**Chapter 1: Loading a Sprite**

**Intro Blurb**  
Every great game starts with a player, an enemy, or even a background image. These visual elements are called **sprites**. In this chapter, you’ll learn how to load a sprite into your game window using Java’s JPanel.

**Step-by-Step Coding**

**STEP 1: Import Libraries**  
We need javax.swing, java.awt, and javax.imageio to handle graphics and images.

|  |
| --- |
| import javax.swing.\*;  import java.awt.\*;  import java.awt.image.\*;  import java.io.\*;  import javax.imageio.ImageIO; |

**Explanation:**

* javax.swing.\* → For JPanel and window components.
* java.awt.\* → For drawing graphics on the panel.
* java.awt.image.\* → Gives us BufferedImage to store the sprite.
* java.io.\* → Needed for reading the image file.
* javax.imageio.ImageIO → Special class for loading image files (PNG, JPG, etc.)

**STEP 2: Create GamePanel Class**

|  |
| --- |
| public class GamePanel extends JPanel {  private BufferedImage sprite; |

**Explanation:**

* GamePanel extends JPanel, meaning it’s a custom panel where we control drawing.
* BufferedImage sprite; stores the image data in memory so it can be drawn later.

**STEP 3: Load the Sprite**

|  |
| --- |
| public GamePanel() {  this.setPreferredSize(new Dimension(800, 600));  try {  sprite = ImageIO.read(new File("player.png"));  } catch (IOException e) {  e.printStackTrace();  }  } |

**Explanation:**

* setPreferredSize(800,600) → Makes the panel 800x600 pixels (so pack() in Main knows the size).
* ImageIO.read(new File("player.png")) → Reads the sprite from your project folder.
* try...catch → Catches errors (e.g., if the file doesn’t exist).

**STEP 4: Draw the Sprite**

|  |
| --- |
| @Override  protected void paintComponent(Graphics g) {  super.paintComponent(g);  if (sprite != null) {  g.drawImage(sprite, 100, 100, null);  }  } |

**Explanation:**

* paintComponent(Graphics g) → The main drawing method for JPanel.
* super.paintComponent(g) → Clears the panel before drawing (avoids overlapping images).
* g.drawImage(sprite, 100, 100, null) → Draws the sprite at position (x=100, y=100).

**What is an ImageObserver?**

* In Java, when drawing images, the system may **not finish loading the image immediately** (especially if it’s from the internet).
* To handle this, Java provides an **ImageObserver interface**.
* It’s like a “watcher” that gets notified when the image finishes loading or changes (for animated GIFs).

**Why do we pass null?**

* If you **don’t care** about being notified when the image updates, you can just pass null.
* That’s fine for **simple games**, since we usually load all sprites from files before drawing (they’re already fully available).

**Example with Observer**

If you wanted to use an ImageObserver, you might do:

|  |
| --- |
| g.drawImage(sprite, 100, 100, this); |

Here, this is the current JPanel (which already implements ImageObserver indirectly).

* Now, the panel would automatically **repaint itself** when the image finishes loading.

**Summary**

* null → Means “no observer, just draw it.”
* this → Can be used if you want the panel to observe image loading.
* For **game development**, null is almost always fine because sprites are preloaded.

**LOGIC CHECK (10 Qs + Answers)**

1. Q: What is a sprite?  
   A: A 2D image used in a game for characters, objects, or backgrounds.
2. Q: Which method is used to draw graphics on a JPanel?  
   A: paintComponent(Graphics g)
3. Q: Which class is used to load images from a file?  
   A: ImageIO
4. Q: What file type can ImageIO.read() load?  
   A: PNG, JPG, BMP, and GIF (common image formats).
5. Q: Why do we call super.paintComponent(g) first?  
   A: To clear the panel before drawing new graphics.
6. Q: What happens if player.png is missing?  
   A: An IOException will be thrown, and nothing will be drawn.
7. Q: Where is the sprite drawn in this code?  
   A: At pixel position (100,100).
8. Q: What happens if you don’t set PreferredSize?  
   A: The panel may shrink, and pack() won’t size it properly.
9. Q: Which Java package provides BufferedImage?  
   A: java.awt.image
10. Q: What command makes the frame visible in Main.java?  
    A: frame.setVisible(true);

**Mini Capstone: Draw Two Sprites**

Modify the code so you load **two images**: player.png and enemy.png. Display them at different positions on the panel.

|  |
| --- |
| import javax.swing.\*;  import java.awt.\*;  import java.awt.image.\*;  import java.io.\*;  import javax.imageio.ImageIO;  public class GamePanel extends JPanel {  private BufferedImage playerSprite;  private BufferedImage enemySprite;  public GamePanel() {  this.setPreferredSize(new Dimension(800, 600));  try {  playerSprite = ImageIO.read(new File("player.png"));  enemySprite = ImageIO.read(new File("enemy.png"));  } catch (IOException e) {  e.printStackTrace();  }  }  @Override  protected void paintComponent(Graphics g) {  super.paintComponent(g);  if (playerSprite != null) {  g.drawImage(playerSprite, 100, 100, null);  }  if (enemySprite != null) {  g.drawImage(enemySprite, 400, 100, null);  }  }  } |

**PRO TIP**  
Organize your game project with folders like assets/images/.

**Bonus: How to Change Sprite Size**

You can use the drawImage method with **extra width and height parameters**:

|  |
| --- |
| g.drawImage(sprite, x, y, newWidth, newHeight, null); |

**Chapter 2: Moving a Sprite with Keyboard Input**

**Intro Blurb**

Right now, your sprite just sits there like a statue.   
It’s time to bring it to life by making it move around the screen using the **arrow keys**.

**Step 1: Track Sprite Position**

Inside **GamePanel.java**, we add x and y variables to remember where the sprite is.

|  |
| --- |
| private int x = 100, y = 100; // sprite starting position |

**Explanation:**

* x → horizontal (left/right) position
* y → vertical (up/down) position
* Starting at (100, 100) means the sprite begins 100px from the left and 100px from the top.

**Step 2: Add Movement Methods**

|  |
| --- |
| public void moveUp() { y -= 5; repaint(); }  public void moveDown() { y += 5; repaint(); }  public void moveLeft() { x -= 5; repaint(); }  public void moveRight() { x += 5; repaint(); } |

**Explanation:**

* Place it below GamePanel Constructor Bracket
* Each method changes x or y by 5 pixels.
* Example: y -= 5 moves the sprite **up**.
* repaint() tells the panel to redraw the screen with the new position.

**Step 3: Draw the Sprite at New Position**

|  |
| --- |
| @Override  protected void paintComponent(Graphics g) {  super.paintComponent(g);  if (sprite != null) {  g.drawImage(sprite, x, y, 50, 50, null);  }  } |

**Explanation:**

* Always called when the panel is refreshed.
* x, y decide where the sprite appears.
* 50, 50 sets the **width and height** of the sprite.
* null = no image observer (we don’t need one for static images).

**Step 4: Create the KeyHandler**

Now, let’s make a new file **KeyHandler.java**:

**Import & Class Setup**

|  |
| --- |
| import java.awt.event.KeyEvent;  import java.awt.event.KeyListener;  public class KeyHandler implements KeyListener { |

**Explanation:**

* KeyEvent → gives info about which key was pressed.
* KeyListener → interface we must implement to detect key events.
* KeyHandler → our custom class that listens to keys.

**Reference to GamePanel**

|  |
| --- |
| private GamePanel panel;  public KeyHandler(GamePanel panel) {  this.panel = panel;  } |

**Explanation:**

* We keep a panel reference so the KeyHandler can tell the GamePanel to move the sprite.
* When we create KeyHandler inside GamePanel, we pass this (the current panel) into it.

**Detect Key Presses**

|  |
| --- |
| @Override  public void keyPressed(KeyEvent e) {  int code = e.getKeyCode();  if (code == KeyEvent.VK\_UP) panel.moveUp();  if (code == KeyEvent.VK\_DOWN) panel.moveDown();  if (code == KeyEvent.VK\_LEFT) panel.moveLeft();  if (code == KeyEvent.VK\_RIGHT) panel.moveRight();  } |

**Explanation:**

* keyPressed runs when a key is pushed down.
* e.getKeyCode() gives us a unique number for each key.
* KeyEvent.VK\_UP, VK\_DOWN, etc. → constants for the arrow keys.
* Each arrow key calls the matching movement method in GamePanel.

**Empty Methods**

|  |
| --- |
| @Override  public void keyReleased(KeyEvent e) {}  @Override  public void keyTyped(KeyEvent e) {}  } |

**Explanation:**

* keyReleased runs when you release a key. We’ll use this later for smoother movement.
* keyTyped is for character keys (like typing letters), so we don’t need it for movement.
* Both are **required** because KeyListener forces us to implement all 3 methods.

**Step 5: Connect KeyHandler to GamePanel**

Inside the GamePanel constructor:

|  |
| --- |
| KeyHandler keyH = new KeyHandler(this);  this.setFocusable(true);  this.addKeyListener(keyH); |

**Explanation:**

* new KeyHandler(this) → passes the current panel to the KeyHandler.
* setFocusable(true) → allows the panel to detect key input.
* addKeyListener(keyH) → attaches the KeyHandler so it listens for key presses.

**LOGIC CHECK (10 Questions)**

1. What do x and y represent?
   * A) Window size
   * B) Sprite position
   * C) Sprite speed
2. What method refreshes the panel after movement?
   * A) repaint()
   * B) draw()
   * C) reset()
3. What does y -= 5; do?
   * A) Moves sprite up
   * B) Moves sprite down
   * C) Moves sprite left
4. Where is the sprite drawn?
   * A) paintComponent()
   * B) main()
   * C) keyPressed()
5. Which class listens to key input?
   * A) Main
   * B) GamePanel
   * C) KeyHandler
6. Why do we need setFocusable(true)?
   * A) To load images
   * B) To detect key input
   * C) To set size
7. What happens if the sprite image is missing?
   * A) Program crashes
   * B) Error is printed
   * C) A square is drawn
8. Which key event runs when a key is pressed?
   * A) keyPressed
   * B) keyReleased
   * C) keyTyped
9. What does g.drawImage(sprite, x, y, 50, 50, null) do?
   * A) Draws sprite at position with size
   * B) Deletes sprite
   * C) Moves panel
10. Which file contains the main game window?
    * A) Main.java
    * B) GamePanel.java
    * C) KeyHandler.java

**PRO TIP**

Right now, movement happens in **steps per key press**. For smoother animation, we’ll later switch to a **game loop** where pressing a key sets a direction, and releasing it stops movement. This makes movement flow naturally — just like in real games.

**BONUS SECTION**

**Changes to GamePanel.java**

First, add a speed variable:

|  |
| --- |
| private int speed = 5; // default speed |

Then, update your move methods to use this speed:

|  |
| --- |
| public void moveUp() { y -= speed; repaint(); }  public void moveDown() { y += speed; repaint(); }  public void moveLeft() { x -= speed; repaint(); }  public void moveRight() { x += speed; repaint(); } |

**Changes to KeyHandler.java**

We’ll check if **Shift** is also pressed when moving.

Above the **if (code == KeyEvent.VK\_UP)    panel.moveUp();**

|  |
| --- |
| // If Shift is held down → increase speed  if (e.isShiftDown()) {  panel.setSpeed(15); // faster movement  } else {  panel.setSpeed(5); // normal speed  } |

We also need a setter in GamePanel: below **public void moveRight() { x += speed; repaint(); }**

|  |
| --- |
| public void setSpeed(int speed) {  this.speed = speed;  } |

**Chapter 3: Animating Sprites in Java — Step-by-Step**

**Intro Blurb**

In this lesson, we’ll learn how to animate a character using sprite images. Sprites are individual frames of movement (like walking up, down, left, or right). By quickly switching between these frames, we create the illusion of animation.

We’ll build on your GamePanel code to handle movement + animation when pressing keys.

**Step-by-Step Coding (Line-by-Line Explanation)**

**1. Import the required libraries**

|  |
| --- |
| import java.awt.\*;  import java.awt.image.BufferedImage;  import java.io.File;  import javax.imageio.ImageIO;  import javax.swing.\*; |

* **java.awt → for graphics.**
* **BufferedImage → stores sprite images.**
* **ImageIO → loads images from files.**
* **JPanel → custom drawing area for the game.**

**2. Create the GamePanel class**

|  |
| --- |
| **public class GamePanel extends JPanel {** |

* **Extends JPanel so we can override paintComponent to draw graphics.**

**3. Define game settings**

|  |
| --- |
| **private final int WIDTH = 800;**  **private final int HEIGHT = 600;** |

* **Width and height of the game window.**

**4. Character state**

|  |
| --- |
| **private int x = 100;**  **private int y = 100;**  **private int speed = 5;** |

* **Starting position (100,100)**
* **Speed controls how many pixels the character moves per key press.**

**5. Load sprites**

|  |
| --- |
| **private BufferedImage up1, up2, up3, up4, up5, up6;**  **private BufferedImage currentImage;** |

* **Each variable stores one frame of the walking animation.**
* **currentImage → the one being drawn at the moment.**

**6. Animation index**

|  |
| --- |
| **private int whatframe = 0;** |

* **Keeps track of which frame we are currently on.**

**7. Constructor**

|  |
| --- |
| public GamePanel() {  setPreferredSize(new Dimension(WIDTH, HEIGHT));  setBackground(Color.BLACK);  loadSprites();  setFocusable(true);  addKeyListener(new KeyHandler(this));  } |

* **Sets panel size and background.**
* **Calls loadSprites() to load images.**
* **addKeyListener(new KeyHandler(this)) → listens for arrow key input.**

**8. Load sprite images**

|  |
| --- |
| private void loadSprites() {  try {  up1 = ImageIO.read(new File("...Krt000\_06.png"));  up2 = ImageIO.read(new File("...Krt000\_07.png"));  up3 = ImageIO.read(new File("...Krt001\_00.png"));  up4 = ImageIO.read(new File("...Krt001\_01.png"));  up5 = ImageIO.read(new File("...Krt001\_02.png"));  up6 = ImageIO.read(new File("...Krt001\_03.png"));  } catch (Exception e) {  e.printStackTrace();  }  } |

* **Loads six walking frames.**
* **Each file must exist at the correct path.**

**9. Movement with animation**

|  |
| --- |
| public void moveUp() {  y -= speed;  whatframe = (whatframe + 1) % 6; // cycle 0–5  switch (whatframe) {  case 0 -> currentImage = up1;  case 1 -> currentImage = up2;  case 2 -> currentImage = up3;  case 3 -> currentImage = up4;  case 4 -> currentImage = up5;  case 5 -> currentImage = up6;  }  repaint();  } |

* **Moves character upward.**
* **Cycles frames to simulate walking.**
* **Calls repaint() → redraw panel with new image.**

**10. Empty placeholders (for later)**

|  |
| --- |
| public void moveDown() { }  public void moveLeft() { }  public void moveRight() { } |

* **To be implemented for other directions.**

**11. Drawing the character**

|  |
| --- |
| @Override  protected void paintComponent(Graphics g) {  super.paintComponent(g);  if (currentImage != null) {  g.drawImage(currentImage, x, y, null);  }  } |

* **Clears panel.**
* **Draws the character at (x,y).**

**12. Extracted Additions (DOWN Movement)**

New Variables (add under your UP sprites section)

|  |
| --- |
| **private BufferedImage d1, d2, d3, d4, d5, d6; // new variables** |

Add this just below your UP movement sprites (up1 … up6).

**13. Set Initial Current Image (inside constructor)**

|  |
| --- |
| **currentImage = d1; // new variable d1** |

**Add this inside GamePanel() constructor, right after loadSprites();**

**14. Load DOWN Sprites (inside loadSprites())**

|  |
| --- |
| d1 = ImageIO.read(new File("...Krt000\_06.png"));  d2 = ImageIO.read(new File("...Krt000\_07.png"));  d3 = ImageIO.read(new File("...Krt001\_00.png"));  d4 = ImageIO.read(new File("...Krt001\_01.png"));  d5 = ImageIO.read(new File("...Krt001\_02.png"));  d6 = ImageIO.read(new File("...Krt001\_03.png")); |

**Add this below the UP movement sprite loading block inside loadSprites().**

**15. New Movement Method (add under moveUp)**

|  |
| --- |
| public void moveDown() {      y += speed;      whatframe = (whatframe + 1) % 6;      switch (whatframe) {          case 0 -> currentImage = d1;          case 1 -> currentImage = d2;          case 2 -> currentImage = d3;          case 3 -> currentImage = d4;          case 4 -> currentImage = d5;          case 5 -> currentImage = d6;      }      repaint();  } |

**16. Add a New Variable (under your sprite declarations)**

|  |
| --- |
| private BufferedImage stoneWalk; // new image |

Place this **below your d1–d6 variables**.

**17. Load the Image (inside loadSprites())**

|  |
| --- |
| stoneWalk = ImageIO.read(new File("C:\\Users\\franzyzyto\\Desktop\\lecturesept\\character\\stonewalk.png")); |

private Rectangle playerBounds;

private Rectangle stoneBounds;  
**18. Draw the Image (inside paintComponent(Graphics g))**

Right now, paintComponent only draws currentImage.  
If you also want to **draw the stone background/object**, you can do this:

|  |
| --- |
| // draw stone first (background or object)  if (stoneWalk != null) {  g.drawImage(stoneWalk, 200, 300, null); // adjust position (x=200, y=300)  } |

Place this code **inside your existing paintComponent**, before drawing the player.

**19. Add Variable idle Image for the key release**

|  |
| --- |
| private BufferedImage endImage; // Idle sprite |

Add after the currentImage Variable

**20. Add endImage value for key Release**

|  |
| --- |
| endImage = d1; |

Add after the currentImage = d1

**21. Idle setter (called when key released)**

|  |
| --- |
| public void setIdle() {  currentImage = endImage;  repaint();  } |

Add this before paintComponent

**22. KeyHandler calls setIdle() when the key is released:**

|  |
| --- |
| @Override  public void keyReleased(KeyEvent e) {  panel.setIdle(); // switch to idle sprite  } |

**📘 Chapter 4: Character Meets the Obstacle**

**Intro Blurb**

In the last chapter, your player could move around freely without worrying about bumping into anything. That was fun—but not realistic. Games need **obstacles** so the world feels alive. Imagine walking through walls in Pokémon—it breaks immersion.

In this chapter, we add **pixel-perfect collision detection** and **scalable object sizes**. That means:

* Your character **won’t walk through walls or objects anymore**.
* Obstacles like stones or trees can be resized easily.
* Movement feels more like a **real game**.

This is the first step toward **map design, dungeons, and interactive worlds**.

**Step-by-Step Coding**

**STEP 1: Add Scalable Sizes**

|  |
| --- |
| private int playerWidth = 100;  private int playerHeight = 100;  private int stoneX = 550;  private int stoneY = 100;  private int stoneWidth = 50;  private int stoneHeight = 50; |

**Where to place:**  
Put these right **below your existing variables** (like private int upFrame = 0;). This keeps all movement and size attributes grouped together.

**Explanation:**

* private int playerWidth = 100; → sets how wide your player sprite is drawn.
* private int playerHeight = 100; → sets how tall your player sprite is drawn.
* private int stoneX = 550; → the stone’s horizontal position on screen.
* private int stoneY = 100; → the stone’s vertical position on screen.
* private int stoneWidth = 50; → width of the stone.
* private int stoneHeight = 50; → height of the stone.

**STEP 2: Pixel-Perfect Collision Method**

|  |
| --- |
| // 🔥 Pixel-perfect collision check with scaling  private boolean pixelPerfectCollision(BufferedImage img1, int x1, int y1, int w1, int h1,  BufferedImage img2, int x2, int y2, int w2, int h2) {  // find overlapping rectangle between two objects  int topX = Math.max(x1, x2);  int topY = Math.max(y1, y2);  int bottomX = Math.min(x1 + w1, x2 + w2);  int bottomY = Math.min(y1 + h1, y2 + h2);  // loop over every pixel in the overlap area  for (int y = topY; y < bottomY; y++) {  for (int x = topX; x < bottomX; x++) {    // map pixel from screen → image1 coordinates  int img1X = (x - x1) \* img1.getWidth() / w1;  int img1Y = (y - y1) \* img1.getHeight() / h1;  int pixel1 = img1.getRGB(img1X, img1Y);  // map pixel from screen → image2 coordinates  int img2X = (x - x2) \* img2.getWidth() / w2;  int img2Y = (y - y2) \* img2.getHeight() / h2;  int pixel2 = img2.getRGB(img2X, img2Y);  // if BOTH pixels are visible (not transparent) → collision  if (((pixel1 >> 24) & 0xff) > 0 && ((pixel2 >> 24) & 0xff) > 0) {  return true; // collision happened  }  }  }  return false; // no collision  } |

**Where to place:**  
Just **below your loadSprites() method** but before moveUp(). This is a **helper function** that movement methods will call.

* private boolean pixelPerfectCollision(...) → defines a helper method returning true/false.
* Math.max(x1, x2) → finds the left boundary of the overlap.
* Math.min(x1 + w1, x2 + w2) → finds the right boundary of the overlap.
* for (int y = topY; y < bottomY; y++) → loops over every pixel row inside the overlap.
* (x - x1) \* img1.getWidth() / w1 → maps screen coordinate back to image coordinate.
* img1.getRGB(img1X, img1Y) → gets the pixel color from image 1.
* ((pixel1 >> 24) & 0xff) → extracts the alpha channel (transparency).
* If both pixels are visible → return true (collision).
* If loop finishes → return false (no collision).

**Sample Computation with Values**

Example:

* img1 = 100×100 pixels (original size).
* img2 = 50×50 pixels (original size).

On screen:

* img1 drawn at (x1=10, y1=20) with size (w1=200, h1=200) → scaled **2× bigger**.
* img2 drawn at (x2=100, y2=150) with size (w2=100, h2=100) → scaled **2× bigger**.

**Step 1: Find overlap rectangle**

|  |
| --- |
| topX = max(10, 100) = 100  topY = max(20, 150) = 150  bottomX = min(10+200, 100+100) = min(210, 200) = 200  bottomY = min(20+200, 150+100) = min(220, 250) = 220 |

Overlap area = rectangle from (100, 150) to (200, 220).

**Step 2: Pick 1 pixel inside overlap, e.g. (x=120, y=160).**

For img1:

|  |
| --- |
| img1X = (x - x1) \* img1.getWidth() / w1  = (120 - 10) \* 100 / 200  = 110 \* 100 / 200  = 55  img1Y = (y - y1) \* img1.getHeight() / h1  = (160 - 20) \* 100 / 200  = 140 \* 100 / 200  = 70 |

Maps to pixel (55, 70) in original img1.  
  
For img2:

|  |
| --- |
| img2X = (x - x2) \* img2.getWidth() / w2  = (120 - 100) \* 50 / 100  = 20 \* 50 / 100  = 10  img2Y = (y - y2) \* img2.getHeight() / h2  = (160 - 150) \* 50 / 100  = 10 \* 50 / 100  = 5 |

Maps to pixel (10, 5) in original img2.

**Step 3: Check pixels**

* pixel1 = img1.getRGB(55, 70)
* pixel2 = img2.getRGB(10, 5)
* If both have alpha > 0 → 💥 collision!

**STEP 3: Update moveUp()**

|  |
| --- |
| int nextY = y - speed; // calculate potential new Y position  // check collision: player vs stone  if (!pixelPerfectCollision(currentImage, x, nextY, playerWidth, playerHeight,  stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight)) {  y = nextY; // only move if no collision  } |

**Where to place:**  
add this to your existing moveUp() method. Remove **y -= speed;** and paste the code

**Explanation:**

* int nextY = y - speed; → calculates where player would move if pressing up.
* if (!pixelPerfectCollision(...)) → only move if no collision.
* y = nextY; → updates position.
* upFrame = (upFrame + 1) % 6; → cycles through 6 frames for animation.
* switch (upFrame) → picks correct sprite for animation.
* repaint(); → redraws everything after movement.

**STEP 4: Update moveDown()**

|  |
| --- |
| int nextY = y + speed;  if (!pixelPerfectCollision(currentImage, x, nextY, playerWidth, playerHeight,  stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight)) {  y = nextY;  } |

**Where to place:**  
add this to your existing moveDown() method. Remove **y += speed;** and paste the code

**Explanation:**

* int nextY = y + speed; → calculates new Y position moving down.
* Same collision check → prevents walking into stone.
* Animation cycle with d1...d6 (down sprites).
* Calls repaint() to refresh the screen.

**STEP 5: Update moveLeft()**

|  |
| --- |
| public void moveLeft() {  int nextX = x - speed;  if (!pixelPerfectCollision(currentImage, nextX, y, playerWidth, playerHeight,  stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight)) {  x = nextX;  }  repaint();  } |

**Replace** your empty moveLeft().

**Explanation:**

* int nextX = x - speed; → calculates new X if moving left.
* Checks if movement would collide.
* If safe → update x.
* Redraw with repaint().

**Explanation:**

* int nextX = x + speed; → calculates new X if moving right.
* Same collision check.
* Updates x if no collision.
* Redraws after movement.

**STEP 6: Update moveRight()**

|  |
| --- |
| public void moveRight() {  int nextX = x + speed;  if (!pixelPerfectCollision(currentImage, nextX, y, playerWidth, playerHeight,  stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight)) {  x = nextX;  }  repaint();  } |

**Replace** your empty moveRight().

**Explanation:**

* @Override → ensures we override JPanel’s paint method.
* super.paintComponent(g); → clears screen before redrawing.
* g.drawImage(stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight, null); → draws the stone at (stoneX, stoneY) with scaling.
* g.drawImage(currentImage, x, y, playerWidth, playerHeight, null); → draws the player sprite scaled.
* if (stoneWalk != null) → prevents crash if image didn’t load.

**STEP 7: Update paintComponent()**

|  |
| --- |
| // draw stone with scalable size  if (stoneWalk != null) {  g.drawImage(stoneWalk, stoneX, stoneY, stoneWidth, stoneHeight, null);  }  // draw player with scalable size  if (currentImage != null) {  g.drawImage(currentImage, x, y, playerWidth, playerHeight, null);  } |

**Replace** your existing paintComponent(Graphics g).

**Chapter 5: Hitboxes & Chasing Enemies**

**Intro Blurb**

So far, your player only collides with objects. Now, let’s add a **hitbox detection system**. When you walk into an invisible **detection zone** around the stone, it wakes up and starts **chasing your character**. This is your first taste of enemy AI!

**Step-by-Step Coding**

**STEP 1: Create Detection Hitbox**

|  |
| --- |
| // Create detection box around the stone (bigger than sprite)  private Rectangle getStoneDetectionBox() {  return new Rectangle(stoneX - 100, stoneY - 100, 300, 300);  } |

**Where to place:**  
Add this as a **new method** under your helper functions (below your collision function is fine).

**STEP 2: Check Player vs Detection Box**

|  |
| --- |
| private void checkChasing() {  Rectangle playerHitbox = new Rectangle(x, y, playerWidth, playerHeight);  Rectangle stoneDetectionBox = getStoneDetectionBox();  if (stoneDetectionBox.intersects(playerHitbox)) {  chasing = true; // start chasing  } else {  chasing = false; // idle again  }  } |

**Where to place:**  
Add this method **below getStoneDetectionBox()**.  
We’ll call it inside actionPerformed() (the game loop).

**STEP 3: Make Stonewalk Follow Player**

|  |
| --- |
| private void updateStonewalk() {  if (chasing) {  if (x > stoneX) stoneX += 2; // move right  if (x < stoneX) stoneX -= 2; // move left  if (y > stoneY) stoneY += 2; // move down  if (y < stoneY) stoneY -= 2; // move up  }  } |

**Where to place:**  
Put this **below checkChasing()**.  
  
**Step 4: Add Timer & Implement ActionListener**

|  |
| --- |
| import java.awt.event.ActionEvent;  import java.awt.event.ActionListener;  import javax.swing.Timer; |

Change class definition so GamePanel **implements ActionListener**:

|  |
| --- |
| public class GamePanel extends JPanel implements ActionListener { |

**Step 5: Add Timer in Constructor**

Inside your GamePanel() constructor, add this:

|  |
| --- |
| Timer timer = new Timer(16, this); // ~60 FPS  timer.start(); |

**Where to place:** at the **end of the constructor**, after addKeyListener(new KeyHandler(this));.

**Step 6: Add actionPerformed()**

At the bottom of your class, add:

|  |
| --- |
| @Override  public void actionPerformed(ActionEvent e) {  checkChasing(); // NEW: check if player is inside detection zone  updateStonewalk(); // NEW: update stonewalk movement  repaint(); // redraw screen  } |

**Step 7: Add Variable for chasing**

|  |
| --- |
| private boolean chasing = false; |

Where to place: Above **private int upFrame = 0;**

Step 8: **Add actionPerformed()**

At the bottom of your class, add:

|  |
| --- |
| @Override  public void actionPerformed(ActionEvent e) {  checkChasing(); // NEW: check if player is inside detection zone  updateStonewalk(); // NEW: update stonewalk movement  repaint(); // redraw screen  } |

**Explanation of Each Line**

* Timer timer = new Timer(16, this); → runs every 16 ms (~60 frames per second).
* this → means the GamePanel is the listener, so its actionPerformed() will be called.
* timer.start(); → starts the game loop ticking.
* @Override public void actionPerformed(ActionEvent e) → the loop callback.
* checkChasing(); → checks if player entered the detection box.
* updateStonewalk(); → moves the stone closer if chasing is true.
* repaint(); → redraws everything with new positions.

**STEP 9: Draw Hitbox for Debugging**

Inside your paintComponent(Graphics g):

|  |
| --- |
| // Debug: draw detection box in red  g.setColor(Color.RED);  Rectangle detectBox = getStoneDetectionBox();  g.drawRect(detectBox.x, detectBox.y, detectBox.width, detectBox.height); |

**Step 10: Add variables at the top**

Right after your other variables (inside the class but **before the constructor**):

|  |
| --- |
| // Enemy behavior settings  private int enemySpeed = 2; // how fast stone follows the player  private int detectRange = 100; // how far detection box extends from stone |

**Step 11: Use these variables in methods**

**a) Replace hard-coded speed (2) in updateStonewalk():**

|  |
| --- |
| private void updateStonewalk() {  if (chasing) {  if (x > stoneX) stoneX += enemySpeed; // move right  if (x < stoneX) stoneX -= enemySpeed; // move left  if (y > stoneY) stoneY += enemySpeed; // move down  if (y < stoneY) stoneY -= enemySpeed; // move up  }  } |

**b) Replace detection box (fixed 100) in getStoneDetectionBox():**

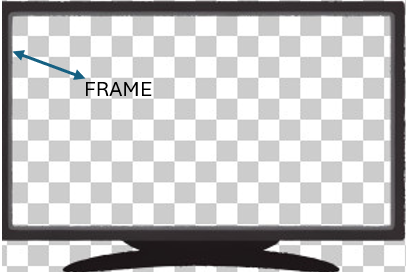
|  |
| --- |
| private Rectangle getStoneDetectionBox() {  return new Rectangle(  stoneX - detectRange,  stoneY - detectRange,  stoneWidth + detectRange \* 2,  stoneHeight + detectRange \* 2  );  } |

**Step 12: How to change them**

Now you can **tweak values** at the top of the file:

|  |
| --- |
| private int enemySpeed = 5; // make enemy faster  private int detectRange = 200; // make hitbox larger |

**FINAL TERM**

**Introduction**

In every Java game, the very first step is to **set up a game window** where everything happens — from **showing the background** to **drawing the player**, **enemies**, and **health bars**. In Java, we usually create this window using **Swing’s JFrame class**.

Think of JFrame as the **canvas frame** that holds your game world. Just like a TV screen holds a movie, the JFrame will hold your game panel, graphics, and user interface.

The code you wrote is the **foundation** of your game project:

|  |
| --- |
| import javax.swing.JFrame;  public class Main {      public static void main(String[] args) {          System.out.println("BSIT Game Starting...");          JFrame frame = new JFrame("1000 Years War");            frame.pack();          frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);          frame.setResizable(false);          frame.setLocationRelativeTo(null);          frame.setVisible(true);      }  } |

1. **import javax.swing.JFrame;**

- This **imports the JFrame class** from the javax.swing package.

2. **public class Main { }**

- Defines a **public class** named Main.  
- In Java, every program must be inside a class. Here, Main is your **entry point class**.

3. **public static void main(String[] args) {}**

- This is the **main method**, where your program starts running.

- public: Accessible from anywhere.

- static: Can run without creating an object of Main.

- void: Does not return anything.

- String[] args: Accepts **command-line arguments** (you’re not using them here, but they can pass extra instructions when starting the program).

**4. System.out.println("BSIT Game Starting...");**

- Prints the text "BSIT Game Starting..." to the **console/terminal**.

- Useful for debugging or letting the player know the game is loading.

**5. JFrame frame = new JFrame("1000 Years War");**

- Creates a **new window (JFrame)** with the title **"1000 Years War"**.

- frame is the variable holding the window object.

6. frame.pack();

- Adjusts the window size to **fit its components** (right now, you haven’t added components, so it makes a tiny window).

- Later, if you add a GamePanel, pack() makes the window the right size for that panel.

**7. frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);**

- Tells the program what to do when you click the **X button** on the window.

- EXIT\_ON\_CLOSE means: close the window **and stop the program**.

**8. frame.setResizable(false);**

- Prevents the user from resizing the window by dragging the edges.

- This is common in games (to keep a fixed resolution).

**9. frame.setLocationRelativeTo(null);**

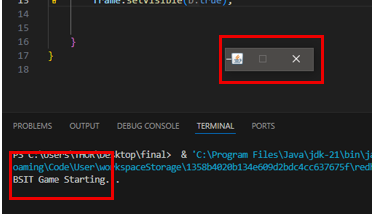
- Centers the window on the screen.

- If you pass null, Java automatically places the window in the middle of the screen.

**10. frame.setVisible(true);**

- Makes the window **appear on screen**.

- By default, a new JFrame is invisible until you call this.

if we run the program this will be your output:

Now if we want to resize the frame we will make changes on Main.java and create new Class Called GamePanel.java

|  |
| --- |
| import javax.swing.JFrame;  public class Main {      public static void main(String[] args) {          System.out.println("BSIT Game Starting...");          JFrame frame = new JFrame("1000 Years War");          GamePanel panel = new GamePanel(); // New          frame.add(panel); // New          frame.pack();          frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);          frame.setResizable(false);          frame.setLocationRelativeTo(null);          frame.setVisible(true);      }  } |

**11. GamePanel panel = new GamePanel();**

- Here you are **creating an object** of the GamePanel class.

- GamePanel is where you Draw shapes, images, or sprites.

**12. frame.add(panel);**

- This tells the JFrame (your game window) to **attach the panel** to itself.

- By adding panel you’re telling “*Hey, display this panel inside you — that’s my game screen.*”

Now Lets Create the **GamePanel.Java**

|  |
| --- |
| import javax.swing.\*;  import java.awt.\*;  public class GamePanel extends JPanel {  public static final int WIDTH = 800;      public static final int HEIGHT = 600;      public GamePanel() {          this.setPreferredSize(new Dimension(WIDTH, HEIGHT));          this.setBackground(Color.WHITE);         this.setDoubleBuffered(true);      }      @Override      protected void paintComponent(Graphics g) {          super.paintComponent(g);      }  } |

**13. import javax.swing.\*; and import java.awt.\*;**

- javax.swing.\*: Lets you use **Swing components** like JPanel.

- java.awt.\*: Lets you use **drawing tools** (colors, graphics, dimensions).

**14. public class GamePanel extends JPanel { }**

- Defines your class **GamePanel**.

- extends JPanel → You are creating a **custom panel** that inherits all behavior of JPanel.

- This means GamePanel *is a panel*, but you can **customize** it (size, background, graphics, etc.).

**15. public static final int WIDTH = 800; and public static final int HEIGHT = 600;**

- Declares constants for the **game screen size**:

\* Width = **800 pixels**

**\*** Height = 600 pixels

- public static final makes them **constants** (unchangeable values).

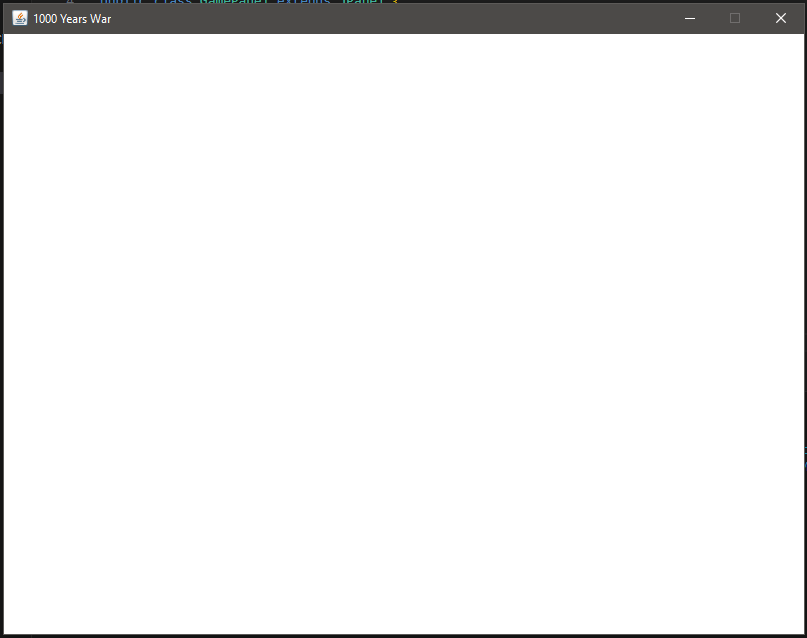
**16. public GamePanel() { }**

- This is the **constructor** → runs when you create a new GamePanel() in Main.

**17. this.setPreferredSize(new Dimension(WIDTH, HEIGHT));**

- Tells Swing: *“I want my panel to be 800 × 600.”*

- This works with frame.pack() (it resizes the window to fit this panel).

**18. this.setBackground(Color.WHITE);**

 JFrame frame = new JFrame("1000 Years War");

- Sets the background color of your game panel to **white**.

  GamePanel panel = new GamePanel();

- Later, you can change it

**19. this.setDoubleBuffered(true);**

- **Double buffering** helps prevent flickering.

**20. protected void paintComponent(Graphics g) { }**

- is where **all your drawing happens**.

**21. super.paintComponent(g);**

this.setPreferredSize(new Dimension(WIDTH, HEIGHT));

- Clears the old frame first (so you don’t paint on top of old drawings).

**Adding Map1**

**Add A new Class name Map1.java**

**22. Adding Libraries**

|  |
| --- |
| import java.awt.\*;  import java.awt.image.BufferedImage;  import javax.imageio.ImageIO;  import java.io.File;  import java.io.IOException; |

* java.awt.\*: Gives you Graphics for drawing.
* java.awt.image.BufferedImage: Represents an image in memory.
* javax.imageio.ImageIO: Used to **read/write images** (PNG, JPG, etc.).
* java.io.File: Lets you open a file from your computer.
* java.io.IOException: Handles errors if something goes wrong (like missing image).

**23. public class Map1 { }**

* This defines a new class Map1.
* It will handle **loading and drawing the first map** in your game.

**24. private BufferedImage background;**

* background is a **BufferedImage** object.
* It stores the map image in memory so it can be drawn later.
* private means only this class can directly access it.

**25. Constructor — load the map**

|  |
| --- |
| public Map1() {  try {  background = ImageIO.read(new File("C:\\Users\\THOR\\Desktop\\final\\map1\\map1.png"));  } catch (IOException e) {  e.printStackTrace();  System.out.println("Map1 image not found!");  }  } |

3. This runs when you create a new Map1() object.

2. ImageIO.read(new File(...)) → Opens the file at the given path and loads it as a BufferedImage.

3. If the file doesn’t exist, or can’t be read:

* IOException is caught.
* e.printStackTrace(); prints the error.
* "Map1 image not found!" message is shown so you know what went wrong.

Note: This ensures your game **won’t crash** if the map image is missing — it just warns you.

**26. Draw method**

|  |
| --- |
| public void draw(Graphics g, int width, int height) {  if (background != null) {  g.drawImage(background, 0, 0, width, height, null);  }  } |

1. public void draw(...) → A method to paint the map onto the screen.

2. Parameters:

* Graphics g: The graphics context used to draw.
* width, height: How big you want to draw the image.

3. if (background != null) → Only draw if the image actually loaded.

4. g.drawImage(background, 0, 0, width, height, null);

* Draws the background image.
* Positioned at **(0, 0)** (top-left corner).
* Scaled to fit the given width × height.
* null means no special observer is needed.

Note: This makes your map **stretch or shrink** to exactly match your game window size.

**Next step: You’ll want to use this in your GamePanel by:**

**27. private Map1 map1;**

* Creates a **reference variable** named map1.
* Type is Map1 (your custom class that loads and draws the map).
* private → Only GamePanel can directly use it.
* At this point, it’s just a variable — not yet an actual object.
* Place with other variables

Note: Think of it like saying: *“I’m reserving space to store a map inside this panel.”*

**28. map1 = new Map1();**

* Here you actually **create the map object** using the Map1 constructor.
* This will trigger the code in Map1() → it tries to load your background image (map1.png).
* Now map1 holds that image in memory, ready to be drawn.
* Place it under **constructor**  public GamePanel()

Note: Without this line, map1 would be null and calling map1.draw(...) would crash the game.

**29. map1.draw(g, WIDTH, HEIGHT);**

* Calls the draw() method from your Map1 class.
* g = the **Graphics object** used for drawing on the panel.
* WIDTH, HEIGHT = the panel size (800×600).
* Inside Map1.draw(), the background image is drawn starting at (0,0) and scaled to **fill the whole panel**.
* Place it Inside paintComponent

Note: This is what actually makes your map **appear on screen** as the background.

**Adding Main Character**

**Note: use the library on Map1 class in this class.**

**Add A new Class name CharacterLoad.java**

**30. private BufferedImage character;**

* Stores the **character sprite** (image of your player).
* BufferedImage lets you hold and manipulate images in memory.

**31. private int x; and private int y;**

* These represent the **top-left corner** of your character on the screen.
* x → horizontal position.
* y → vertical position.

Example: (100, 200) means the character will appear 100 pixels from the left, 200 pixels from the top.

**32. private int width = 150; and private int height = 100;**

* Defines how **big** you want the sprite to appear on screen.
* Even if the original image file is larger/smaller, Java will **resize it** when drawing.
* This lets you fit the sprite to your game world scale.

**33. Constructor**

|  |
| --- |
| public CharacterLoad(int startX, int startY) {  this.x = startX;  this.y = startY;  try {  character = ImageIO.read(new File("C:\\Users\\THOR\\Desktop\\final\\walkright\\right1.png"));  } catch (IOException e) {  e.printStackTrace();  System.out.println("Character image not found!");  }  } |

* Called when you create a new character, e.g. new CharacterLoad(100, 200);.
* this.x = startX; this.y = startY; → Places the character at the starting position.
* ImageIO.read(new File(...)) → Loads your sprite image into memory.
* If the file is missing, it prints an error message but doesn’t crash the game.

Note: This makes your character **spawn at a given position with an image loaded**.

**34. Draw method**

|  |
| --- |
| public void draw(Graphics g) {  if (character != null) {  g.drawImage(character, x, y, width, height, null);  }  } |

* Draws the character on screen using the Graphics object.
* g.drawImage(...): paints the sprite at (x, y) and scales it to (width, height).
* if (character != null) → Only draw if the image was loaded successfully.

Note: This is called inside paintComponent(Graphics g) in your GamePanel.

**35. Movement method**

|  |
| --- |
| public void move(int dx, int dy) {  x += dx;  y += dy;  } |

* Changes the character’s position.
* dx = change in x (positive = right, negative = left).
* dy = change in y (positive = down, negative = up).
* Each time this method is called, the character shifts by that amount.

Example:

* move(5, 0) → move 5 pixels right.
* move(0, -5) → move 5 pixels up.

Next step: To actually see the character **on top of your map**, you’d add in your GamePanel.paintComponent

**36.  private CharacterLoad character;**

* Creates a **reference variable** named map1.

**37.  character = new CharacterLoad(100, 100);**

* Creates a **new character object** starting at position (100, 100)

**38. character.draw(g);**

* Calls the draw(Graphics g) method of CharacterLoad.

**Adding Movement arrows**

**Add A new Class name KeyHandler.java**

**39. public class KeyHandler implements KeyListener { }**

* Defines the class KeyHandler.
* implements KeyListener → Means this class must provide the three required methods of the KeyListener interface:
  + keyTyped(KeyEvent e)
  + keyPressed(KeyEvent e)
  + keyReleased(KeyEvent e)

Note: This makes KeyHandler a **keyboard listener** that reacts whenever keys are pressed, typed, or released.

**40. private boolean upPressed, downPressed, leftPressed, rightPressed;**

* These variables track whether each movement key is currently being held down.
* Example:
  + If you press the ↑ arrow, upPressed = true.
  + When you release it, upPressed = false.

Note: This allows continuous movement (not just single steps).

**41. Empty method (required by interface)**

|  |
| --- |
| @Override  public void keyTyped(KeyEvent e) {} |

* keyTyped is required by KeyListener, but you don’t need it here.
* Left empty → means typed keys (like characters) are ignored.

**42. When a key is pressed**

|  |
| --- |
| @Override  public void keyPressed(KeyEvent e) {  int code = e.getKeyCode();  if (code == KeyEvent.VK\_UP) upPressed = true;  if (code == KeyEvent.VK\_DOWN) downPressed = true;  if (code == KeyEvent.VK\_LEFT) leftPressed = true;  if (code == KeyEvent.VK\_RIGHT) rightPressed = true;  } |

* Called automatically when a key is pressed down.
* e.getKeyCode() → Gets the numeric code of the pressed key.
* KeyEvent.VK\_UP, KeyEvent.VK\_DOWN, etc. → Special constants for arrow keys.
* Sets the corresponding boolean to **true** when the key is pressed.

Note: Example: If you press the Right Arrow, rightPressed = true.

**43. When a key is released**

|  |
| --- |
| @Override  public void keyReleased(KeyEvent e) {  int code = e.getKeyCode();  if (code == KeyEvent.VK\_UP) upPressed = false;  if (code == KeyEvent.VK\_DOWN) downPressed = false;  if (code == KeyEvent.VK\_LEFT) leftPressed = false;  if (code == KeyEvent.VK\_RIGHT) rightPressed = false;  } |

* Called automatically when you let go of a key.
* Sets the corresponding boolean to **false**.

Note: Example: When you release the Right Arrow, rightPressed = false.

**44. Getters (to check from GamePanel)**

|  |
| --- |
| public boolean isUpPressed() { return upPressed; }  public boolean isDownPressed() { return downPressed; }  public boolean isLeftPressed() { return leftPressed; }  public boolean isRightPressed() { return rightPressed; } |

- These let other classes (like GamePanel) check which keys are being held down.

Next Step: You’ll need to **attach this KeyHandler to your GamePanel** like so:

**43. implements Runnable (class header)**

|  |
| --- |
| public class GamePanel extends JPanel implements Runnable { } |

* This means the GamePanel class promises to implement the Runnable interface.
* Runnable requires a run() method → this is where we’ll write the game loop.
* Without Runnable, you can’t run this panel in a thread.

**44. private KeyHandler keyHandler;**

* Stores an instance of your KeyHandler (the class you wrote to detect keyboard input).
* Lets your game panel check which keys are pressed.
* Place under  **private CharacterLoad character;**

**45. private Thread gameThread;**

* **This is the separate game loop thread.**
* **Instead of freezing the GUI, your update & rendering happens in the background so the game runs smoothly at ~60 FPS.**

**46. Inside constructor**

|  |
| --- |
| keyHandler = new KeyHandler();  this.addKeyListener(keyHandler);  this.setFocusable(true); // new |

* **keyHandler = new KeyHandler();** → Creates the input manager.
* **this.addKeyListener(keyHandler);** → Tells the panel to listen for key presses.
* **this.setFocusable(true);** → Makes sure this panel can receive keyboard focus, otherwise key events won’t work.
* Place under  **this.setDoubleBuffered(true);**

Note: Without these three lines, pressing arrow keys wouldn’t move your character.

**47. Game loop thread**

|  |
| --- |
| gameThread = new Thread(this);  gameThread.start(); |

* **new Thread(this)** → Creates a new thread and tells it to run this panel’s run() method.
* **gameThread.start()** → Actually starts the thread and begins executing the game loop.
* Place under **character = new CharacterLoad(100, 100);**

**48. The run() method**

|  |
| --- |
| @Override  public void run() { // new  while (true) {  update();  repaint();  try {  Thread.sleep(16); // ~60 FPS  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

* Runs **forever** in a loop (while (true) = infinite).
* Calls update() (game logic: movement, collisions, etc.).
* Calls repaint() (redraws everything on the screen).
* Sleeps for 16 ms (~60 FPS).

Note: This is your **game engine heartbeat**.

**49. The update() method**

|  |
| --- |
| public void update() {  int speed = 4;  if (keyHandler.isUpPressed()) character.move(0, -speed);  if (keyHandler.isDownPressed()) character.move(0, speed);  if (keyHandler.isLeftPressed()) character.move(-speed, 0);  if (keyHandler.isRightPressed()) character.move(speed, 0);  } |

* Checks which keys are pressed.
* Moves the character in the correct direction by a fixed **speed (4 pixels per frame)**.
* Example: If UP arrow is pressed → character moves up on the screen.

**RUN MULTIPLE SPRITE**

**Add this updated code on CharacterLoad.java**

**50. Lest add new Variables**

|  |
| --- |
| private int frameIndex = 0;  private int frameDelay = 10;  private int frameCount = 0;  private BufferedImage[] rightSprites; |

**51.  rightSprites = new BufferedImage[5];**

* Load all 5 right-walk sprites
* Place under **this.y = startY;** inside the **constructor**

**52. Update the try…catch**

|  |
| --- |
| try {  for (int i = 0; i < 5; i++) {  rightSprites[i] = ImageIO.read(  new File("C:\\Users\\THOR\\Desktop\\final\\walkright\\right" + (i + 1) + ".png")  );  }  } catch (IOException e) {  e.printStackTrace();  System.out.println("Error loading right walk sprites!");  } |

* Place it below  **rightSprites = new BufferedImage[5];** inside **Constructor**

**53. update animation if moving right**

|  |
| --- |
| public void update(boolean movingRight) {          if (movingRight) {              frameCount++;              if (frameCount >= frameDelay) {                  frameIndex = (frameIndex + 1) % rightSprites.length;                  frameCount = 0;              }          } else {              frameIndex = 0; // reset to right1 when idle          }      } |

* Place it below  **closing curly bracket of constructor**

1. **if (movingRight)**

* Checks if the character is currently moving right.
* If true → advance the animation frames.
* If false → reset the animation to the first frame (idle).

2. **frameCount++**

* Counts how many "ticks" (updates) have passed.
* Prevents the animation from going too fast.

3. **if (frameCount >= frameDelay)**

* Only after enough ticks (frameDelay = 10) should the frame change.
* This slows the animation to look natural.

4. **frameIndex = (frameIndex + 1) % rightSprites.length;**

* Moves to the next sprite image.
* % rightSprites.length makes it loop back to 0 when it reaches the last sprite (like cycling through frames).

5. **frameCount = 0;**

* Reset tick counter after switching frames.

6. **else { frameIndex = 0; }**

* If the character is NOT moving right, always show the first frame (right1.png) = idle stance.

**54. Update draw method**

|  |
| --- |
| public void draw(Graphics g) {          if (rightSprites[frameIndex] != null) {              g.drawImage(rightSprites[frameIndex], x, y, width, height, null);          }      } |

1. **rightSprites[frameIndex]**

* Picks the correct sprite image based on the current animation frame.
* Example:
  + frameIndex = 0 → right1.png
  + frameIndex = 1 → right2.png
  + etc.

2. **g.drawImage(...)**

* Draws the sprite at (x, y) with scaling (width, height).

3. **if (rightSprites[frameIndex] != null)**

* Safety check to avoid errors in case the image failed to load.  
    
  Next Step: You’ll need to **attach this to your GamePanel**

**55. Update GamePanel.java → update()**

|  |
| --- |
| public void update() {          int speed = 4;           boolean movingRight = false; // new          if (keyHandler.isUpPressed()) character.move(0, -speed);          if (keyHandler.isDownPressed()) character.move(0, speed);          if (keyHandler.isLeftPressed()) character.move(-speed, 0);          if (keyHandler.isRightPressed()) {              character.move(speed, 0);              movingRight = true;          }          character.update(movingRight);      } |

**Down Movement**

**56. private BufferedImage[] downSprites;**

* Creates an **array of images** that will hold the character’s **walking down animation frames**.

**57. Inside Constructor — Loading Down Walk Sprites**

|  |
| --- |
| downSprites = new BufferedImage[3];  try {  for (int i = 0; i < 3; i++) {  downSprites[i] = ImageIO.read(  new File("C:\\Users\\THOR\\Desktop\\final\\downwalk\\down" + (i + 1) + ".png")  );  }  } catch (IOException e) {  e.printStackTrace();  System.out.println("Error loading down walk sprites!");  } |

**58. update(boolean moving, String direction)**

|  |
| --- |
| public void update(boolean moving, String direction) {  if (moving) {  currentDirection = direction;  frameCount++;  if (frameCount >= frameDelay) {  if (direction.equals("right")) {  frameIndex = (frameIndex + 1) % rightSprites.length;  } else if (direction.equals("down")) {  frameIndex = (frameIndex + 1) % downSprites.length;  }  frameCount = 0;  }  } else {  frameIndex = 0;  }  } |

* **Inputs:**
  + moving → tells whether the character is walking or standing still.
  + direction → tells which way the character is moving ("right" or "down").
* **If moving:**
  + currentDirection = direction;
    - Save the direction so draw() knows which sprites to use.
  + frameCount++;
    - Increments a counter each frame.
  + if (frameCount >= frameDelay)
    - Once enough frames pass (e.g., 10 ticks), switch animation.
  + frameIndex = (frameIndex + 1) % spriteArray.length;
    - Move to the next animation frame.
    - % ensures looping back to frame 0 at the end.
    - Uses the correct sprite set (rightSprites or downSprites) depending on direction.
  + Reset frameCount = 0;.
* **If not moving:**
  + Reset animation to frame 0 → idle pose.

Note: This ensures smooth animation at a controlled speed, and resets to idle when you stop.

**59. draw(Graphics g)**

|  |
| --- |
| public void draw(Graphics g) {  BufferedImage currentFrame = null;  if (currentDirection.equals("right")) {  currentFrame = rightSprites[frameIndex];  } else if (currentDirection.equals("down")) {  currentFrame = downSprites[frameIndex];  }  if (currentFrame != null) {  g.drawImage(currentFrame, x, y, width, height, null);  }  } |

* Chooses the **current frame image** depending on the direction the character last moved.
  + If facing right → pick from rightSprites.
  + If facing down → pick from downSprites.
* Uses frameIndex to grab the correct frame (e.g., down1.png, down2.png, etc.).
* Finally, draws the chosen sprite at (x, y) with scaling (width, height).

Note: The character **keeps facing the last direction** even if idle, since currentDirection is stored.

**60. Update GamePanel update() method**

|  |
| --- |
| boolean moving = false;  String direction = "";        if (keyHandler.isRightPressed()) {          character.move(speed, 0);          moving = true;          direction = "right";      }      if (keyHandler.isDownPressed()) {          character.move(0, speed);          moving = true;          direction = "down";      } |

* Add the variables below **int speed** and update the if statement down press and right press

**UP, LEFT Movement**

**61. Variable for up and left movement**

|  |
| --- |
| private BufferedImage[] upSprites;  private BufferedImage[] leftSprites; |

**62. Load the Sprite with try…catch**

|  |
| --- |
| // Load up walk (3 frames)          upSprites = new BufferedImage[3];          try {              for (int i = 0; i < 3; i++) {                  upSprites[i] = ImageIO.read(                      new File("C:\\Users\\THOR\\Desktop\\final\\upwalk\\up" + (i + 1) + ".png")                  );              }          } catch (IOException e) {              e.printStackTrace();              System.out.println("Error loading up walk sprites!");          }          // Load left walk (5 frames)          leftSprites = new BufferedImage[5];          try {              for (int i = 0; i < 5; i++) {                  leftSprites[i] = ImageIO.read(                      new File("C:\\Users\\THOR\\Desktop\\final\\leftwalk\\left" + (i + 1) + ".png")                  );              }          } catch (IOException e) {              e.printStackTrace();              System.out.println("Error loading left walk sprites!");          } |

**63. Update animation**

|  |
| --- |
| public void update(boolean moving, String direction) {          if (moving) {              currentDirection = direction;              frameCount++;              if (frameCount >= frameDelay) {                  switch (direction) {                      case "right":                          frameIndex = (frameIndex + 1) % rightSprites.length;                          break;                      case "down":                          frameIndex = (frameIndex + 1) % downSprites.length;                          break;                      case "up":                          frameIndex = (frameIndex + 1) % upSprites.length;                          break;                      case "left":                          frameIndex = (frameIndex + 1) % leftSprites.length;                          break;                  }                  frameCount = 0;              }          } else {              frameIndex = 0;          }      } |

**64. Update Draw method**

|  |
| --- |
| public void draw(Graphics g) {          BufferedImage currentFrame = null;          switch (currentDirection) {              case "right":                  currentFrame = rightSprites[frameIndex];                  break;              case "down":                  currentFrame = downSprites[frameIndex];                  break;              case "up":                  currentFrame = upSprites[frameIndex];                  break;              case "left":                  currentFrame = leftSprites[frameIndex];                  break;          }           if (currentFrame != null) {              g.drawImage(currentFrame, x, y, width, height, null);          }      } |

**GAMEPANEL UPDATE**

**65. Update the update() method**

|  |
| --- |
| if (keyHandler.isUpPressed()) {          character.move(0, -speed);          moving = true;          direction = "up";      }         if (keyHandler.isLeftPressed()) {          character.move(-speed, 0);          moving = true;          direction = "left"; } |

**Check Every Pixel Position**

**66. Imports**

|  |
| --- |
| import java.awt.event.MouseAdapter;  import java.awt.event.MouseEvent; |

* MouseAdapter is a helper class that already implements the MouseListener interface, so you don’t need to implement all the methods yourself.
* MouseEvent represents a mouse action (like a click, press, or release).

**67. Define Class**

|  |
| --- |
| public class PixelPosition extends MouseAdapter { } |

* It **extends** MouseAdapter, which means it can listen for mouse events like clicks without writing extra unused methods.

**68. variables**

|  |
| --- |
| private int mouseX;  private int mouseY;  private boolean clicked; |

* mouseX → stores the **X coordinate** of the mouse click.
* mouseY → stores the **Y coordinate** of the mouse click.
* clicked → true when the mouse has been clicked, false otherwise.

**68. Method that runs automatically**

|  |
| --- |
| @Override  public void mouseClicked(MouseEvent e) {  mouseX = e.getX();  mouseY = e.getY();  clicked = true;  System.out.println("Pixel clicked at: X=" + mouseX + ", Y=" + mouseY);  } |

* @Override → means we are **overriding** the default method from MouseAdapter.
* e.getX() → gets the X coordinate of the mouse click.
* e.getY() → gets the Y coordinate of the mouse click.
* clicked = true; → marks that a click happened.
* System.out.println(...) → prints the coordinates to the console so you can see where you clicked.

**69. getter methods**.

|  |
| --- |
| // Getters  public int getMouseX() {  return mouseX;  }  public int getMouseY() {  return mouseY;  }  public boolean isClicked() {  return clicked;  } |

Other classes (like GamePanel) can call them to check:

* getMouseX() → the last X position clicked.
* getMouseY() → the last Y position clicked.
* isClicked() → whether a click has occurred since last reset.

**70. reset the click status**

|  |
| --- |
| public void resetClick() {  clicked = false;  } |

* After you process a click, you can call this so the program doesn’t keep thinking a click is active forever.

**USE IT IN GamePanel:**

**71. private PixelPosition pixelPosition;**

* Add this variable along with other variables on top

**72. inside constructor**

|  |
| --- |
| pixelPosition = new PixelPosition();  this.addMouseListener(pixelPosition); |