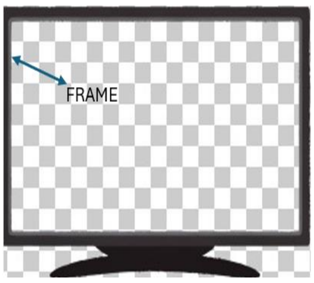
## Introduction



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);**

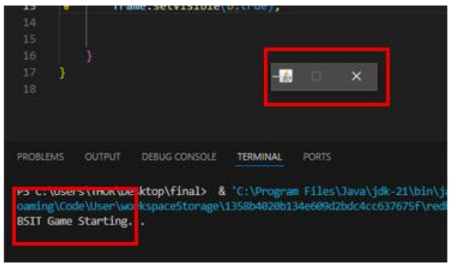
- Centers the window on the screen.

- 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.



```
            frame.setVisible(b:true);

        }
    }
```

```
PS C:\Users\THOR\Desktop\final>  & 'C:\Program Files\Java\jdk-21\bin\j
oaming\Code\User\workspaceStorage\1358b4020b134e609d2bdc4cc637675f\red
BSIT Game Starting...
```

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).

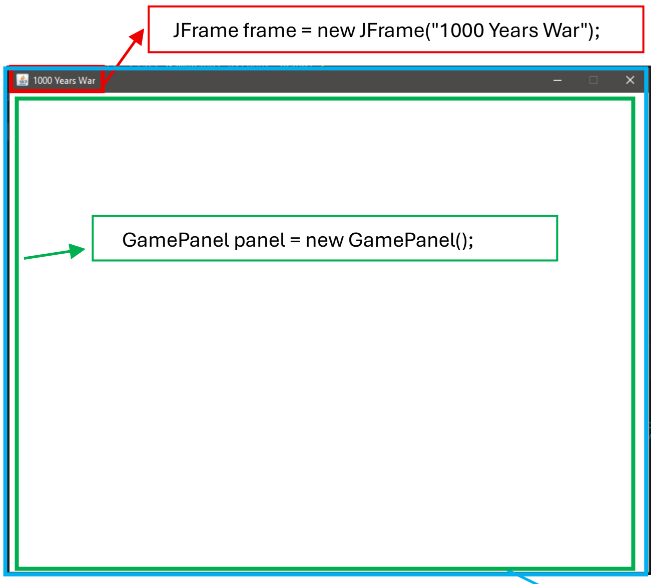
### 16. public GamePanel() { }

- 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).



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

GamePanel panel = new GamePanel();

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

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

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

- 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);**

- 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:**

1. currentDirection = direction;
    - Save the direction so draw() knows which sprites to use.
2. frameCount++;
    - Increments a counter each frame.
3. if (frameCount >= frameDelay)
    - Once enough frames pass (e.g., 10 ticks), switch animation.
4. 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.
5. 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:**

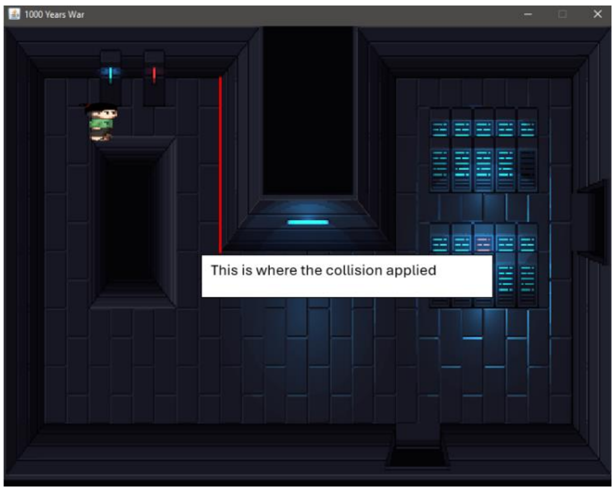
**71. private PixelPosition pixelPosition;**

- Add this variable along with other variables on top

**72. inside constructor**

```
pixelPosition = new PixelPosition();
this.addMouseListener(pixelPosition);
```

# Add Collision



This is where the collision applied

**Create your new class name Collision.java**

**73. import java.awt.Rectangle;**

- This imports the Rectangle class from the java.awt package.

- **Rectangle** is very useful for collision detection because it has built-in methods like .**intersects**() to check if two rectangles overlap.

**74. public class Collision { }**

- Defines a **public class** named Collision.
- This class will handle checking whether the player collides with certain obstacles (in this case, a wall).

**75. Variables for wall coordinates**

```
    private int x1 = 283;
    private int y1 = 69;
    private int x2 = 289;
    private int y2 = 284;
```

- These are the coordinates that define a wall.
- (x1, y1) is one corner of the wall, and (x2, y2) is the opposite corner.
- So instead of defining a wall with width and height directly, you're defining it by two points.
- Example: this represents a vertical wall between x=283–289 and y=69–284.

### 76. Checking Character collision in Boolean form

```
public boolean checkCollision(int charX, int charY, int charWidth, int charHeight) { }
```

- This method checks if the character is colliding with the wall.
- Parameters:
  - charX, charY: top-left position of the character.
  - charWidth, charHeight: the character's dimensions.

### 77. Character rectangle

```
    Rectangle charRect = new Rectangle(charX, charY, charWidth, charHeight);
```

- Creates a Rectangle representing the character's bounding box (the area it occupies on the screen).
- Now you can use this rectangle to check overlaps with other rectangles.

### 78. Wall rectangle

```
    int wallLeft = Math.min(x1, x2);
    int wallTop = Math.min(y1, y2);
    int wallWidth = Math.abs(x2 - x1);
    int wallHeight = Math.abs(y2 - y1);
```

- These lines calculate the **wall rectangle** based on the two corner coordinates (x1, y1 and x2, y2).
- wallLeft = Math.min(x1, x2) → ensures we always pick the smaller x as the left edge.
- wallTop = Math.min(y1, y2) → ensures we always pick the smaller y as the top edge.
- wallWidth = Math.abs(x2 - x1) → gets the positive width between the two x-coordinates.
- wallHeight = Math.abs(y2 - y1) → gets the positive height between the two y-coordinates.

This way, no matter how the coordinates are ordered, we always get a proper rectangle.

### 79. wall has at least 1px width/height

```
    if (wallWidth == 0) wallWidth = 1;
    if (wallHeight == 0) wallHeight = 1;
```

- Prevents the wall from having **zero width or height** (which would make it invisible and useless for collision).
- Example: if x1 == x2, the wall would have no width. By forcing it to at least 1px, you can still detect collisions properly.

### 80. Rectangle wallRect = new Rectangle(wallLeft, wallTop, wallWidth, wallHeight);

- Now a Rectangle object is created for the wall using the corrected coordinates, width, and height.
- This represents the physical area of the wall on the screen.

### 81.   return charRect.intersects(wallRect);

- This checks if the character's rectangle overlaps (collides) with the wall's rectangle.

- .intersects() returns true if they overlap, otherwise false.

**In short**:
The class defines a wall with two coordinates, converts both the wall and character into rectangles, and uses Rectangle.intersects() to check if they collide.

## Apply the collision in GamePanel.java

### 82. private Collision collision;

- It means your GamePanel has access to the collision detection system.
- Right now it's just declared — no object is created yet.

### 83. collision = new Collision();

- Inside the GamePanel constructor, this line **creates a new instance** of the Collision class.
- Now collision can be used to check if the player (character) collides with walls.

### 84. The update() method

```
int speed = 2;
boolean moving = false;
String direction = "";
```

- speed = 2 → how many pixels the character moves per frame.
- moving = false → tracks if the character is actually walking (used for animation).
- direction = "" → will store which way the character is moving ("up", "down", "left", "right").

### 85. Potential Position

```
int nextX = character.getX();
int nextY = character.getY();
```

- These represent the **character's potential new position** before moving.
- You don't move immediately — instead, you **test the new position first** to check for collisions.

### 86. Moving UP

```
if (keyHandler.isUpPressed()) {
   nextY -= speed;
   if (!collision.checkCollision(nextX, nextY, character.getWidth(), character.getHeight())) {
      character.setPosition(nextX, nextY);
   }
   moving = true;
   direction = "up";
}
```

- If the **UP arrow key** is pressed:
- nextY -= speed; → move the character upwards by decreasing the Y coordinate.
- Check with collision.checkCollision(...).
- If there's **no collision**, update the character's position with character.setPosition(...).
- Mark moving = true and set direction = "up" for animation.

### 87. Moving LEFT

```
if (keyHandler.isLeftPressed()) {
   nextX = character.getX() - speed;
   nextY = character.getY();
   if (!collision.checkCollision(nextX, nextY, character.getWidth(),
character.getHeight())) {
       character.setPosition(nextX, nextY);
   }
   moving = true;
   direction = "left";
}
```

- Same as UP, but decreases **X** instead of Y → moves left.
- Collision is checked before moving.

### 88. Moving RIGHT

```
if (keyHandler.isRightPressed()) {
   nextX = character.getX() + speed;
   nextY = character.getY();
   if (!collision.checkCollision(nextX, nextY, character.getWidth(),
character.getHeight())) {
       character.setPosition(nextX, nextY);
   }
   moving = true;
   direction = "right";
}
```

- Increases **X** → moves right.
- Collision check ensures you don't walk through walls.

### 89. Moving DOWN

```
if (keyHandler.isDownPressed()) {
   nextX = character.getX();
   nextY = character.getY() + speed;
   if (!collision.checkCollision(nextX, nextY, character.getWidth(),
character.getHeight())) {
       character.setPosition(nextX, nextY);
   }
   moving = true;
   direction = "down";
}
```

- Increases **Y** → moves down.
- Collision prevents passing through obstacles.

### 90. character.update(moving, direction);

- After movement and collision checks are done, the **character's animation is updated**.
- moving tells if the animation should play (walking) or reset (idle).
- direction tells which set of sprites (up, down, left, right) should be used.

**In short:**

- The Collision object ensures the player cannot move through defined walls.
- The update() method predicts the new position → checks collision → updates position only if safe.
- Animation is updated based on movement and direction.

# Manipulate Walls

## Update your Collision.java

### 91. New added Libraries

```
import java.util.ArrayList;
import java.util.List;
```

- ArrayList and List → used to store **many walls** instead of hardcoding a single one.

### 92. public class Collision { }

- Declares the Collision class, which will manage all collision detection in the game.

### 93. private List<Rectangle> walls;

- A list of Rectangle objects.
- Each rectangle represents a **wall or obstacle** in the game.

### 94. Constructor

```
public Collision() {
    walls = new ArrayList<>();

    walls.add(makeRectangle(283, 69, 289, 284));
}
```

- walls = new ArrayList<>(); → creates an empty list to hold walls.
- walls.add(...) → adds one wall to the list
- You can add more walls.add(...) calls here to build a map with multiple walls.

### 95. method to convert two points into a valid Rectangle.

```
private Rectangle makeRectangle(int x1, int y1, int x2, int y2) {
    int left = Math.min(x1, x2);
    int top = Math.min(y1, y2);
    int width = Math.abs(x2 - x1);
    int height = Math.abs(y2 - y1);

    if (width == 0) width = 1;
    if (height == 0) height = 1;

    return new Rectangle(left, top, width, height);
}
```

- (x1, y1) and (x2, y2) define two corners.
- Math.min → ensures you always pick the correct **top-left corner**.
- Math.abs → ensures width/height are positive.
- If width or height is 0, force it to at least 1 pixel (so the wall exists).
- Finally, returns a proper Rectangle.

This keeps the code clean because you don’t have to rewrite the math every time you add a wall.

### 96. Rectangle for the character’s current (or future) position

```
public boolean checkCollision(int charX, int charY, int charWidth, int charHeight) {
    Rectangle charRect = new Rectangle(charX, charY, charWidth, charHeight);
    for (Rectangle wall : walls) {
        if (charRect.intersects(wall)) {
            return true;
        }
    }
    return false;
}
```

- Creates a Rectangle for the character’s current (or future) position.
- Loops through **every wall** in the list.
- charRect.intersects(wall) checks if the character overlaps with that wall.
- If **any wall collides**, return true (collision found).
- If none intersect, return false (movement is safe).

### 97. Update Collision constructor for new wall

```
   // Second wall
    walls.add(makeRectangle(49, 61, 286, 63));

    // Third wall
    walls.add(makeRectangle(47, 63, 52, 519));
```

# ADD Enemy

### 98. imports.

```
import java.awt.*;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;
```

- java.awt.* → For graphics, shapes, and colors (like Graphics, Rectangle).
- BufferedImage → Stores an image in memory.
- ImageIO → Loads image files.
- File → Lets you point to a file on your computer.
- IOException → Error handling if the image can’t be loaded.

### 99. public class Enemy { }

- Defines a new **class** named Enemy. This represents your enemy object in the game.

### 100. Fixed position

```
private int x = 385, y = 391;
private int width = 100, height = 100;
```

- x and y → The **position** of the enemy on the screen.
- width and height → The **size** of the enemy when drawn.
- Right now, x and y are fixed, so the enemy **does not move**.

## 101. Variables

```
private BufferedImage[] enemySprites;
private int frameIndex = 0;
private int frameDelay = 15;  // Adjust for animation speed
private int frameCount = 0;
```

- enemySprites → An **array** of images (3 frames of your enemy).
- frameIndex → Keeps track of which frame is currently being shown.
- frameDelay → Controls how many updates must pass before switching to the next frame (bigger number = slower animation).
- frameCount → Counts updates until it reaches frameDelay.

## 102. constructor (Enemy())

```
public Enemy() {
    enemySprites = new BufferedImage[3];
    try {
        enemySprites[0] = ImageIO.read(new File("C:\\Users\\THOR\\Desktop\\final\\enemy\\tiktik.png"));
        enemySprites[1] = ImageIO.read(new File("C:\\Users\\THOR\\Desktop\\final\\enemy\\tiktik2.png"));
        enemySprites[2] = ImageIO.read(new File("C:\\Users\\THOR\\Desktop\\final\\enemy\\tiktik3.png"));
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error loading enemy sprites!");
    }
}
```

- This is the **constructor** (Enemy()), called when you create a new enemy.
- It creates an array of size 3 (enemySprites = new BufferedImage[3];).
- Then loads 3 images (3 animation frames) from your computer.
- If something goes wrong (e.g., file not found), it prints an error.

## 103. Updates the animation

```
public void update() {
    frameCount++;
    if (frameCount >= frameDelay) {
        frameIndex = (frameIndex + 1) % enemySprites.length;
        frameCount = 0;
    }
}
```

- **Updates the animation** (called every game tick).
- frameCount++ → increases the counter each update.
- When frameCount reaches frameDelay, it switches to the next frame (frameIndex + 1).
- % enemySprites.length makes it **loop back to 0** when it reaches the last frame.
- frameCount = 0; → resets counter so the next delay starts.
- This does **not move** the enemy, it only makes it animate while standing still.

## 104. Draws the current frame

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

- Draws the current frame of the enemy on the screen.
- g.drawImage(...) → paints the enemy at (x, y) with width and height.
- Uses the correct frame (frameIndex) of your sprite animation.

### 105. collision detection

```
public Rectangle getBounds() {
        return new Rectangle(x, y, width, height);
    }
```

- Creates a Rectangle around the enemy's position and size.
- Useful for **collision detection** (checking if the player hits the enemy).

### 106. In GamePanel Constructor:

```
enemy = new Enemy();
```

### 107. Update loop:

```
enemy.update();
```

### 108. Paint:

```
enemy.draw(g);
```