;
else System.out.println(“division by 0 error”);
```

19. Try to compare to zero instead of any other number

```
public class Silly {

 // ...

 public void sillyMethod ()
 {
  int len = char_array.length;
  for (int count = 0; count &lt; len; count++)
  System.out.println(char_array[count]);
  return;
 }
```

```
}
```

In fact, however, there is still one more optimization we can make. On most platforms, it is much faster to compare a variable against zero than against some arbitrary value. The reasoning behind this is that at the machine code level, on some platforms (PowerPC of note), the code to compare count against len consists of subtracting the two and comparing the difference against zero.

```
public class Silly {

 // ...

 public void sillyMethod ()
 {
  int len = char_array.length;
  for (int count = len; count &gt; 0; count--)
  System.out.println(char_array[count-1]); // Different output, but only for example
 }
```

The above code prints the array in reverse order, which is probably not what we wanted, but for a more realistic example where we might be adding the values in an array, etc., this is a real optimization. These and other techniques were discussed during technical sessions at JavaOne.

20. Avoid Synchronized Blocks

Adding Synchronized blocks creates extra bytes. The programmers are responsible for ensuring that all critical methods are protected with synchronized. The extra CPU cycles involved in obtaining and releasing a lock can have a noticeable effect on a program's speed.

Instead of doing like this:

```
Class Example {
  private synchronized void doSomething()
  {
    public synchronized void action()
  { doSomething(); }
  }}
```

Do this:

```
Class Example { //assumes synchronized 'this'
 private void doSomething() {
   public synchronized void action()
   { doSomething(); } } }
```

21. Don't make assumptions about fonts Fonts cannot be created by applications. Instead, an application requests a font based on attributes (i.e., size, face, style) and the underlying implementation will attempt to return a font that closely resembles the requested font. The Font class represents various fonts and metrics. There are three font attributes defined in the Font class, and each may have different values, as follows: - Face MONOSPACE, PROPORTIONAL, SYSTEM - Size SMALL, MEDIUM, LARGE - Style BOLD, ITALIC, PLAIN, UNDERLINED

For example, to specify a medium size font, use Font.SIZE_MEDIUM, and to specify an italic style, use Font.STYLE_ITALIC, and so on. Values for the style attributes may be combined using the OR (|) operator; values for the other attributes may not be combined. For example, the value of this style attribute specifies a plain, underlined font: STYLE_PLAIN | STYLE_UNDERLINED

However, the following is illegal: SIZE_SMALL | SIZE_MEDIUM This is also illegal: FACE_SYSTEM | FACE_MONOSPACE Each font in the system is actually implemented individually, so in order to obtain an object representing a font, use the getFont( ) method. This method takes three arguments for the font face, size, and style, respectively. For example, the following snippet of code obtains a Font object with the specified face, style, and size attributes: Font font = Font.getFont(FACE_SYSTEM,STYLE_PLAIN,SIZE_MEDIUM); If a matching font does not exist, the implementation will attempt to provide the closest match, which is always a valid Font object. Once a font is obtained, you can use methods from the Font class to retrieve information about that font. For example, you can use the methods getFace(), getSize( ), and getStyle( ) to retrieve information about the face, size, and style of the font, respectively.

22. Use bit shift operators instead of division or multiplication by a power of two Instead of divide like this, int mid = (hi + lo) / 2; Do the right shift operator to get the result. int mid = (hi + lo) >> 1;

23. Array access is slower than C, so cache array elements Cache Class Variables Instead of doing this:

```
Class Example {
int total_sum;
void accumulate (byte a[]){
for(int i = a.length – 1; i>=0; --i;)
{ total_sum += a[i]; } } }
```

Do this: cache the class variables

```
Class Example {
int total_sum;
void accumulate (byte a[]) {
 int totalsum; for(int i = a.length – 1; i >=0; --i; {
totalsum += a[i];
}
total_sum = totalsum; } }
```

Ensure that the class variable doesn't change its value during the method or reflect changes to the cached local.

24. Use small, close constants in switch() statements.-

When you use a switch, it is good programming practice to write code like this:

```
  public static final int STATE_RUNNING = 1000;
```

```
 public static final int STATE_JUMPING = 2000;
 public static final int STATE_SHOOTING = 3000;
 switch ( n ) {
 case STATE_RUNNING:
 doRun();
 case STATE_JUMPING:
 doJump();
 case STATE_SHOOTING:
 doShoot();
}
```

There's nothing wrong with this, and the int constants are nice and far apart, in case we might want to stick another constant in between RUNNING and JUMPING, like STATE_DUCKING = 2500. But apparently switch statements can be compiled into one of two byte codes, and the faster of the two is used if the ints used are close together, so this would be better:

```
public static final int STATE_RUNNING = 1;
 public static final int STATE_JUMPING = 2;
 public static final int STATE_SHOOTING = 3;
```

25. Avoid Inner Class: Inner classes have their disadvantages. From a maintenance point of view, inexperienced Java developers may find the inner class difficult to understand. The use of inner classes will also increase the total number of classes in your code. Moreover, from a development point of view, most Java tools come up a bit short on their support of inner classes. Inner Classes provide many syntactic conveniences but have hidden implementation costs (extra methods and class variables) Use Static inner classes initially, only move to non-static inner classes when necessary.

Example program:

```
class Example { static class Nested { } }
```

Don't access private member variables from nested classes. Example program:

```
class Example { private static intBallX; static class Nested {
 void doSomething()
{
 ++intBallX;
} } }
```

Avoid inner classes within methods.

```
Example program:
```

```
class Example {
 void doSomething()
{
 class Local implements Runnable
{ public void run() { } }
} }
```

26. Null Out the Old References Be sure to help garbage collector to do its work by setting object reference to null whenever you are finished with them For Example: You might define a deinitialize method for a class in order to explicitly clear out its members

```
Public class myClass {
 private SomeObject someObject;
  public myClass(Object obj)
  {
  --------------------
  }
 public void deinitialize()
  {
  someObject = null;
  }
}
```

By clearing out object references, you make it easier for the garbage collector to find and reclaim and unreferenced objects.This technique is simple and does not add too much code to your classes.

```
The garbage collector works automatically, whether you request it to or not. However, it is possible to make a garbage collector req
```

27. Don't wait() if you can callSerially(). In addition, you can use the callSerially( ) method of the Display class to execute code after all pending repaints are served, as shown in the following segment of code:

```
class TestCanvas extends Canvas implements Runnable {
 void doSomething( ) {
  // code fragment 1
```

```
  callSeriallÿ(this);
  }


 public void run( ) {
  // code fragment 2
  }

}
```

Here, the object's run( ) method will be called after the initial call.

28. Use Tried and Tested Algorithm: A*: The Tried and Tested Solution for Pathfinding A* is an improved version of Dijkstra's shortest-path algorithm.1,2 Though it can be used for a range of search problems, its primary application area is pathfinding. For those of you unfamiliar with the A* algorithm, here is a more detailed explanation. As explained in the accompanying article, the map is represented by a set of location nodes/waypoints, some of which have connections of certain distances. Given a start node and a destination node, the algorithm has to find the shortest path between those two points. The algorithm searches stepwise from the start node toward the destination. The algorithm maintains two lists: open and closed. The open list contains all new nodes that could be visited in the next step from the already-visited nodes. The closed list contains all nodes that were already visited. The open list is initialized with the start node. The algorithm has found the shortest path once the destination node is added to the open list. The closed list starts empty. The nodes of the open list are ranked according to the formula: f(n) = g(n) + h(n) where g(n) is the shortest distance along the already-visited connections from the start node to node n; and h(n) is an estimate of the remaining distance from node n to the destination node. It is important that the estimate is lower than or equal to the actual shortest distance along possible connections.

29. CatchOutOfMemory Why would you want to catch an OutOfMemoryError? If an OutOfMemoryError is generated by the JVM, there is not much you can do, so what use is the OutOfMemoryError? Encountering an OutOfMemoryError means that the garbage collector has already tried its best to free memory by reclaiming space from any objects that are no longer strongly referenced. If it could not reclaim enough space, then it also tried to obtain memory from the underlying operating system, unless heap space is already at the JVM upper memory bound set by the -Xmx parameter (-mx in JVMs prior to Java 2). So encountering the OutOfMemoryError means that there is no more heap space that can currently be reclaimed, and that either the operating system cannot provide any more memory to the JVM or you have reached the JVM upper memory bound. In any case, there is not much you can do, so when would you ever want to catch an OutOfMemoryError? The following sections describe a few special situations when it can be useful to catch an OutOfMemoryError. Expanding Memory and Determining Memory Limits The JVM heap space is the memory area where all objects reside. In addition to objects, the heap can also contain memory reserved for the garbage collector and for some other JVM activities. The overall heap size is normally set by two parameters of the Java executable: • -Xms (-ms before Java 2) to specify the initial heap size when the JVM starts up; and • -Xmx (-mx before Java 2) to specify the maximum heap size that the JVM is allowed to grow to. If these parameters are not specified, the JVM uses default values that vary, depending on the JVM version and vendor. The default initial values are usually one or two megabytes, and the default maximum values are typically between 16 and 64 Mbytes. Clearly, if you control how the JVM is started, you can specify the heap values you want. But what if your application is running in a JVM which was not started under your control? How can you determine what size the JVM can reach? There is a method in the Runtime class, Runtime.totalMemory(), which looks like it would give us this information, but the value returned is the current memory size, not the largest possible memory size. In Java version 1.4 there will be a new method, Runtime.maxMemory(), which returns the -Xmx value, but that doesn't help with earlier JVM versions. In addition, even if we know the value passed to -Xmx, we are not guaranteed that the JVM can reach the indicated size, since a size may specified that is too big for the underlying system. One way to reliably determine the maximum possible heap size is to grow the heap until it can no longer expand. Doing this is quite simple: keep creating objects until we hit an OutOfMemoryError. The following testMemory() method repeatedly creates objects with a size of one megabyte until an OutOfMemoryError is generated:

```
public static final int MEGABYTE = 1048576; public static long testMemory(int maximumMegabytesToTest) {

 //Hold on to memory, or it will be garbage collected
  Object[] memoryHolder = new Object[maximumMegabytesToTest];
  int count = 0;
  try
  {
  for (; count &lt; memoryHolder.length; count++)
  {
  memoryHolder[count] = new byte[MEGABYTE];
  }
  }
  catch(OutOfMemoryError bounded){}
  long highWater = Runtime.getRuntime().totalMemory();
  // System.out.println("High water in bytes: "
  // + highWater);
  // System.out.println("Megabytes allocatable in
  // megabytes: " + count);
  memoryHolder = null; //release for GC
  //We know we could allocate "count" megabytes and
  //have a high water mark of "highWater". Return
  //whichever you prefer.
  //return count;
  return highWater;

}
```

The method returns the size that the heap reached when the OutOfMemoryError is generated. There are, however, some consequences to using this method. Firstly, although I use a one megabyte size to incrementally request memory, the actual size allocated will be more than one megabyte since the byte[] array object has some additional size overhead from being an object. Secondly, if the heap is fragmented, and the garbage collector cannot or does not defragment the heap sufficiently, there will actually be further space in the heap for object creation, even though we cannot create further megabyte-sized objects. Finally, the JVM may have grown so much that it will now be paged by the operating system (see the "Operating system paging" sidebar), which will cause a significant decrease in performance. This would occur anyway if the JVM needed to grow large enough during the normal operation of the program, but running the testMemory() method would impose the overhead sooner. None of these points matter if you are only interested in the high-water mark, that is, the maximum size that the heap can be grown to, but can be important in other situations. With the help of code optimization, the developers can reduce the coding or source memory size within 28 KB or 44% of our game budget(64 KB).