Spring Framework

Transaction Management

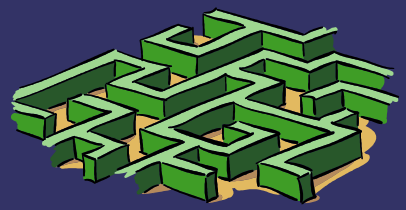
By Uma

Why Transaction?

- Data Integrity
- Consistency
- Data and Resources may be corrupted and left in an inconsistent state.
- Important for recovering from unexpected errors in a concurrent and distributed environment .

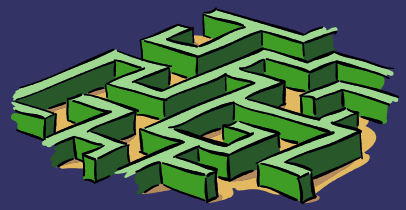
What is ACID?

- Atomicity:
  - A transaction is an atomic operation that consists of a series of actions.The atomicity of a transaction ensures that the actions either complete entirely or take no effect at all.
- Consistency:
  - Once all actions of a transaction have completed, the transaction is committed. Then your data and resources will be in a consistent state that confirms to business rules.



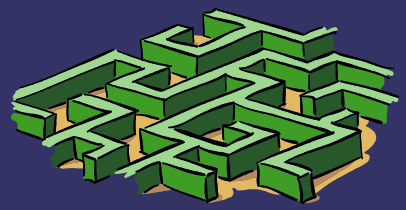
What is ACID?

- Isolation:
  - Because there may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
- Durability:
  - Once a transaction has completed, its result should be durable to survive any system failure (imagine if the power to your machine was cut right in the middle of a transaction's commit). Usually, the result of a transaction is written to persistent storage.



ACID Example

- Atomicity: an entire document gets printed or nothing at all
- Consistency: at end-of-transaction, the paper feed is positioned at top-of-page
- Isolation: no two documents get mixed up while printing
- Durability: the printer can guarantee that it was not "printing" with empty cartridges.



Types of Transaction Management

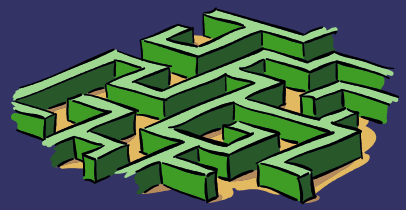
- Programmatic Transaction Management
- Declarative Transaction Management

Programmatic vs Declarative Transactions

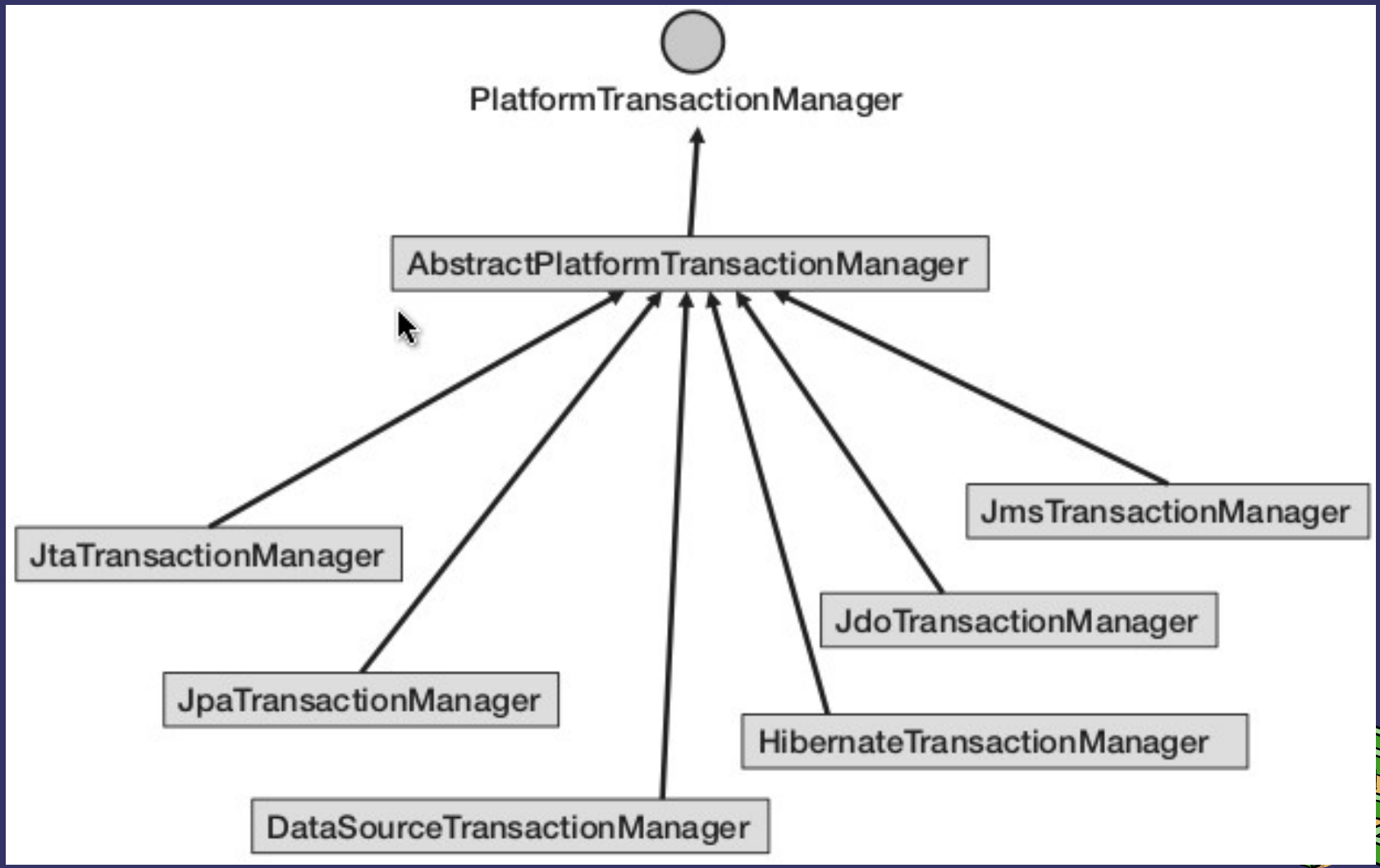
Programmatic Transaction Management	**Declarative Transaction Management**
Embedding transaction management code in your business methods	separating transaction management code from your business methods via declarations.
No support for AOP	Supports AOP
More flexible	Less flexible

Platform Transaction Manager

- Encapsulates a set of technology-independent methods for transaction management
- TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException
- void commit(TransactionStatus status) throws TransactionException;
- void rollback(TransactionStatus status) throws TransactionException;

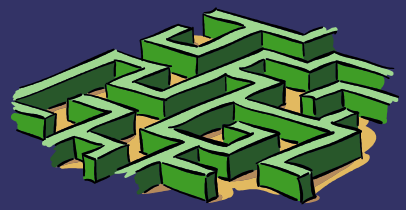


Choosing Transaction Manager Implementation



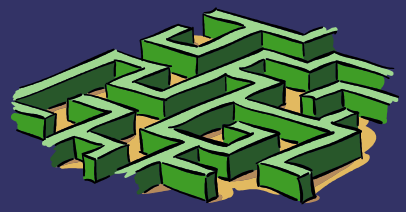
Programmatically with Transaction Manager API

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTrans
    actionManager">
<property name="dataSource" ref="dataSource"/>
</bean>
```



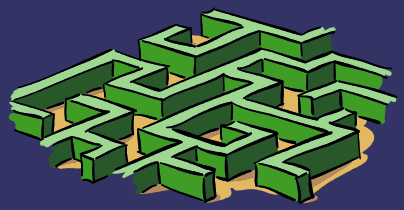
Programmatically with a Transaction Template

```
<bean id="transactionTemplate"
    class="org.springframework.transaction.support.Transaction
    Template">
 <property name="transactionManager"
    ref="transactionManager"></property>
</bean>
```

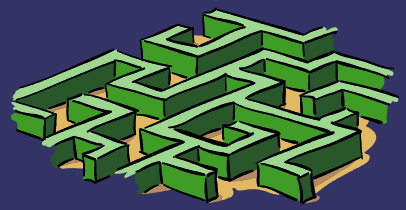


Declaratively with transaction Advices

```
<tx:advice id="bookShopTxAdvice"
transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="purchase"/>
</tx:attributes>
</tx:advice>
```



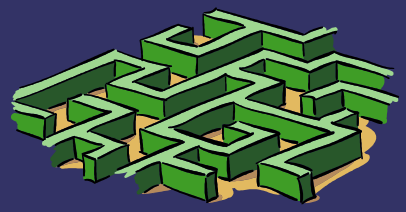
```
<aop:config>
<aop:pointcut id="bookShopOperation" expression=
"execution(* com.apress.springrecipes.bookshop.spring.
BookShop.*(..))"/>
<aop:advisor advice-ref="bookShopTxAdvice"
pointcut-ref="bookShopOperation"/>
</aop:config>
```



Declaratively with @Transactional Annotation

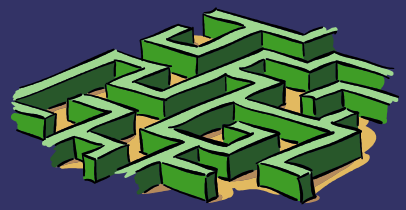
```
<beans ...>
<tx:annotation-driven />
</beans>

@Transactional
public void purchase(String isbn, String username)
{ }
```



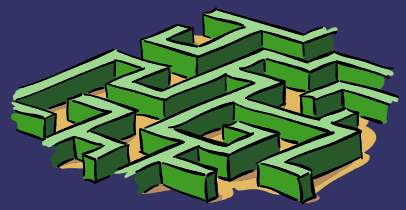
Propagation Transaction Attribute

- Transactional method is invoked by another method ,the transaction should be propagated.
- Method run with existing transaction or start a new transaction.
- Having 7 transaction propagation attribute which is defined in TransactionDefinition Interface.

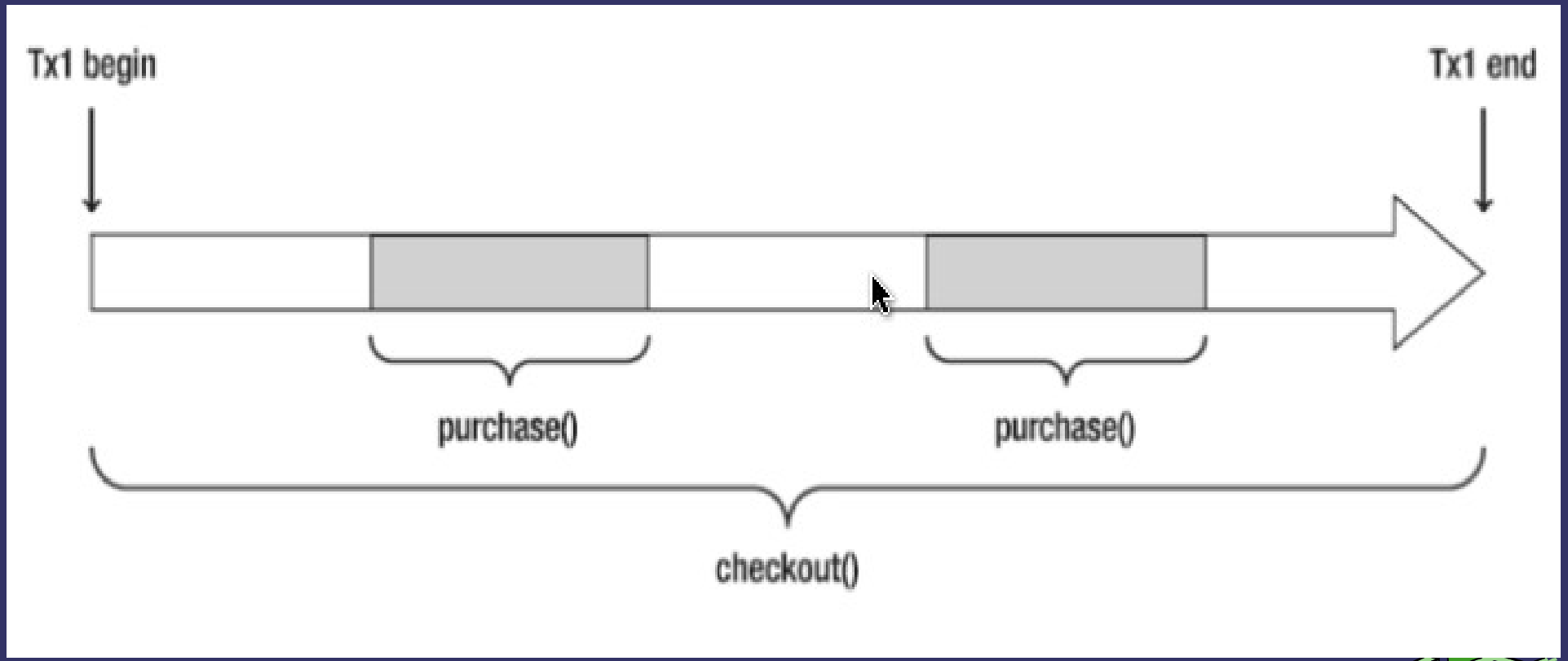


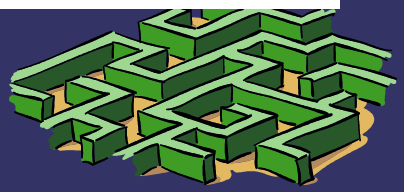
7 Propagation Behaviors

- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- MANDATORY
- NEVER
- NESTED

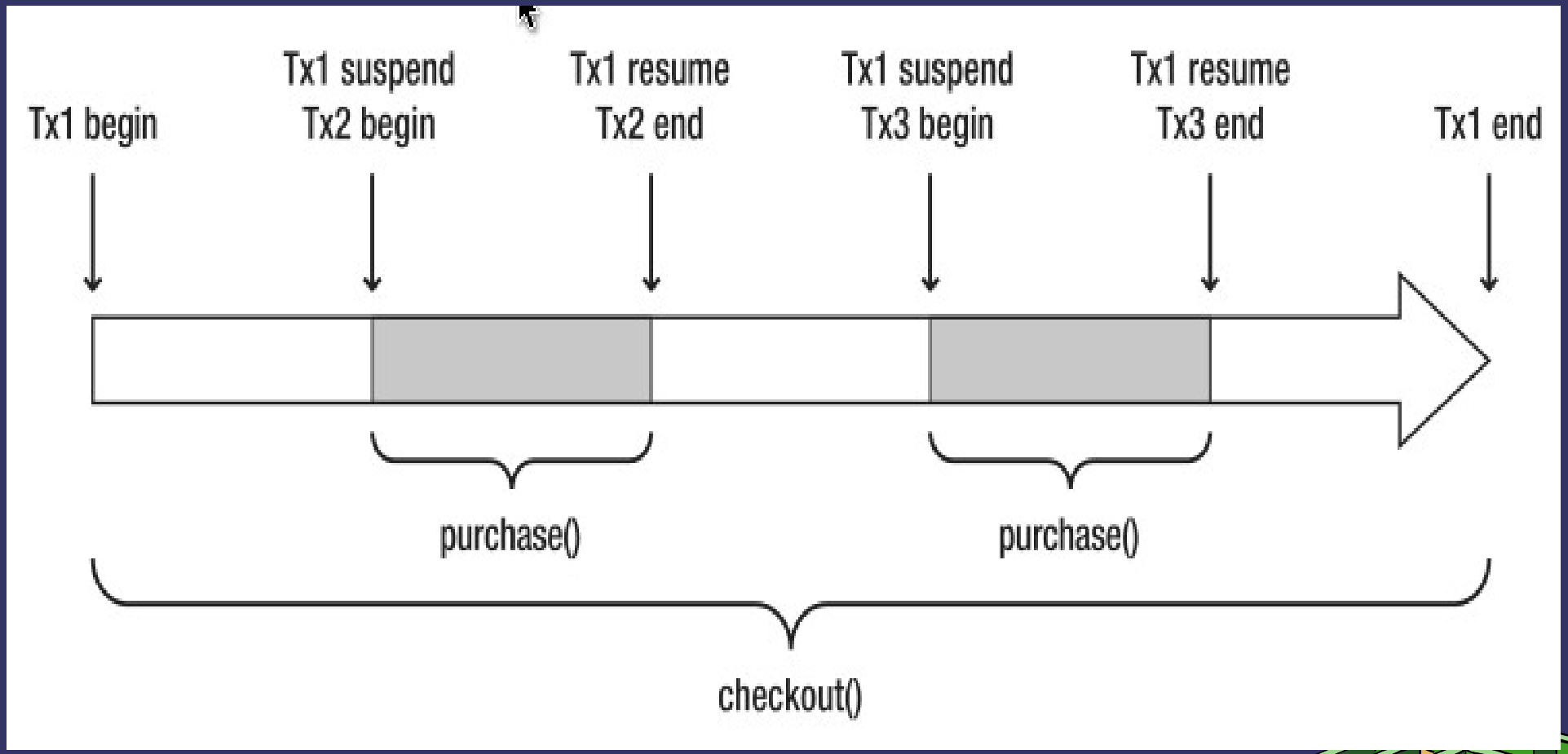


REQUIRED Transaction Propagation Behavior



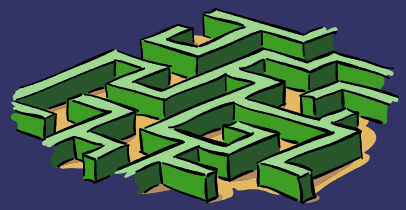


REQUIRES_NEW Propagation Behavior



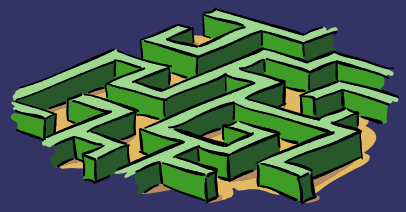
Setting the Propagation Attribute in Advice & API

- <tx:advice ...>
<tx:attributes>
<tx:method name=”...” propagation="REQUIRES_NEW"/>
</tx:attributes>
</tx:advice>
- DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);

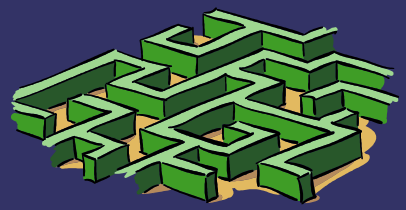


Isolation Transaction Attribute

- Problems due to concurrent transactions:
  - Dirty read: For two transactions T1 and T2, T1 reads a field that has been updated by T2 but not yet committed. Later, if T2 rolls back, the field read by T1 will be temporary and invalid.
  - Nonrepeatable read: For two transactions T1 and T2, T1 reads a field and then T2 updates the field. Later, if T1 reads the same field again, the value will be different.

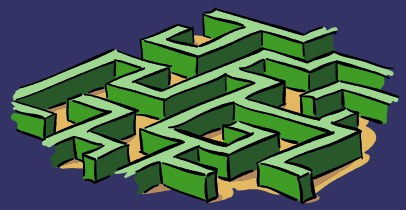


- Phantom read: For two transactions T1 and T2, T1 reads some rows from a table and then T2 inserts new rows into the table. Later, if T1 reads the same table again, there will be additional rows.
- Lost updates: For two transactions T1 and T2, they both select a row for update, and based on the state of that row, make an update to it. Thus, one overwrites the other when the second transaction to commit should have waited until the first one committed before performing its selection.



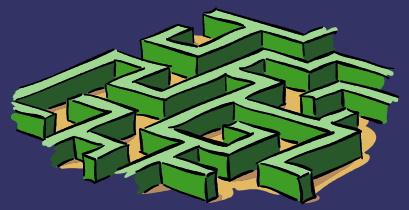
Isolation Levels

- DEFAULT
- READ_UNCOMMITED
- READ_COMMITED
- REPEATABLE_READ
- SERIALIZABLE



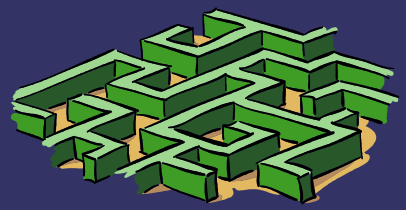
Setting the Isolation level in Advice and API

- <tx:advice ...>
<tx:attributes>
<tx:method name="*"
isolation="REPEATABLE_READ"/>
</tx:attributes>
</tx:advice>
- DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);



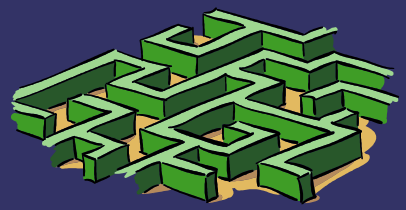
Rollback Transaction Attribute

- Transaction Rollback by Checked Exception.
- Attributes used for rollback
  - rollbackFor - IOException
  - noRollbackFor – ArithmeticException



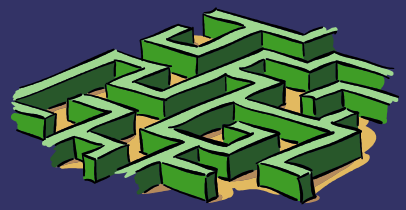
Set the Rollback attribute in Advice and API

- <tx:advice.>
<tx:attributes>
<tx:method name="..." rollback-for="java.io.IOException"
no-rollback-
for="java.lang.ArithmeticException"/></tx:attributes>
</tx:advice>
- RuleBasedTransactionAttribute attr = new
RuleBasedTransactionAttribute();
attr.getRollbackRules().add(new
RollbackRuleAttribute(IOException.class));
attr.getRollbackRules().add(new
NoRollbackRuleAttribute(SendFailedException.class));



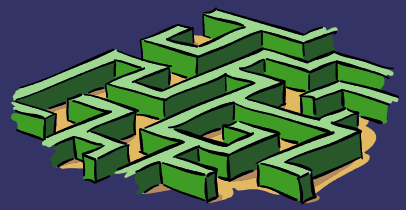
Timeout and Read-only Transaction Attributes

- timeout - how long your transaction can survive before it is forced to roll back.
- read-only -transaction will only read but not update data.



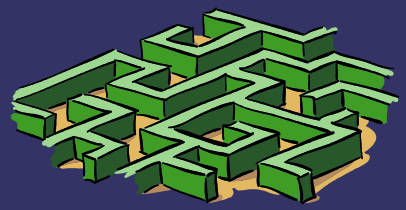
Set timeout and Read-only attribute in Advice and API

- <tx:advice>
<tx:attributes>
<tx:method name="checkStock" timeout="30"
read-only="true"/>
</tx:attributes>
</tx:advice>
- DefaultTransactionDefinition def = new
DefaultTransactionDefinition();
def.setTimeout(30);def.setReadOnly(true);



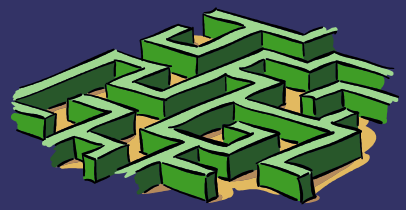
Managing Transactions with Load-time Weaving

- AnnotationTransactionAspect – manage transactions for any methods of any objects, even if the methods are non-public or the objects are
  created outside the Spring IoC container.
- Use AspectJ's compile-time weaving or load-time weaving to enable this aspect.
- Add @Configurable to the Domain class



Continued..,

- <context:load-time-weaver />
- <context:annotation-config />
- <context:spring-configured />
- <tx:annotation-driven mode="aspectj"/>
- Run the Application on Spring Agent spring-instrument.jar at load time.
- VM args -javaagent:spring-instrument.jar



THANK YOU