Prog 2 Exam Notes

Daniel Einars IT15a_ZH

Throwing Exceptions:

Options for Errorhandling: Resolve, continue with minimal loss, terminate
Use Defensive Programming to anticipate errors

Caught Ex.: (Checked)
- occur at compile time

UnCaught (Unchecked)
- occur at runtime (NPE)

Throwable

Extends Exception → Checked → Exception
Extends RuntimeE. Unchecked → Runtime → Exception

Either declare OR check → Subclass of Exception → for Anticipated

```
public void test() throws SomeException {
  if (x) { throw new SomeException("Error") }}
```

```
@Test (expected = SomeException.class)
```

```
public Class SomeException extends Exception
  public SomeException(String message) { sysout(Message) }
```

Detect Deadlock
- nested synchronized Blocks?
- calls synchronized from synchronized?

Avoid Deadlocks
- Lock\release in correct order
- Lock conditions (reentrant)

	Extend	Implement
Inheritence option	no	extends any
Reusability	no	yes
Obj. O. Desi.	Bad	good
Loose Coupl.	no	yes
Function Overhead	yes	no

```
if (Thread.currentThread != this) //wenn nicht main
public class X extends Thread/implements Runnable

  public void run(){
    while (condition) {       //run until cond changes
    x try{ x }                //if required
      catch (InterruptedException e) {...} }
                              //while(true) → runs
x ~~can be wait() or~~        //indefinitely .: Server accept
sleep ~~or join etc.~~          connection
x execute desired functions
```

Threadprin a/b = new Threadprin()
- a.run(): calls method
  - does **not** create new Thread
  - runs in main
  - a.run() } executes a, finishes,
  - b.run() } then executes b
- If only "run" is used, synchronized not required
- a/b.start(): creates new Thread(s)
  - shared recourses require synchronized
- a/b.join(): if Threads are started join waits for previous thread to finish before starting
- a/b.yield(): if a is running and b.yield() is called a pauses and yields to b
- notify(): wakes up waiting Thread
- notifyAll(): wakes up All waiting → equivalent to signalAll();

Wait on Condition: (Ampel)

```
while (red) { try { wait(); }
    catch (in.e) {..} }
notify(); OR notifyAll();
more car;
```

ReentrantLocks:

```
private final Lock mutex = new ReentrantLock();
private final Condition condA = mutex.newCondition();
private final Condition condB = mutex.newCondition();
```

shared Recourse 1:

```
public void\boolean etc method1() {
  mutex.lock();
  try { while (condition) {
        condA.await(); }
    do shared stuff AND
      condB.signal(); }
  finally { mutex.unlock(); } }
```

Shared Recourse 2:
throws InterruptedException

```
public void\boolean etc method2() {
  mutex.lock();
  try { while (condition) { condB.await() }
  do shared stuff AND condA.signal() }
  finally { mutex.unlock(); } }
```

Wenn Runnable
- Muss Runnable Objekt an Thread übergeben

```
Objekt a = new Objekt;
Thread ab = new Thread (a, "Name");
```

```
public void reader (ArrayList<String[]>) {
try (BufferedReader in = new BuffR. (new FileReader
                         "Path.txt") {
    String currentLine
  while ((currentLine = in.ReadLine()) =! null) {
    words.add (currentLine.split(" "); }}
```

```
public void writer (ArrayList <String[]> words)
                              throws IOExc {
  BufferedFileWriter fw = new BufferedFileWriter(
              new FileWriter ("path.txt");
  for (String[] word: words) {
    for (int i=0 i< word.length; i++) {
      fw.write/append (word[i] + " ")
    } fw.write("\n"); } fw.close(); }
```

Read until EOF:

```
while (in.ready() {...}
```

Always flush() + close()!

White Box

- uses knowledge of Software as part of testing process

+ Finding Errors
+ Thorough testing
+ Helps optimize Code

- Might not find missing features
- Requires High-Level knowledge of Code
- Requires Code Access

Black Box

- Test interfaces to ensure software functions

+ Large Code Segments
+ No Codeaccess required
+ Seperate User/Developer perspective

- Limited Coverage
- Inefficient testing

Testing Principles

1. Define expected Output/Result
2. Don't test your own code
3. Test ALL inputs
4. No "throw away" tests
5. Probability of errors increases in section already containing errors

---

Dummy: used to fill parameterlist
Stubs: minimal functionality → void = no implementation
→ other = return hardcoded values
Spies: records which members were invokated + uses "real" methods
Fakes: increased complexity but take shortcuts not valid in production (in-memory DB)
Mock object: pre-programmed expectation and interaction

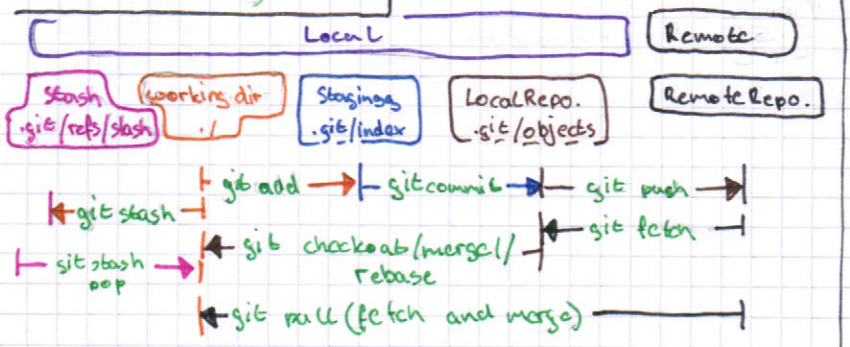
Benefits Mockito

+ Only configuration of Mocks required (no writing of Mock methods)
+ supports return values exceptions, verification over order checking

Limitations Mockito

- cannot test static, final, anonymous classes or primitive types

Toolings



Initialize git: git init
create new branch: git branch swagBranch
Switch to Branch: git checkout swagBranch
Merge Branch into master: 1. git checkout master
2. git merge swagBranch
Edit last commit msg: git commit --amend
push new local Branch to remote Repo: git push --set-upstream origin swagBranch
remove Local Branch: git branch -D swagBranch

Mock Usage Pattern

1. Create:

```
@Mock Swag swagA; @Spy Swag spySwag
@Before public void create() {
MockitoAnnotations.initMocks(this);}
```

2. Specify expected Behavior

a) Return: when(swagA.add(anyString())).then Return(false);

b) Throw exception on Method Call

```
@Test(expected=.)
public void testing() {
when(swagA.add()).thenThrow(Exception.class);
```

configured to throw exception

c) Throw Exception when x

doThrow(NPE.class).when(x);

d) Multiple Calls

when(swagA.doSomething()).thenReturn(x)
.thenThrow(OR E.class);

1st Call: returns x
2nd Call: throws NPE

d) Mock method call with custom Answer

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

e) Call real underlying method

when(swagA.add()).thenCallRealMethod();

3. Use Mocks then → 4. Verification

e) Max/Min Invocation
verify(swagA, atLeast(1)).someMethod();
OR .atMost(10)...

a) n interactions with Mock
verify(swagA, times(n)).someMethod();

b) no interaction: verifyZeroInteraction(swagA);

f) Invocation flexible (arguments)
verify(swagA).someMethod(anyString()/anyInt());

no Interaction with specific Method
verify(swagA, times(0)).someMethod();

g) order Invocation

```
inOrder.verify(swagA).someMethod();
inOrder.verify(swagA).someOtherMethod();
```