Exercise 1

Selected Nouns:

- level
- Player
- ball
- Timer
- Score
- titlescreen
- Button
- Drops - Powerups (Grouped together, powerups is a subclass)
- Hook
- Mouse Click - Key Press
- Game Mode (1p/2p) - Attribute not Class

Level	
Superclasses:	
Subclasses:	
Create the Level	
Keep track of objects in level	Ball
	Player

Player	
Superclasses:	
Subclasses:	
Move	
Shoot Hooks	Hook
	Ball

Ball	
Superclasses:	
Subclasses:	
Hit Hooks + Spawn More Balls + possible drop drops	Hook, Drops
Bouncy Move	
Hit Players	Player

TimeSystem	
Superclasses:	
Subclasses:	
Update Visual Time	

Button	
Superclasses:	
Subclasses: SinglePlayerButton, MultiPlayerButton, QuitButton	
Get Clicked	
Change Level	Level

Score

Superclasses:	
Subclasses:	
Update visual score on ball-hook collision	Ball, Hook

MainMenu

Superclasses:	
Subclasses:	
Show Buttons	Button
Show Title	

Drops

Superclasses:	
Subclasses: Powerups	
Move	
Add score	Score
Drop From Balls	Ball

Hook

Superclasses:	
Subclasses:	
Move	
Be Shot by Player	Player
Collide With Ball	Ball

InputHandler	
Superclasses:	
Subclasses:	
Key Press	Level
Mouse Click	Player
	Button

Comparison:

The main difference is the lack superclasses in the design, in our implementation we use a GameObject class for all object that need to be displayed (Player, Button, etc.). We also have a Screen class for Screens that need to be displayed (TitleScreen, Level.) and a Board class to hold the screens and create an actual screen.

We use the Class NumberToken to display the digits of the Score- and TimeSystem.
Score is renamed to ScoreSystem in our implementaion.

Our implementation does not contain Drops or its Subclass Powerups, we didn't think this had priority in the first two weeks.

We included a ControlsScreen to show the controls before the game starts, (Can be linked to the requirement: "Controls should be clear..."). ControlsObj is an object to draw an image on this screen.

Hook is renamed HookAndRope because there is also a rather important Rope involved.

We have an Application Class that starts the application.

1.2 Main Classes:

<Abstract> **GameObject**	
Superclasses:	
Subclasses: HookAndRope, Player, Ball, NumberToken, Button, ControlsObj, TitlePang	
Enforces Objects to have a place	
Provides an Image and draw method	
Provides an Image Update methode	
Provides bounds for collision	

<Abstract> **Screen**	
Superclasses:	
Subclasses: MainMenu, Level, ControlsScreen	
Provides Update and Drawing Methods	
Keeps a list of Objects	

Level	
Superclasses: Screen	
Subclasses:	
Fills an ObjectList by reading an XML-file	
Sets the Background	

Board	
Superclasses: JPanel	
Subclasses:	
Allows Objects to be added to the ObjectList	Screen
Handles Update / Draw requests	
Starts Key/Mouselistener	InputHandler
Creates a Visible Window	
Allows Screens to be changed	

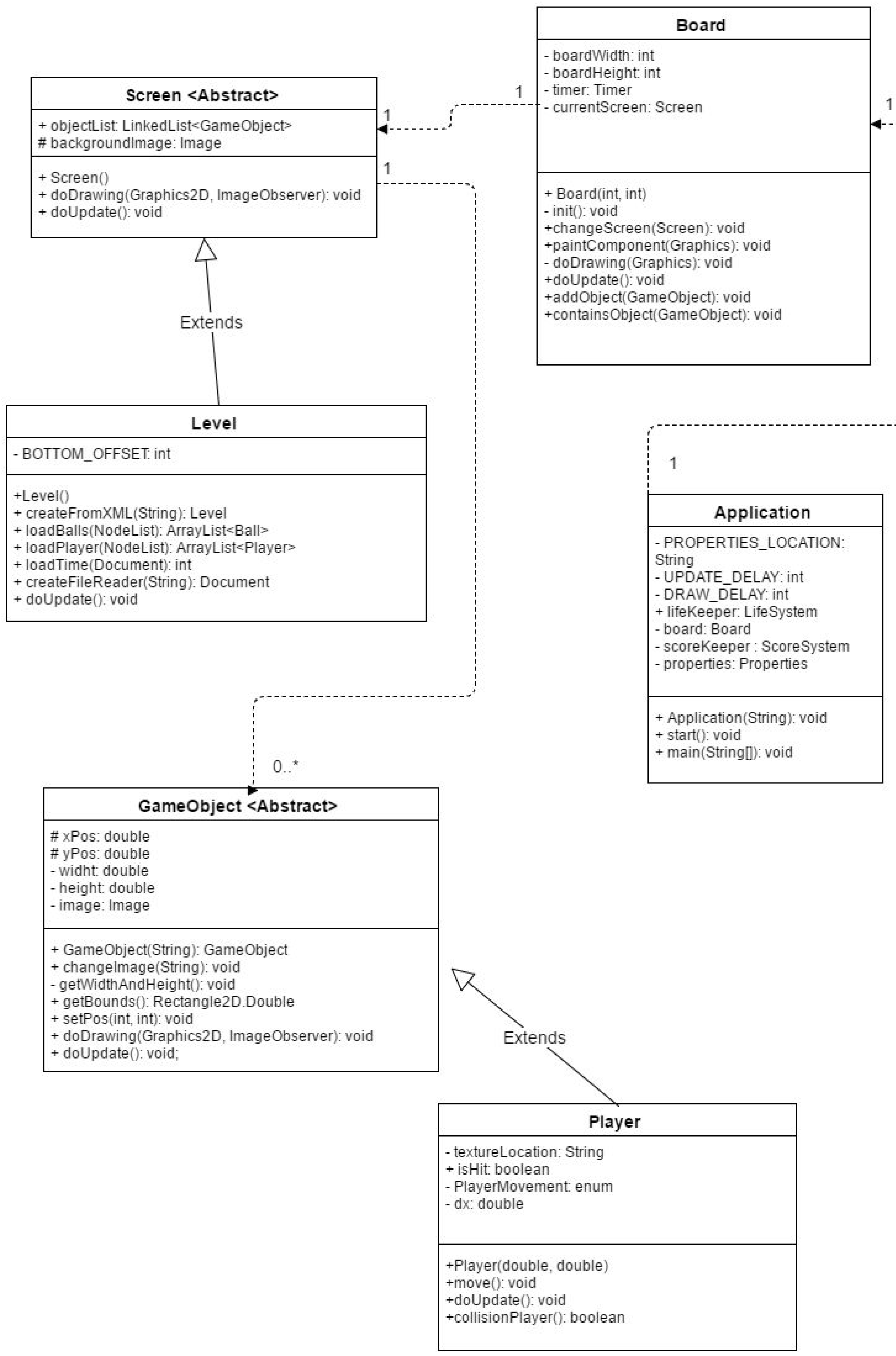
Application	
Superclasses: JFrame	
Subclasses:	
Starts Application	LifeSystem
Runs the update Loop	ScoreSystem
	Board

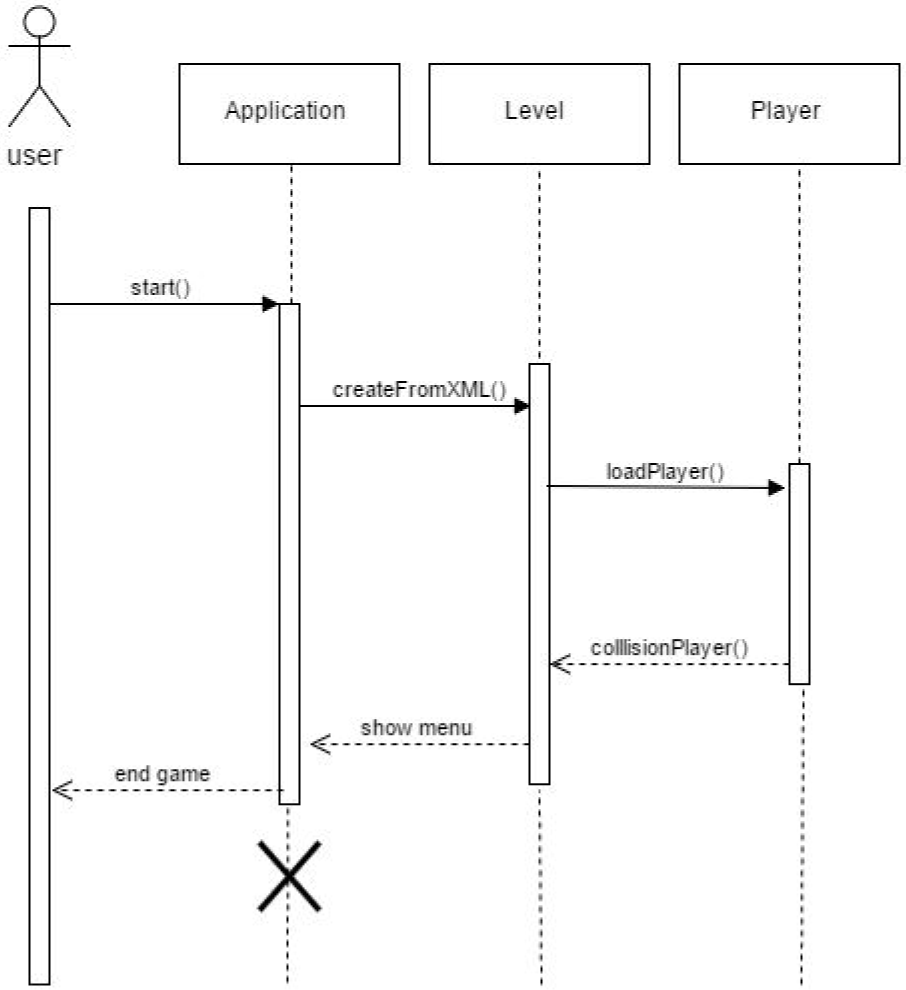
Player	
Superclasses: GameObject	
Subclasses:	
Shoots Hooks	HookAndRope
Moves on user input	

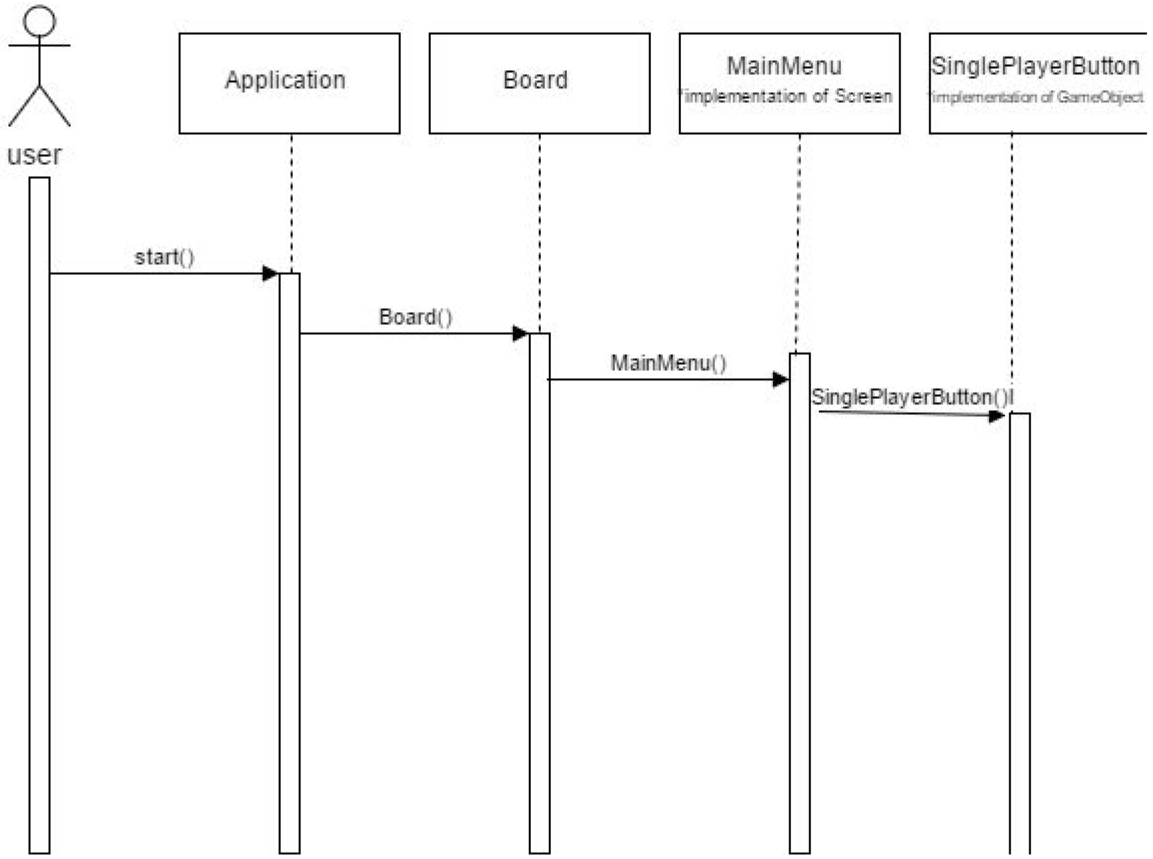
1.3

The classes that we chose as main classes (except Player) are the classes needed to start the application and draw our objects, without these the non-main classes, (and player), would be useless as they can’t be drawn or used. However Player was chosen as main class because it’s the most important interaction for the user during the game.

SinglePlayerButton could be merged into button by giving button a constructor with image and desired onclick behaviour parameters, however we think this is not an improvement because many buttons could have very different behaviour which requires more complex onclick behaviour then we want to handle in a general constructor.TitlePang and ControlsObj are purely images that could be merged into the background image, this would improve loading time and removes 2 redundant classes.


Board
- boardWidth: int
- boardHeight: int
- timer: Timer
- currentScreen: Screen
+ Board(int, int)
- init(): void
+changeScreen(Screen): void
+paintComponent(Graphics): void
- doDrawing(Graphics): void
+doUpdate(): void
+addObject(GameObject): void
+containsObject(GameObject): void
Screen <Abstract>
+ objectList: LinkedList<GameObject>
# backgroundImage: Image
+ Screen()
+ doDrawing(Graphics2D, ImageObserver): void
+ doUpdate(): void
1
1
1
1
1
1
Extends
Level
- BOTTOM_OFFSET: int
+Level()
+ createFromXML(String): Level
+ loadBalls(NodeList): ArrayList<Ball>
+ loadPlayer(NodeList): ArrayList<Player>
+ loadTime(Document): int
+ createFileReader(String): Document
+ doUpdate(): void
Application
- PROPERTIES_LOCATION: String
- UPDATE_DELAY: int
- DRAW_DELAY: int
+ lifeKeeper: LifeSystem
- board: Board
- scoreKeeper : ScoreSystem
- properties: Properties
+ Application(String): void
+ start(): void
+ main(String[]): void
0..*
GameObject <Abstract>
# xPos: double
# yPos: double
- widht: double
- height: double
- image: Image
+ GameObject(String): GameObject
+ changeImage(String): void
- getWidthAndHeight(): void
+ getBounds(): Rectangle2D.Double
+ setPos(int, int): void
+ doDrawing(Graphics2D, ImageObserver): void
+ doUpdate(): void;
Extends
Player
- textureLocation: String
+ isHit: boolean
- PlayerMovement: enum
- dx: double
+Player(double, double)
+move(): void
+doUpdate(): void
+collisionPlayer(): boolean


user
Application
Level
Player
start()
createFromXML()
loadPlayer()
colllisionPlayer()
show menu
end game


user
Application
Board
MainMenu
"implementation of Screen
SinglePlayerButton
"implementation of GameObject
start()
Board()
MainMenu()
SinglePlayerButton()I

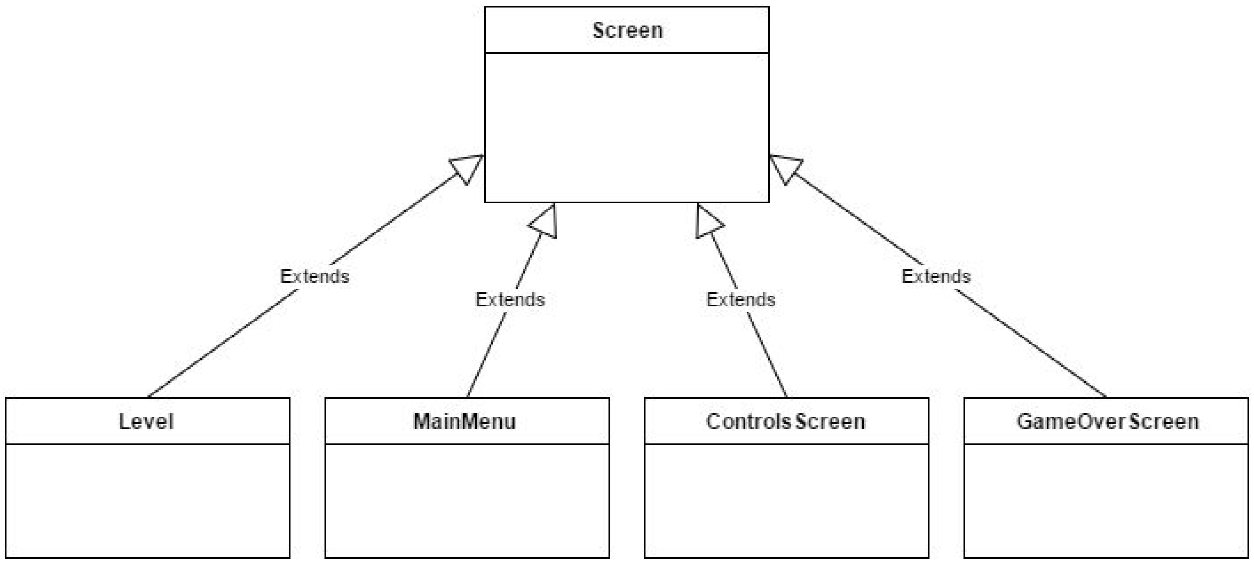
Exercise 2

2.1
A composition implies that the child cannot exist independent of its parent. With an aggregation, this is not necessarily true: the child can exist without its parent.

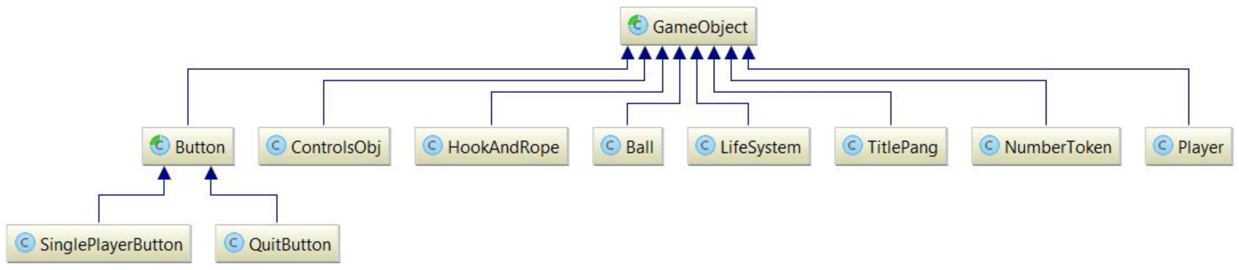
We have a Screen class and a GameObject class. GameObject instances cannot exist without a Screen object. When the Screen is removed, its GameObjects are removed with it. This is the only composition in our current project.

2.2
We do not use parameterized classes in our project. You would use a parameterized class when you want to have a method or class which you can use with multiple different types. If that class were non-parameterized, you would have to use objects instead of those specific types and cast them to the desired types when using them.

2.3



The Screen hierarchy.



The GameObject hierarchy.
All of our hierarchies are of the 'is-a' type.
We don't think any of these hierarchies should be removed, as they are very useful and seem to make sense.

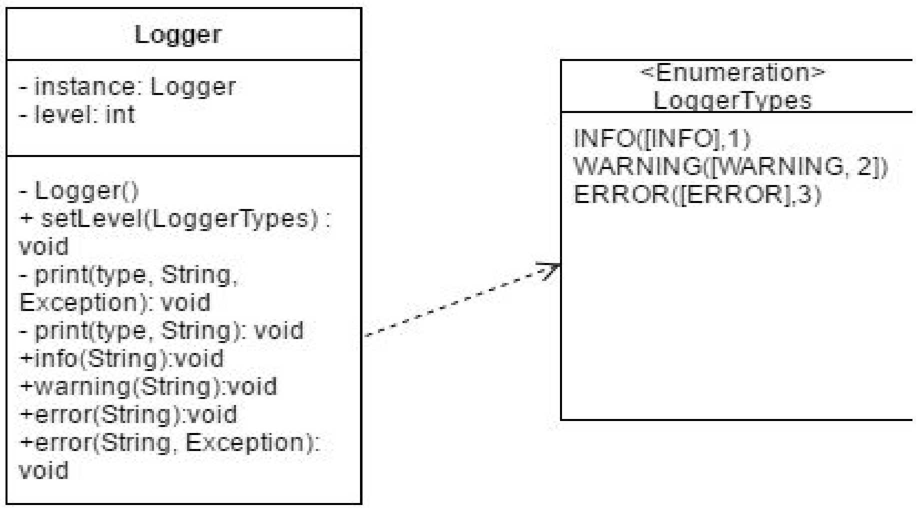
Exercise 3

3.1
Requirements:

- Being able to use different kinds of message: Info, Warning and Error, to show different priority. Error should also be able to Print the Stacktrace.
- We also want to use this logger for movement, registering objects, and the general start-up of the application and processes, we also use this logger for exceptions. (Except the newer ones because we are still working parallel).
- We also log on:
    - Changing screen
    - Starting a life-/time-/scoresystem
    - Starting the randomizer
    - On Board init

(We also added the ability to set a minimal level needed for it to actual print, this because the logger becomes useless with the amount of [INFO] messages for movement.)

3.2
The Logger is responsible for printing out provided messages with the right prefix, and when it is an error message with an exception to also provide the exception.
This object does not share any information, but the enum for the Logger Types is accessible for others if needed.


Logger
- instance: Logger
- level: int
- Logger()
+ setLevel(LoggerTypes) : void
- print(type, String, Exception): void
- print(type, String): void
+info(String):void
+warning(String):void
+error(String):void
+error(String, Exception): void
<Enumeration>
LoggerTypes
INFO([INFO],1)
WARNING([WARNING, 2])
ERROR([ERROR],3)