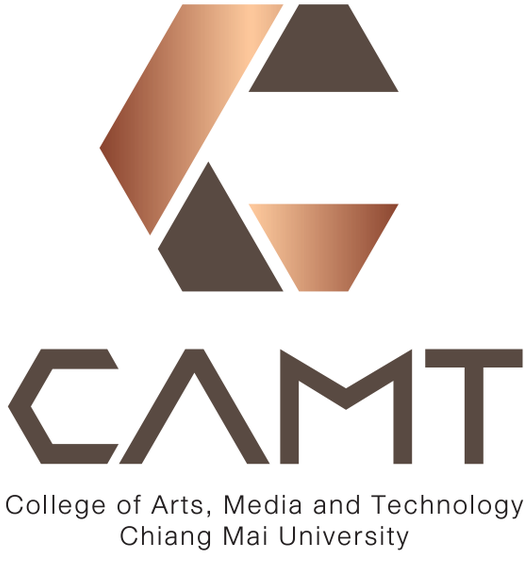
# SE 233 Advanced Programming

## Chapter IV Debugging




**Lect** Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

# Agenda

- Logging
- Debugging

# Fault and error

- Fault
  - A condition that causes the software to fail to perform its required function
- Error
  - Difference between Actual Output and Expected output

**They appears to be inevitable**

# Causes of error

- Data input error
- Wrong parameter
- Wrong variable
- Wrong operator
- Wrong operand
- Wrong computation
- Wrong algorithm
- Etc

These are bugs !

# Causes of error

- Data input error
- Wrong parameter
- Wrong variable
- Wrong operator
- Wrong operand
- Wrong computation
- Wrong algorithm
- Etc

Fixing them is called debugging

# When an error occurs

- Errors should happen in some places
- We have to find it

# How to localize the error ?

- **Probing** — Modify the code to show the value of some specific variables on the screen
- **Logging** — Chronological and systematic record of data processing events in a program
- **Debugging** — Tracing the program on the fly

# Probing

- Print to the screen
  - Simply add **System.out.println()** to probe the value of variables
  - Look on the screen
  - Add some code in the source code to observe the change in the behavior

# Probing

- What can be probed
  - Variable value
  - Return value

- Pros
  - Easy to start
  - Effective if the developer is familiar with it

# Probing — possible concerns

- Not easy to customize
- Reduction in the performance
- Time consuming
- Code used in probing needs to be removed after the error is fixed.
- What if we only want to see some important errors
- What if we also want to look at multiple variable's value at the same time?
- etc..

# Logging

- Logging is the **process** of writing log messages during the execution of a program to a central place
- This logging allows you to **report** and persist **error** and warning messages as well as info messages (e.g., runtime statistics) so that the messages can later be retrieved and **analyzed**.
- The object which performs the logging in applications is typically just called **Logger**.

# Logging

- Used during development to identify errors;
- Used during production for troubleshooting.

# Logging

- Used during development to identify errors;
- Used during production for troubleshooting.

- The logging assets are also used in the mining software repository process
  - analyzes the data generated during the software development to uncover interesting and actionable information about software systems and projects.

# Essential components

- Logging frameworks
- Logging levels
- Logger
- Appender
- Layout
- Configuration

# Logging frameworks

- A logging framework provides the necessary components to create and store log messages.
- These components include objects, methods, and other configurations.
- Java provides a built-in framework in the java.util.logging package.
- However, alternative third-party frameworks, such as Log4j and Slf4j appear to be more popular

# Slf4j

- Simple Logging Façade for Java (Slf4j)
- Simple façade/ abstraction for various logging framework
- Not much log configuration
- Simple usage

# Log4j2

- Log4j2 + Slf4j is a de-facto standard for logging
- Three main components can be defined
  - Loggers
  - Appenders
  - Layouts

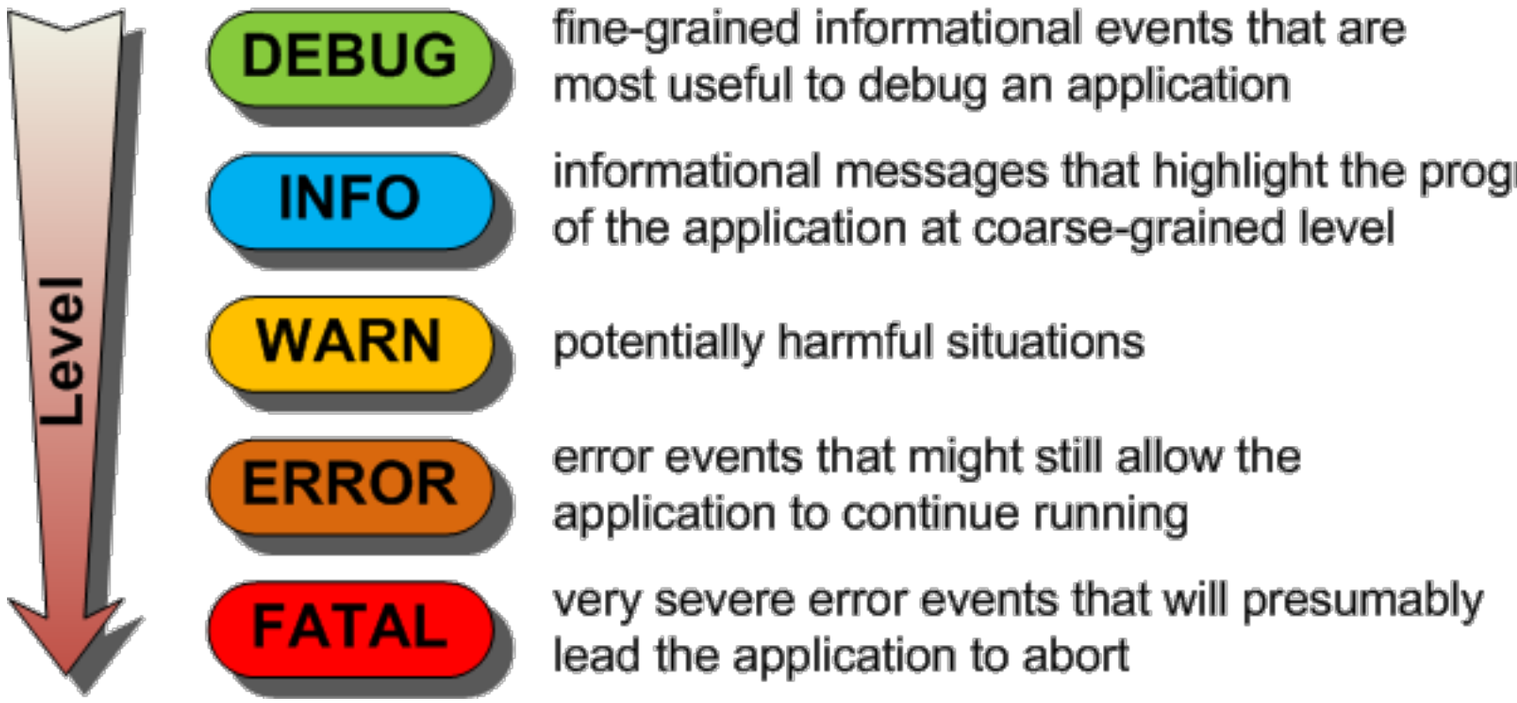
# Logging levels

- The categorized labels of a log entry based on the **level of urgency**.
  - Urgent messages generated during the production may often mean that a too unexpected behavior has occurred.
- Clearly defined labels will help anyone who reads the log entries to be able to **quickly separate** the kinds of information by the level of urgency.

# Example

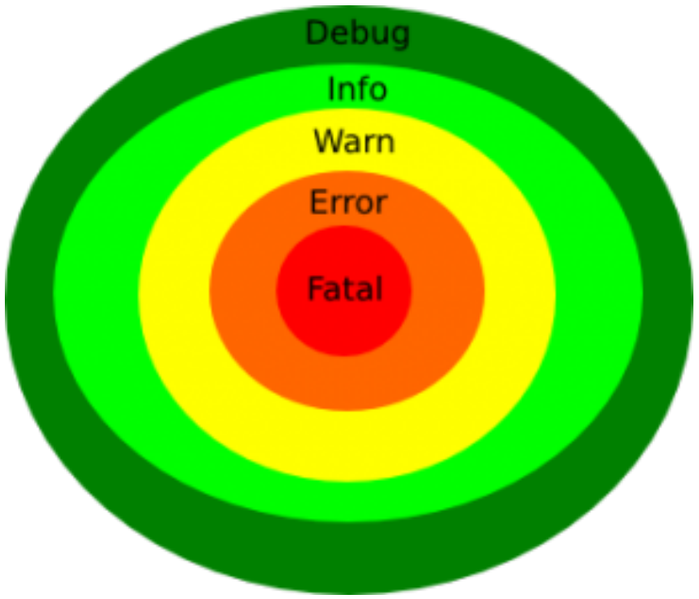
- A log entry stating that it is related to a severe error should be caught by every readers' eye.
  - E.g., intruder detected, system crash, etc.
- On the other hand, only some readers may be interested in the log entry suppling trivial information.
  - E.g., a known user logged in, updated successfully, etc.

# Logging levels




**ref:** https://ayende.com/blog/173761/the-role-of-logs (access May 2020)

# Logging — Levels




- A log request of level `p` in a logger with level `q` is enabled if `p >= q`

ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF

| | | | x: Visible | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | FATAL | ERROR | WARN | INFO | DEBUG | TRACE | ALL |
| OFF | | | | | | | |
| FATAL | x | | | | | | |
| ERROR | x | x | | | | | |
| WARN | x | x | x | | | | |
| INFO | x | x | x | x | | | |
| DEBUG | x | x | x | x | x | | |
| TRACE | x | x | x | x | x | x | |
| ALL | x | x | x | x | x | x | x |

**ref:** https://stackoverflow.com/questions/7745885/log4j-logging-hierarchy-order (access May 2020)

# Loggers

- Loggers are responsible for capturing log events, e.g., those that were not denied due to log levels configured and passing the log events to the appropriate **Appender**.
- They give the programmer runtime control on which statements are sent to the selected Appender or not.

```
final Logger logger = LoggerFactory.getLogger(Character.class);

...

logger.info("Moving left @ {} m/s",xVelocity);
```

```
[Thread-4] INFO platformer.model.Character - Moving left with 5 m/s
[Thread-4] INFO platformer.model.Character - Moving left with 5 m/s
[Thread-4] INFO platformer.model.Character - Moving left with 5 m/s
[Thread-4] INFO platformer.model.Character - Moving left with 5 m/s
[Thread-4] INFO platformer.model.Character - Moving left with 5 m/s
```

# Loggers

- logger.debug("Speed {}",xVelocity);
  - is somewhat equal to

```
if(logger.isDebugEnabled()) {
    logger.debug(“speed” + xVelocity);
}
```

# Loggers

```
public static void main(String[] args) {
  LOGGER.setLevel(Level.WARN);
  LOGGER.trace("Trace Message!");
  LOGGER.debug("Debug Message!");
  LOGGER.info("Info Message!");
  LOGGER.warn("Warn Message!");
  LOGGER.error("Error Message!");
  LOGGER.fatal("Fatal Message!");
}
```

**Output:**

```
Warn Message!
Error Message!
Fatal Message!
```

# Appenders

- Appenders are responsible for recording log events to one or more output destinations
- The destination can be
  - Console
  - File/Rolling file
  - Database
  - E-mail
  - etc.

# Appenders

- Multiple Appenders can be attached to any Logger, so it's possible to log the same information to multiple outputs.

# Appenders — Example

- ConsoleAppender – the most frequently used.
- FileAppender – writes messages to the file.
- DailyRollingFileAppender – creates a new file, add the year, month and day to the name.
- RollingFileAppender – creates a new file when the specified size, adds to the file name index, 1, 2, 3.
- SMTPAppender – sending e-mails.

# Layout

- Customize the message
  - An Appender uses a Layout to format a LogEvent into a form that meets the needs of whatever will be consuming the log event.
- Not only the message sent by the logger will be shown
  - Other information can be injected automatically
  - To investigate where the error is

# PatternLayout

- %d{ABSOLUTE}
  - Displays time; ABSOLUTE – in format HH:mm:ss,SSS
- %5p
  - Displays the log level (ERROR, DEBUG, INFO, etc.); use 5 characters, the rest padded with spaces;
- %t
  - Displays the name of the thread;
- %c{1}
  - class name with the package (indicates how many levels to display);

# PatternLayout

- %M
  - The method name
- %L
  - Line number
- %m
  - Message that is sent to the log; %c{1}
- %n
  - Newline

# Configuration

- Create a log4j.properties file and put it into the resources folder, e.g., under the project/classes directory
- Example

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\log4j-application.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

# Logging — recommended practice

- Declare the logger to be both **static** and **final** to ensure that every instance of a class shares the common logger object.
- Add code to check whether logging has been enabled at the right level.
- Use meaningful log messages that are relevant to the context.

# Logging — recommended practice

- Better to use logging to log the followings:
  - Method entry + method input params
  - Method exit
  - Root cause message of exceptions that are handled at the exception's origin point

# Debugging

- The process of detecting and correcting errors in a program on the fly.
- The problem that requires a debugger to figure it out is typically a more complicated case.
- More likely to be related to logic or algorithms such that if it happens early in the program, it may not manifest itself until much later when the implementation of many other components is competed and severely affected by it.

# Debugging

- program state anytime during its runtime by pausing the execution at any line of code to verify:
  - Memory
  - Variable value
  - Methods to be run
  - Stack trace

# Walking through a simple example

```
+ 1  public class Summation {
+ 2     public static void main(String[] args) {
+ 3        double sum = calculateSum(args);
+ 4        System.out.println("The summation is " + sum);
+ 5     }
+ 6     private static double calculateSum(String[] input) {
+ 7        double result = 0;
+ 8        for (String s : input) {
+ 9           result *= Integer.parseInt(s);
+ 10       }
+ 11       return result;
+ 12    }
+ 13 }
```

Neither compile error nor runtime error is presented

# Walking through a simple example

```
+ 1  public class Summation {
+ 2     public static void main(String[] args) {
+ 3        double sum = calculateSum(args);
+ 4        System.out.println("The summation is " + sum);
+ 5     }
+ 6     private static double calculateSum(String[] input) {
+ 7        double result = 0;
+ 8        for (String s : input) {
+ 9           result *= Integer.parseInt(s);
+ 10       }
+ 11       return result;
+ 12    }
+ 13 }
```

The particular output for the input [1,2,3,4,5] is 0!

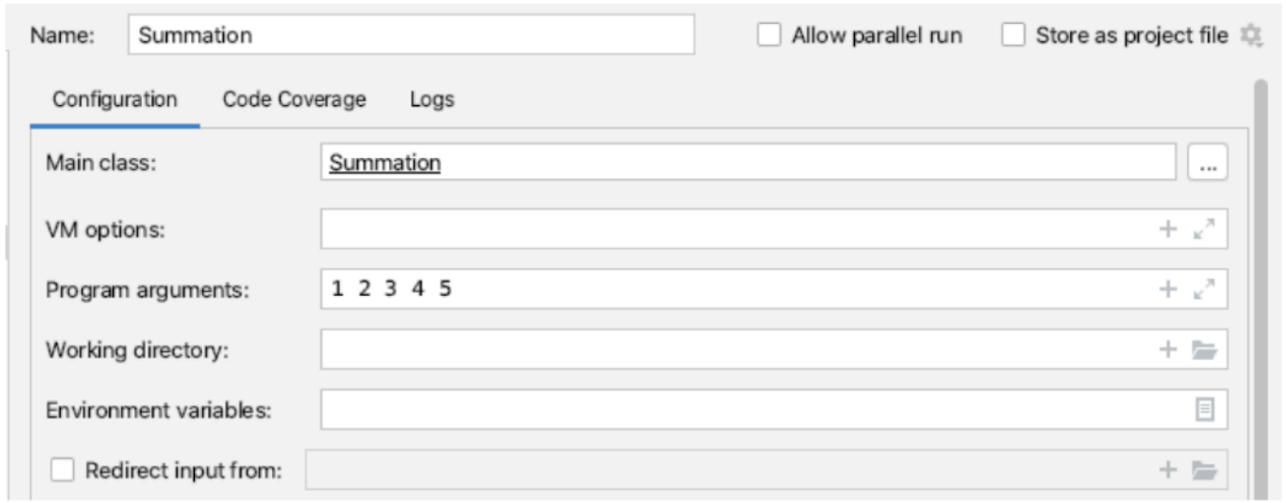
# Breakpoint

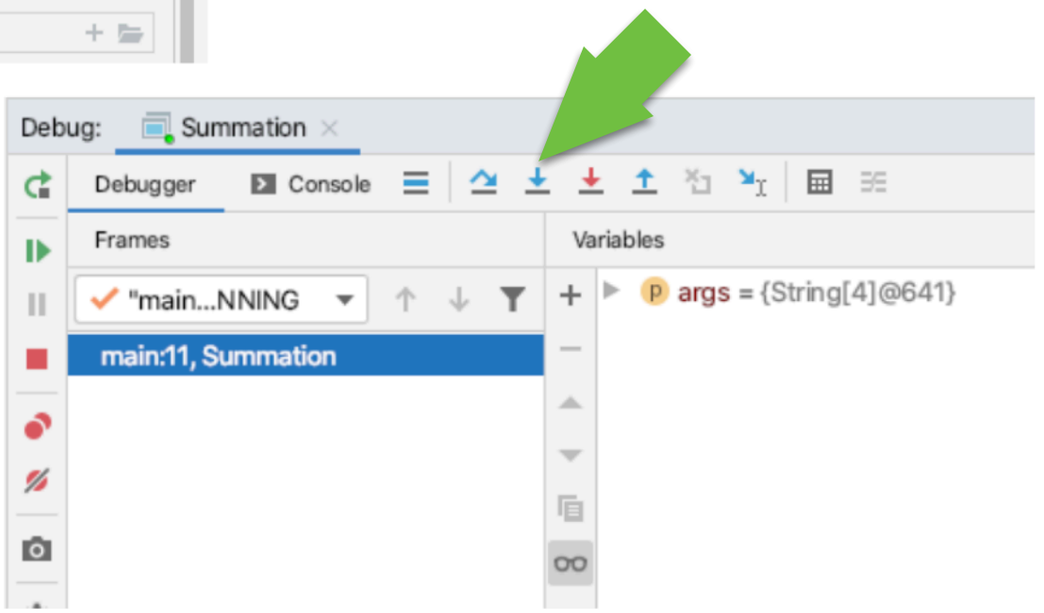
- The place to break the program
- Define the break point is to pause the program when it executes at that line of code

```
public class Summation {
    public static void main(String[] args) {
        double sum = calculateSum(args);
        System.out.println("The summation is " + sum);
    }
    private static double calculateSum(String[] input) {
        double result = 0;
        for (String s : input) {
            result *= Integer.parseInt(s);
        }
        return result;
    }
}
```

# Running the debugger

- Run -> Edit Configurations



# Step over/Step into

- Step over
  - Move to the next line of code
  - Compute all expression in the current line
- Step into
  - If there are some method calls
  - The break point move to the method

# Resume/End

- Resume
  - Continue the execution until the next operation
- End
  - Stop the application operation

# Bonus topic — Sprite animation

- A game is just a compositing of loops and some maths
- Generally there are 2 loops
  - Logic loops run e.g., 10 times a second and do the math
  - Drawing loops run e.g., 60 times a second and makes every drawing super-smooth

# For example

```
+ 1 for (int i = 0; i < 1000; i++) {
+ 2     x++;
+ 3     y++;
+ 4     c.redraw();
+ 5 }
```

- We will not see any animation in this way.

# Slow it down

```
   1 for (int i = 0; i < 1000; i++) {
   2    x++;
   3    y++;
   4    c.redraw();
+  5    try {
+  6       Thread.sleep(1000/60);
+  7    } catch(Exception e) {
+  8       e.printStackTrace();
+  9    }
  10 }
```

# Time-Based Motion

```
public void redraw() {
    // Draw image in first location
    // Erase first location
    // Draw image in final location
}
```

# Time-Based Motion

- Thing to consider
  - Teleportation != Animation
  - Too fast or too slow
  - We have to specify how fast the object gets from the initial position to the final position

# Animation

"Animation is the time-based alteration of graphical objects through different states, locations, sizes and orientations."

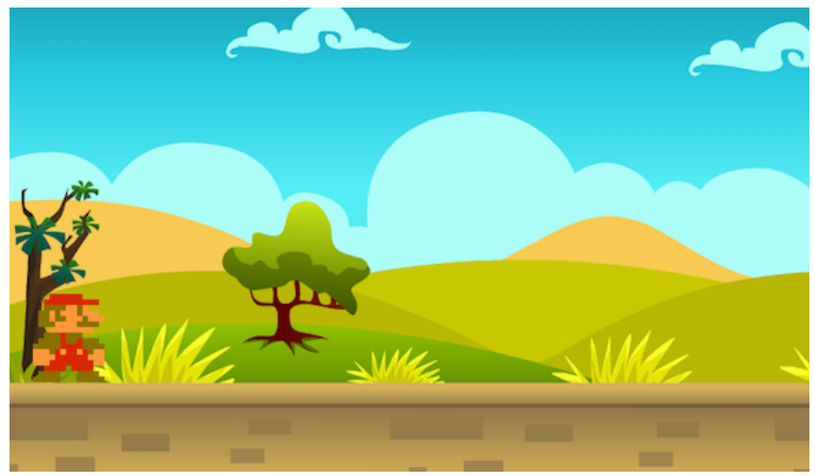
— Haase & Guy

# Key Components

- Drawing
- Clocks
- Interpolation
- Movement
- Collision

# Drawing

- E.g., Extend object to javafx.scene.layout.Pane and use it as a JavaFX node
  - Add the character pane Character as a child of the Stage pane.
  - What about other characters/npc?
  - User coordinate systems — (0,0) is the top left



# Clock

We have one clock that runs every millisecond

- Every 1/60 seconds, we do all the logic works related to drawing
- Every 1/10 seconds, we do all the other logic works

# Multiple Loops — 1) Logic Loop

```
+ 1  //Imports are omitted
+ 2  public class LogicLoop {
+ 3     public LogicLoop(Platform platform) {
+ 4        this.platform = platform;
+ 5        frameRate = 10;
+ 6        interval = 1000.0f / frameRate;
+ 7        running = true;
+ 8     }
+ 9     @Override
+10     public void run() {
+11        while (running) {
+12           update(platform.getCharacter());
+13           checkCollisions(platform.getCharacter());
+14           try {
+15              Thread.sleep((long) interval);
+16           } catch (InterruptedException e) {
+17              e.printStackTrace();
+18           }
+19        }
+20     }
+21  }
```

One clock tick

Work as thread

Update the model(s)

Check for collision(s)

Sleep the thread

Exception handling

# Multiple Loops — 2) Drawing Loop

```
+ 1  //Imports are omitted
+ 2  public class DrawingLoop {
+ 3     public DrawingLoop(Platform platform) {
+ 4        this.platform = platform;
+ 5        frameRate = 60;
+ 6        interval = 1000.0f / frameRate;
+ 7        running = true;
+ 8     }
+ 9     @Override
+10     public void run() {
+11        while (running) {
+12           checkDrawCollisions(platform.getCharacter());
+13           paint(platform.getCharacter());
+14           try {
+15              Thread.sleep((long) interval);
+16           } catch (InterruptedException e) {
+17              e.printStackTrace();
+18           }
+19        }
+20     }
+21  }
```

One clock tick

Examine collision that should be checked at the time of drawing

Draw/Redraw

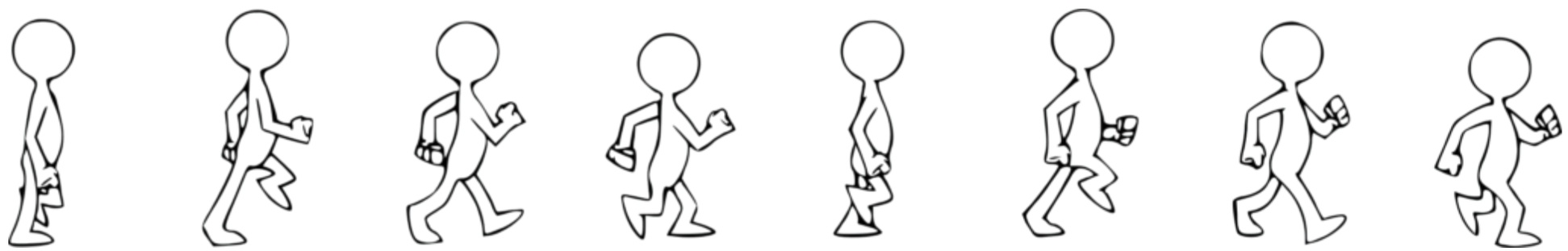
Sleep the thread

Exception handling

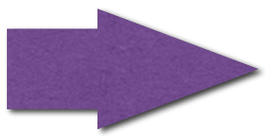
# Interpolation

The key is to have the drawing run more often than what is actually updated.

- Sprite sheet animation is one of the common technique



# Interpolation — E.g.,



# Interpolation — E.g.,

3 Columns and 2 Rows

Viewport is moved once per tick

| curIndex | curColumnIndex | curRowIndex |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 2 | 2 | 0 |
| 3 | 3 | 0 |
| 4 | 4 | 0 |
| 5 | 0 | 1 |
| 6 | 1 | 1 |
| 7 | 2 | 1 |
| 8 | 3 | 1 |
| 9 | 4 | 1 |

```
+  1  public void tick() {
+  2     curColumnIndex = curIndex % columns;
+  3     curRowIndex = curIndex / columns;
+  4     curIndex = (curIndex+1) / (columns * rows);
+  5     interpolate();
+  6  }
+  7  protected void interpolate() {
+  8     final int x = curColumnIndex*width+offsetX;
+  9     final int y = curRowIndex*height+offsetY;
+ 10     this.setViewport(new Rectangle2D(x, y, width, height));
+ 11  }
```

# Movement

- Every movable object must have a position, a velocity, and an acceleration.
- Acceleration changes your velocity, then your velocity changes your position
- In short, let acceleration control the movement.
- Outside forces can change your velocity directly

# Movement — implementation

- The event listener should be listened at the highest tree node, e.g., Stage or Scene in JavaFX
- Set the keyState handler in the 1/10s loop.
  - Key Event Handling
  - Key interact as a force that changes the acceleration
- Render the movement, e.g., change in position due to acceleration, in the 1/60s loop.
  - Use different method to respond to the key events

# Collision

- To detect if object is out of bounds or overlap with other objects.
  - Math works needs to be carried out
  - E.g., move back, HP reduction, dead scene render, score calculation, etc.

# Summary

- Faults and errors should happen in some places, we have to find it in order to fix it.
- Probing is an approach to modify the code to show the value of some specific variables on the screen.
- Logging is a chronological and systematic record of data processing events in a program.
- Debugging is an approach to trace the program on the fly.
- Sprite animation is a great tool to help us learn a lot of state and thread programming.

# Questions