Situations
- Incorrect implementation
- Inappropriate Obj. Request
- Inconsistency
  ⇒ Decide CaseByCase

- Meant to be caught
- Compiler ensures they are controlled
- Try/Catch + Throw

· Handling = optional
· unchecked by compiler
· Prog. termination not caught

Error ← Throwable → Exception (checked)
↳ Runtime (unchecked)
- Illegal Argument
- index out of Bounds

Create New Exception
```
public Class SomeEx extends Exception
public someEx (string x) {
sysout(x); }
```

Concurrency, **Resolve DL**
- lock/unlock in same order
- lock conditions (Reentrant)

Find Deadlock
- nested synchronized
- run access to shared methods
- calls synchronized from synchronized

Methods
- join(); waits until previous Thread is finished
- run(); runs in main
- start(); starts new Thread
- set/getDaemon();
- sleep (milliseconds)
- wait(); pause thread
- notify/notifyAll(); wake sleeping Threads
- yield(); pauses current thread to let other Threads get lock
- signal/signalAll(); used for wake reentrant lock

wait() + ~~sleep~~ notify
ALWAYS TRY/CATCH + in Synchronized!

```
Class MyT extends Thread
public void run() { }
Thread MyT = new MyT(..);
MyT.start();

Class MyT implements Runnable
public void run() { }
MyT myT = new MyT(..);
Thread T = new Thread(myT, "name");
T.start();
```

Stop a Thread
```
Class MyT extends Thread {
volatile Boolean running = true;
public void run() { ...
  while (running) {...}}
public void terminate() {
  running = false;
  this.interrupt(); }}
```

Synchronized
· Problem → access shared Resource
↳ coop./coordination
· Types
- Mutual Exclusion
- condition synchronization
- synchronize methods called in run → if those call other Methods which change variables, change those!

Types continued.
- Hold-and-wait
- cycle condition
↳ Chain of processes which are waiting for a resource the successor holds

Reentrant Locks
when? - interruptable lock-wait
- multiple-condition variable

Create Reentrant Locks
```
private final lock mutex = new ReentrantLock();
private final Condition condA = mutex.newCondition();
private final Condition condB = mutex.newCondition();

// shared Resource 1:
public void SR1() {
mutex.lock();
try { while ~~[...]~~ (condition) { condA.await(); }
// do shared stuff AND condB.signal(); }
catch (interrupted exception) {...}
finally { mutex.unlock(); } }

// shared Resource 2:
public void SR2() { mutex.lock();
try { while (other Condition) { condB.await(); }
{ // do shared stuff AND condA.signal(); }
catch (interrupted exception) {...}
finally { mutex.unlock(); } }
```

Producer / Consumer.
```
class Cook extends Thread {
private SQue quee;
 public Cook (sque q) {
  quee = q; }
public void run() {
while (true) { try { que.remove(); Thread.sleep(); }
catch (Interrupted Exception) {...} } }

Class Guest extends Thread {
private SQue sue
 public void run() {
  while (true) try { que.add(..);
  Thread.sleep(100); catch (int) {}
  {}

Class SQue { private String[]
que; int count = 0; // Konstruktor
public synchronized void add
(string x) { while (count > que.length())
 try { wait(); } catch (int Ex) {}}
 que[count++] = s; notifyAll(); }
public synchronized remove() {
 while (count == 0) { wait(); }
catch () {} String ret = que[0];
count --; notifyAll(); return ret; }
```

Set up connection TCP client
```
public Class TcpClient { main .. {
 Socket clSocket = new Socket();
String server = // name or IP;
int server Port = 999;
InetSocketAddress = endpoint;
 try { endpoint = new InetSocketAddress (InetAddress.getbyname(server), serverPort);
  clSocket.connect(endpoint); catch (IOException e) {..}
 finally { clSocket.close(); // Disconnect from server }}}
```

Server
- Per Client 1 socket!
- connection requests are sequential!

Thread Zugriff
inner mit "Thread.currentThread()"

Accept TCP connection Server
```
main { int serverport = 999; Server Socket srSocket = new ServerSocket(serverPort);
 while (true) { Socket clSocket = srSocket.accept(); SocketAddress clAddress = clSocket.getRemoteSocketAddress();
 Sysout("Handling Client at: " + clAddress); clSocket.close(); }}}
```

TCP Server
```
Class Server {
serversocket srv = new ServerSocket(port);
while (true) { Socket clientS = srv.accept();
Thread thread = new Thread (new ClientHandler (ClientSocket));
thread.start();
```

ClientHandler
```
Class ClientHandler implements Runnable() {
private socket clientsocket; // Konstruktor
public void run() { recieveMsg(); sendMsg();
try { Client socket.close(); } catch (IOEx) {...}}}
~~sendmsg~~ void sendMsg (String x) {
 try { outputstream out = clientSocket.getoutputstream();
 out.write (x.getBytes, int, int); } catch (IOE) {...}}
void recieveMsg() { int totalBytesRecieved = 0; int bytesRecieved;
*while (totalBytesRecieved < data.length) { if((
 bytesRecieved = in.read (data, totalByteRecieved, data.length - totalBytesRec) == -1)
 {throw new SocketException ("connetion closed prematurely"); }
 totalByteRecieved += bytesRecieved; }
 Sysout("Recieved: " + ~~data~~ new String(data)); }
```

Alternative
```
* try { Inputstream in = clientSocket.getInputstream();
byte buffer = new byte [30]; int msgSize = -1;
do { if(msgSize = in.read(buffer)) != -1 {
  String s = new String (buffer); } while (...)
 catch (IOE) {...}}}
```

Set up In/outputstreams
in/outputstream in/out = new in/outputstream;
in/out = clsocket/servsocket.
get in/outputstreams;

I/O
Byte stream.
↳ Inputstream
↳ outputstream
Character stream.
↳ (Buffered) Reader
(Buffered) Writer
Predefined
System.in (keyboard)
System.out (console)
System.err (console ERR)

Char-Orientated
```
Buffered Reader bw = new BufferedReader( new
 FileReader ("Path.txt"));
```

Read File
```
public void reader() { *
try ( Buffered Reader bw = new BuffR. (new FileReader(
"Path.txt") { String currentLine = "";
while ((currentLine = bw.readLine()) != null) {
ArrayList <String[]> words = new ArrayList <Str..>
 words.add(currentLine.split(" ") *3 } } *4
```
*3 catch (FileNotFoundEx) {}
*4 finally { bw.close() }

Write File
```
public void writer (*) {
try { BufferedFileWriter fw = new Bufferfw.(new
 FileWriter (" path-to-file-where-to-write.txt")
ArrayList <String []> words for (String[] word :
 words) { for(int i=0; i < word.?; i++) {
 fw.write(word [i] + " "); fw.write("\n");
 *2 } fw.close()
```
*2 finally { fw.close }
*1 catch (IOEx) {...}

Tooling Git commands:
- git init, git branch name,
- git checkout name, git commit --amend,
- git branch -D name, git add,
- git push --set-upstream origin name,
- git checkout master + git merge name // merge to master
- git fetch (to local Repo)

Maven Commands: (mvn)
- clean: delete target dir
- compile: compile source code + classes stored in target
- test: runs tests
- install: install into local repo
- deploy: tests + copies files to remote repo

```
archetype:create -DgroupID = org.company.project -DartifactID =
application // java Project - ~~DartifactID~~ -DarchetypeartifactID =
 maven-archetype-webapp ⇒ new webapp/java project
< project xmls = ... >
 <model version> ... </modelversion>
 <group ID / artifactID ... > </>
 <name> ProjectDone </name>
 <dependencies>
 <dependency>
  <groupID> junit </groupID>
  <artifactID> junit </artifactID>
  <version> Version# / Latest (incl. beta) release (stable) </ver>
  <scope> compile // default runtime // not required for compile
  but for execution - test // not for normal use, only test </scope>
```

why?
+ Manage package dependencies
+ Force standard dir structure
+ simplify Build

WhiteBox
- use knowledge of product for test building
+ Find Errors + Thorough testing + Helps Opt. code
- might not find missing feat.
- Requires High Lvl knowledge of code - Req. code Access

BlackBox
- Test interfaces to ensure software functions
+ Large Code Segments
+ No Code Access Required
+ Separate User/Dev perspective
- limited coverage
- inefficiant testing

Principles
1. Define expected output
2. Don't test own code
3. Test expected, unexpected false, true inputs
4. Do throw-away test unless Software is throw-away
5. Prop. of errors increases in code segments already containing code errors

Benefits Mockito
+ only config of Mocks
+ supports return values, exception, verification + order checking

Limitations Mockito
- cannot test static, final, anonymous class or primitive types

1. Create

@Mock Swag swagA; } Test Class variables
@Spy Swag swagB;
@Before public void setup(){
MockitoAnnotations.initMocks(this);}

2. Specify expected behavior

*behavior needed in every test!

when(swagA.add(anyString)).thenReturn(x);

@Test(expected = Exception.class)
public void test10E(){
// do something that throws IOE}

doThrow(IOE.class)when(swagA.doX());

Multiple Calls: behavior can also be defined in setup();
when(swagA.doX()).thenReturn(x).
thenThrow(IOE); 1. Returns x 2. Throw IOE

define Answer (custom)
doAnswer(new Answer<String>(){
@Override
public String Answer(InvocationOnMock invocation){
return "Always the same";}}).when(x).get(anyInt());}

Call "Real" method
when(swagA.something()).thenCallRealMethod()

3. use mocks + verifications (done in actual Tests)
// test get Playlist
@Test public void testgetPlaylist(){
jukebox.add(mockSong)
jukebox.add(mockSong)
jukebox.add(mockSong)
} using Mocks: jukeBox is Class to be tested but relies on "song" class to work -> Mock the song class!
assertEquals(3, jukebox.getplaylist().size());
verify(mockSong, times(3)).getTitle();
// getPlaylist gets Titels of Songs
// verify checks .getTitle is called 3 times
} verification

Mock Verification

Struktur: verify(mockObj, verificationType()).behaviorDefined which class to be tested relies on();
Example: Behavior -> when(mockSong.getTitle()).thenReturn("1", "2", "3")
Test: jukebox.getPlaylist(); verify(mockSong, times(x)).getTitle();

Verification Types:
- atLeast(int)
- atLeastOnce()
- atMost(int max)

inOrder.verify() // checks if method calls in Order
Example: @Test public void testPlayMockInOrder(){
InOrder inOrder = inOrder(mockSong);
mockSong.getTitle();
mockSong.getPlaytime();
mockSong.getTitle();
mockSong.start();
inOrder.verify(mockSong).getTitle();
inOrder.verify(mockSong).getPlayTime();
inOrder.verify(mockSong).getTitle();
inOrder.verify(mockSong).start();}

Stub: minimal functionality
-> void return => no impl
-> other => Return hardcoded
Spies: can use real methods // avoid by using callRealMethod
Fakes: increased complexity but uses shortcuts not allowed in production
Mocks: Pre-programmed expectation and invocation

More Behavior
when(mock.x()).
for spies //
- .doAnswer()
- .doCallRealMethod
- .doNothing
- .doReturn
- .doThrow(IOE.class)

Model: contains Data + implementation of business logic
usually not visible to the user

UI: View -> contains GUI widgets
-> knows MODEL to get data for Display
-> listens to changes in data (model)

Controler
- listens to events from view (user input)
- Knows model to control/invoke it
- Knows View to refresh (e.g. deactivate Button)

class Button { priv. JFrame frame; priv JButton button;
priv. int klicks; public static main (String[] args){
new Button().go();} public void go(){ frame=new
JFrame(); frame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE); button = new Button(
"Klick ME"); button.addActionListener(new
ButtonListener());
frame.getContentPane().add(Button);
frame.setSize(200,100); frame.setVisible(
true);} private class ButtonListner
implements ActionListener {
@Override public void actionPerformed
(ActionEvent e){ button.setText(
++klicks + " mal geklickt");}}}