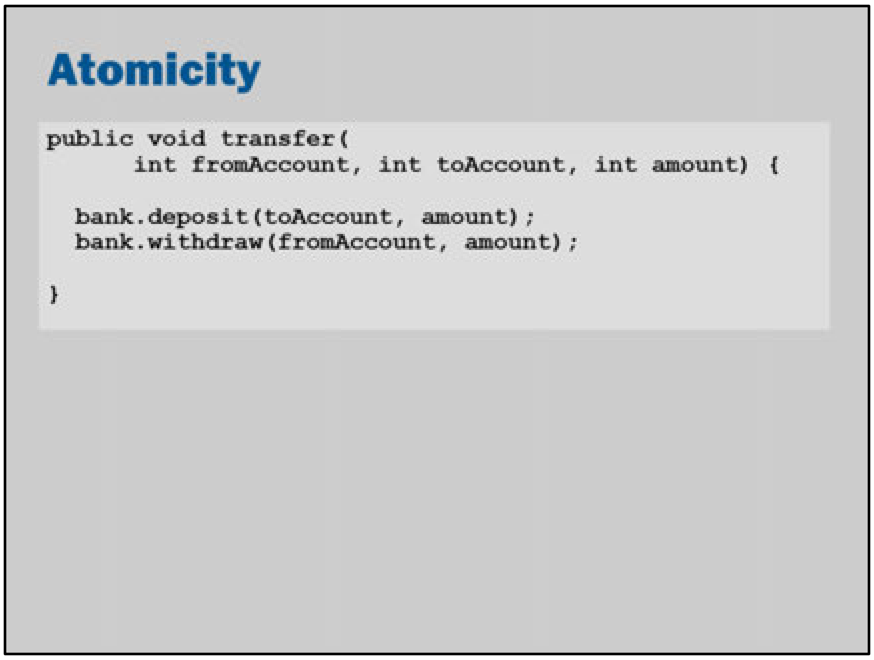
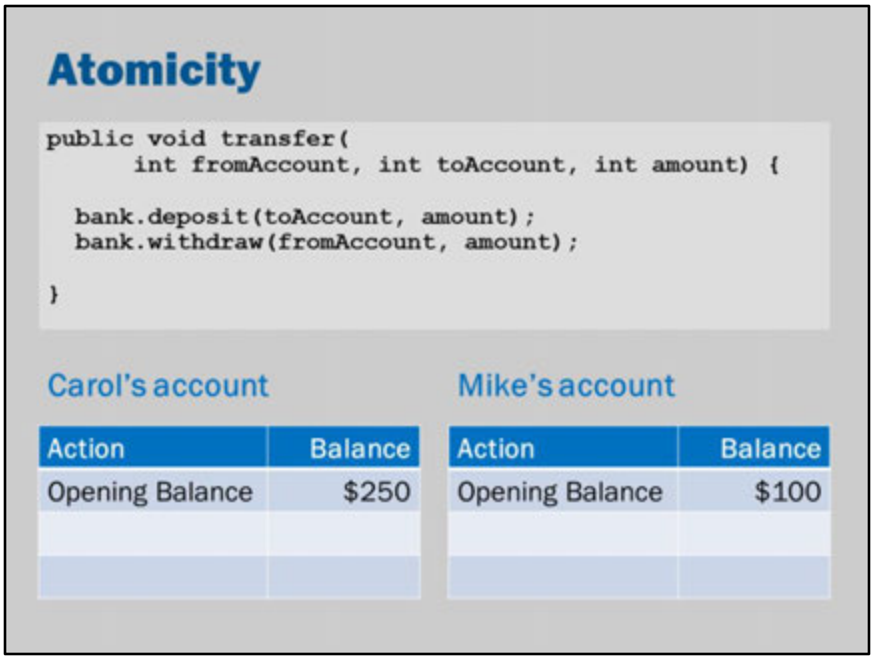
Transactions


Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

This scenario – transferring money between bank accounts – is the classic example to demonstrate the need for transactions.

What could go wrong with this code?



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

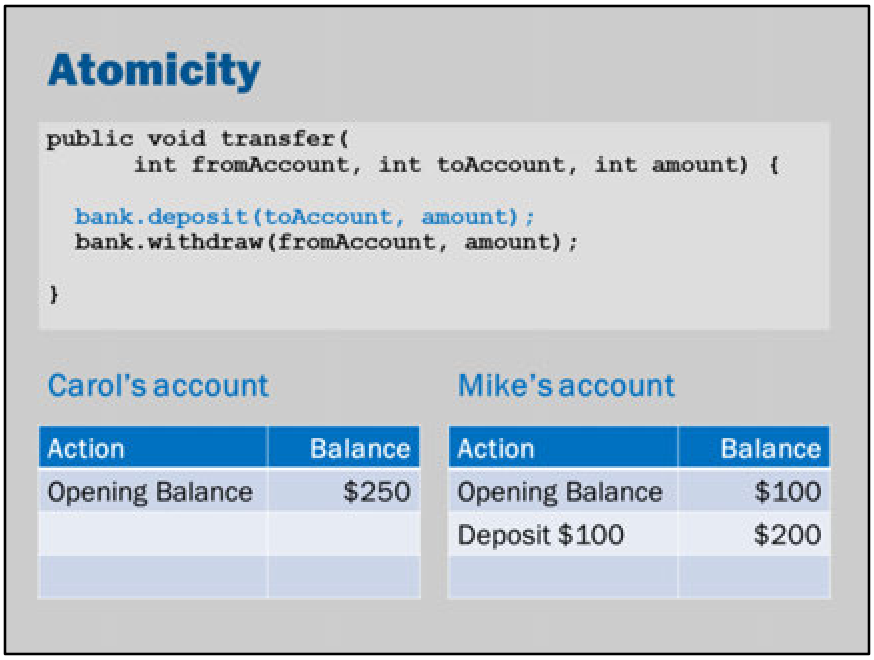
Carol's account

Action	Balance
Opening Balance	$250

Mike's account

Action	Balance
Opening Balance	$100

If transfer gets called, the first thing that will happen is money will be deposited in the target account.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

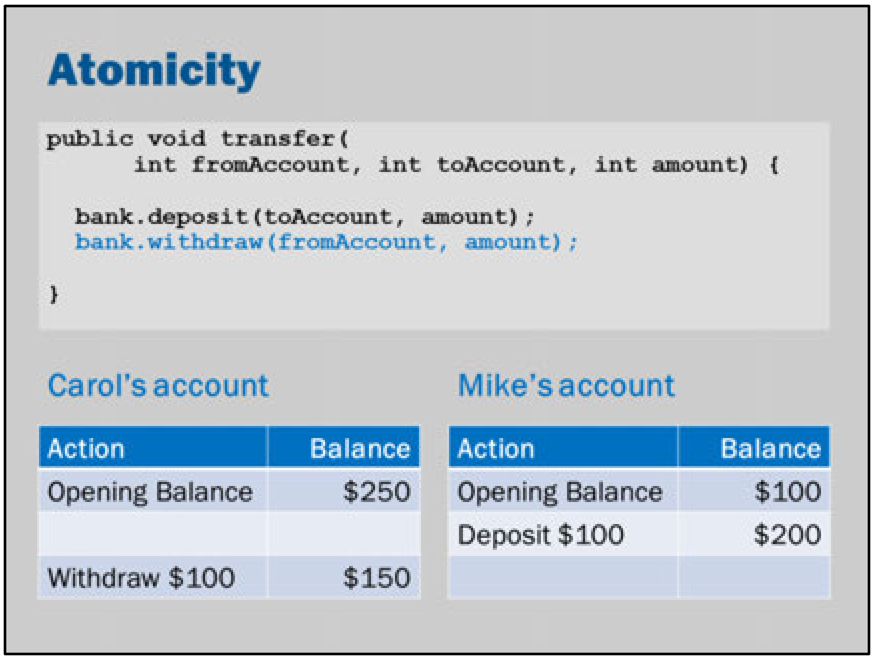
Carol's account

Action	Balance
Opening Balance	$250

Mike's account

Action	Balance
Opening Balance	$100
Deposit $100	$200

If transfer gets called, the first thing that will happen is money will be deposited in the target account.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

Carol's account

Action	Balance
Opening Balance	$250
Withdraw $100	$150

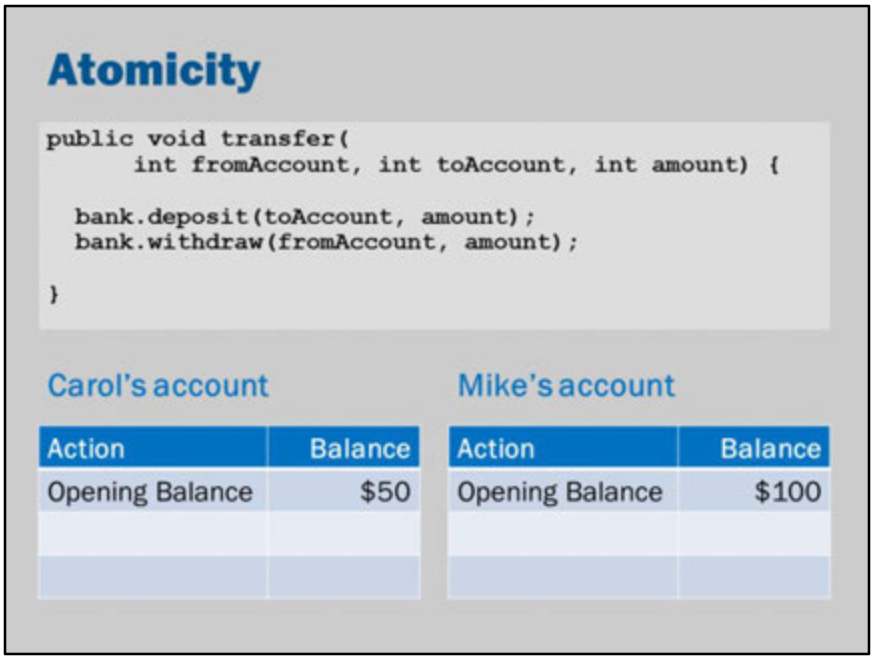
Mike's account

Action	Balance
Opening Balance	$100
Deposit $100	$200

If transfer gets called, the first thing that will happen is money will be deposited in the target account.
Then the $100 will be withdrawn from the source account.

There are many ways the application might fail:
- The database connection might fail
- The computer might crash
- The computer might lose power
- The fromAccount might not be valid
- The fromAccount might not have enough money
- The system might run out of memory

If any of these happen just before withdrawing the money, you will have a bank account with more money and no account with less money. The bank has just made a "loss" – somebody has received "free" money!



Atomicity

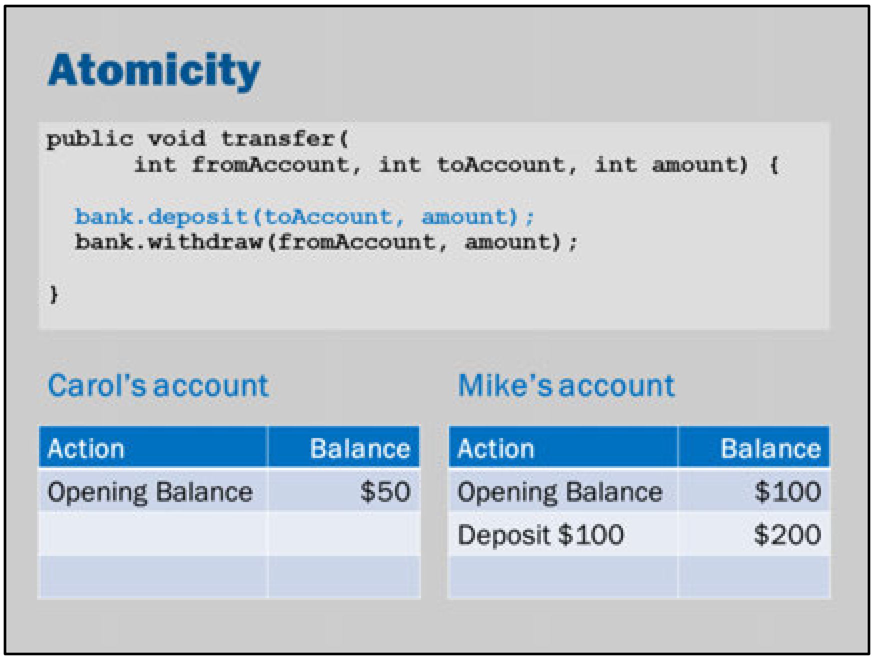
```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

Carol's account

Action	Balance
Opening Balance	$50

Mike's account

Action	Balance
Opening Balance	$100



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

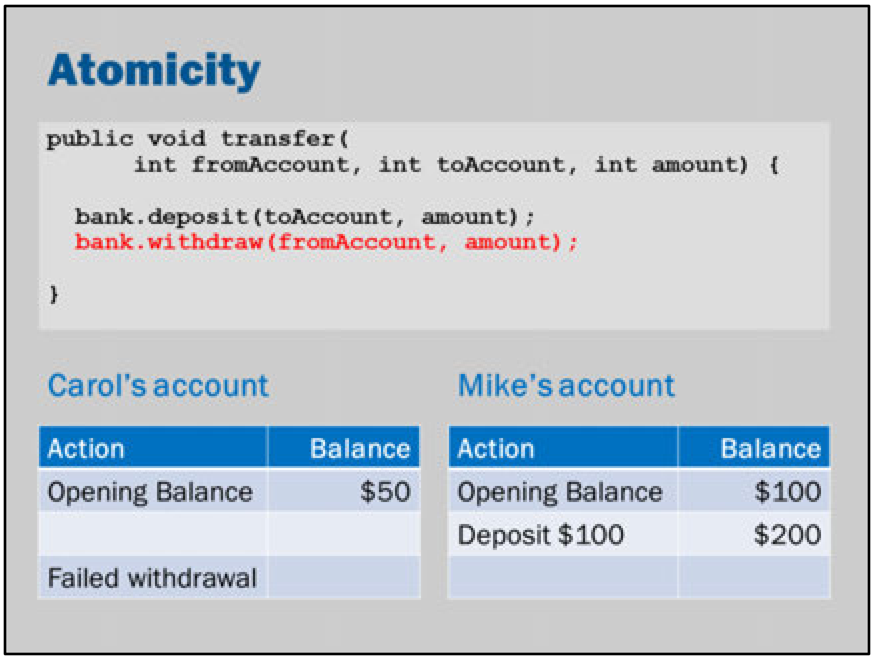
Carol's account

Action	Balance
Opening Balance	$50

Mike's account

Action	Balance
Opening Balance	$100
Deposit $100	$200

If transfer gets called, the first thing that will happen is money will be deposited in the target account.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.deposit(toAccount, amount);
  bank.withdraw(fromAccount, amount);

}
```

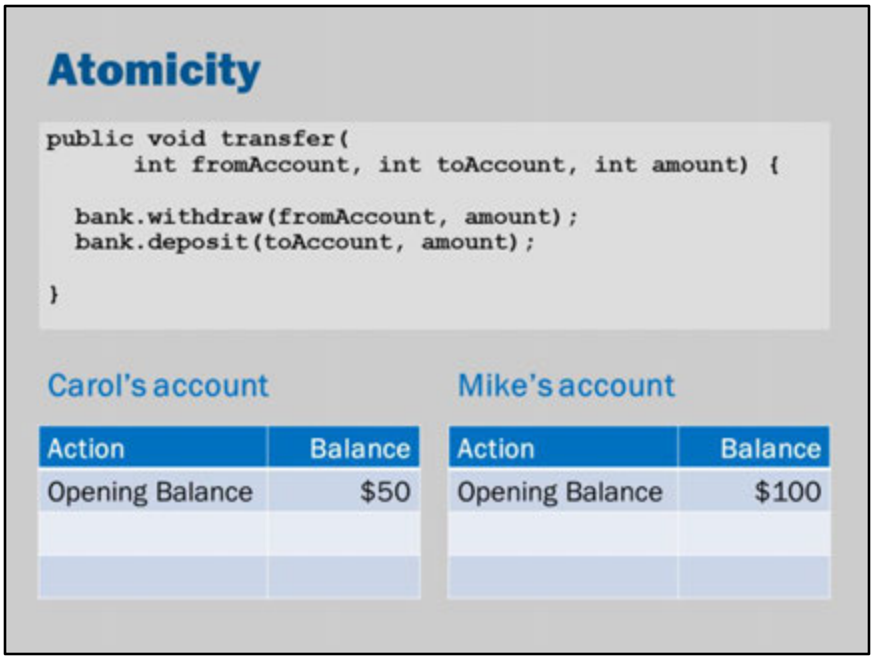
Carol's account

Action	Balance
Opening Balance	$50
Failed withdrawal	

Mike's account

Action	Balance
Opening Balance	$100
Deposit $100	$200

Then the withdrawal fails – there's not enough money.
So now Mike is $100 richer and Carol still has only $50.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.withdraw(fromAccount, amount);
  bank.deposit(toAccount, amount);

}
```

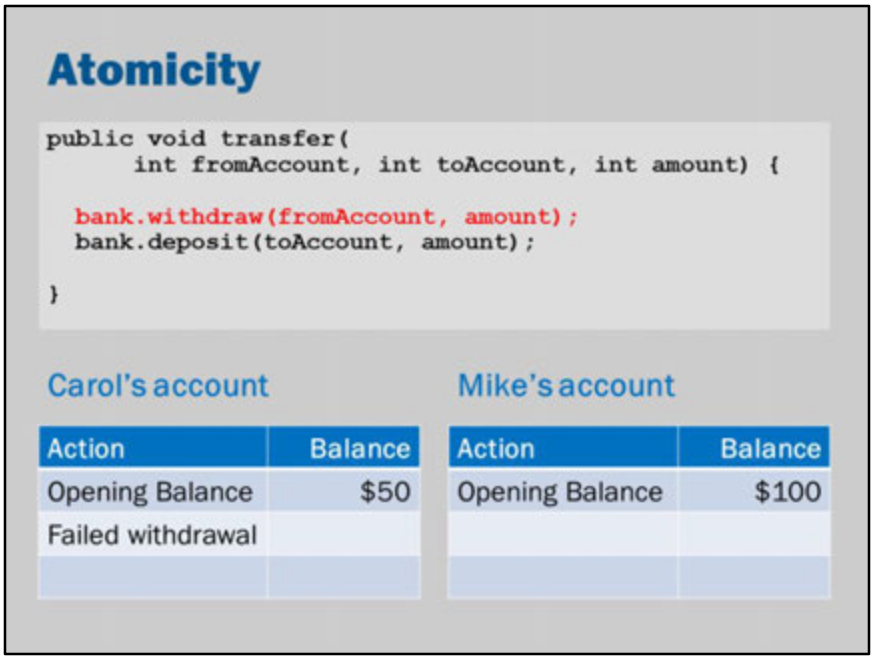
Carol's account

Action	Balance
Opening Balance	$50

Mike's account

Action	Balance
Opening Balance	$100

We could reverse the order.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.withdraw(fromAccount, amount);
  bank.deposit(toAccount, amount);

}
```

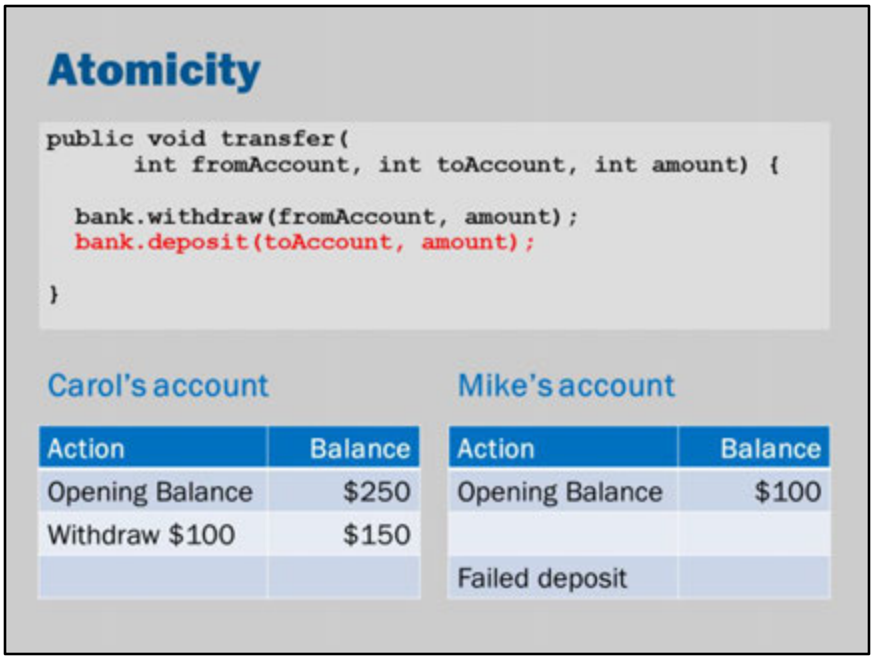
Carol's account

Action	Balance
Opening Balance	$50
Failed withdrawal	

Mike's account

Action	Balance
Opening Balance	$100

That way if Carol's account hasn't got enough money then the transaction will fail.



Atomicity

```
public void transfer(
      int fromAccount, int toAccount, int amount) {

  bank.withdraw(fromAccount, amount);
  bank.deposit(toAccount, amount);

}
```

Carol's account

Action	Balance
Opening Balance	$250
Withdraw $100	$150

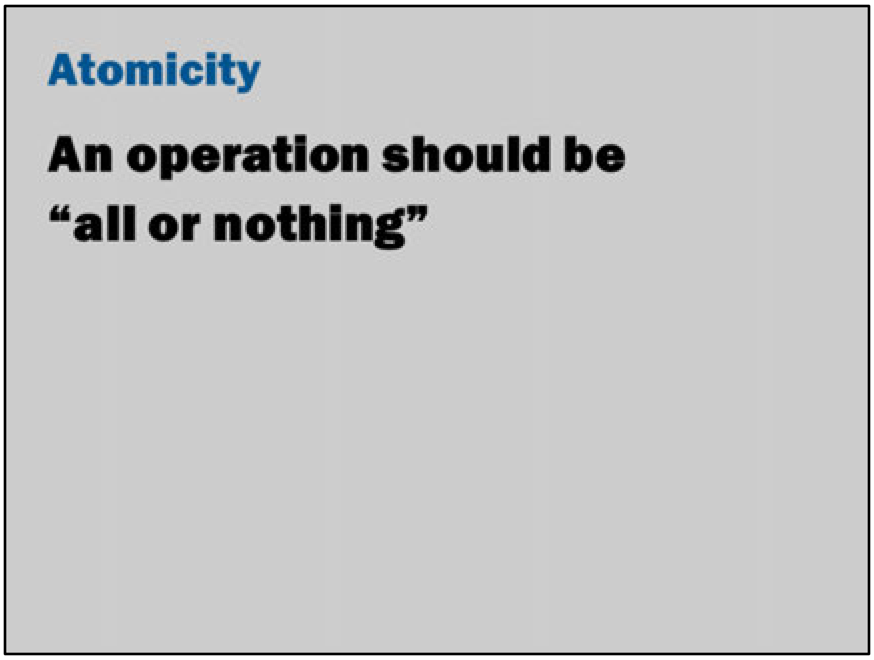
Mike's account

Action	Balance
Opening Balance	$100
Failed deposit	

But suppose that the deposit into Mike's account fails for some other reason:

- The database connection might fail
- The computer might crash
- The computer might lose power
- The toAccount might not be valid
- The system might run out of memory

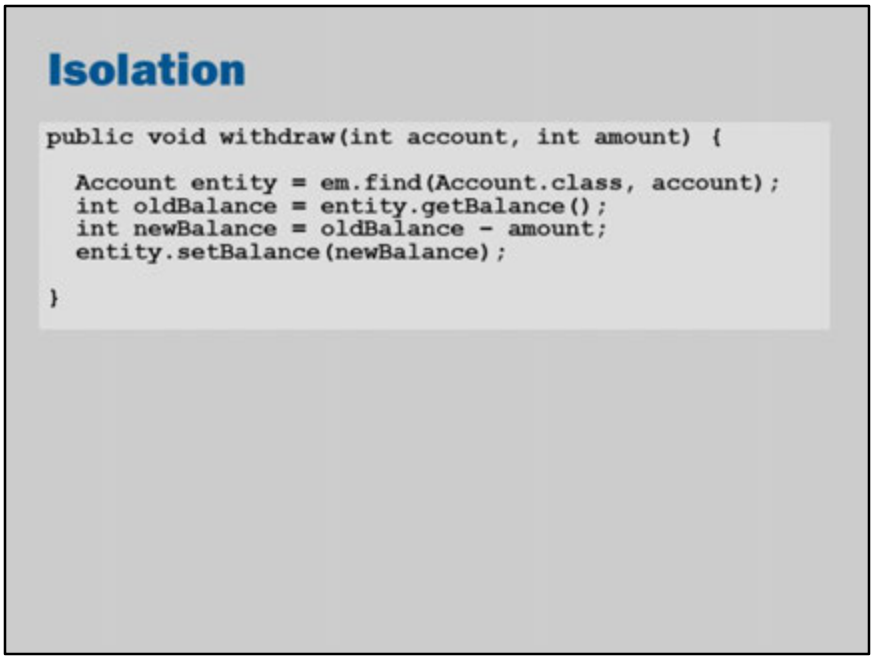
Then the bank will have received $100 of free money.
Obviously this is less bad for the bank, but they'll have some very unhappy customers.



Atomicity

An operation should be "all or nothing"

Ideally, you want the deposit and withdrawal to either both work or both not-work.
You don't want this operation to be half-successful.
You want it to be "all or nothing".



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Even if we ignore the possibility of the withdrawal failing, there are other ways that our system could encounter problems.

Can you see any possibility of problems in this method?

Imagine you're withdrawing money $100 from an account with a balance of $1000. At exactly the same time somebody another transfer is withdrawing $100 from the account.

The first operation may give the current balance of the account.
Account entity = em.find(Account.class, account);
int oldBalance = entity.getBalance();
So, oldBalance will be $1000.

In the other transfer, the current balance of the account is also retrieved.
Account entity = em.find(Account.class, account);
int oldBalance = entity.getBalance();
So, oldBalance will be $1000.

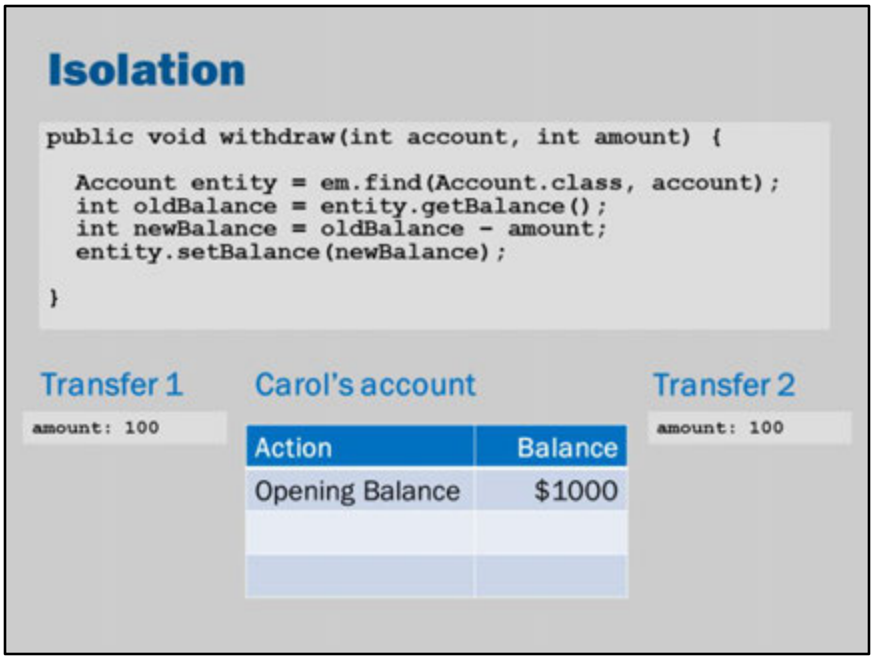
Then, next, in the new transfer, the new balance is correctly calculated.
int newBalance = oldBalance – amount;
So, newBalance will be $900 (i.e., $1000 - $100)

However, in the other transfer, an incorrect balance is calculated:
Then, next, in the new transfer, the new balance is correctly calculated.
int newBalance = oldBalance – amount;
So, newBalance will be $900 (i.e., $1000 - $100)

Finally, when the new balance is set:
entity.setBalance(newBalance);
The account balance is now $900.

And then the other transfer will set exactly the same amount:
entity.setBalance(newBalance);
The final account balance is now $900.

The withdrawal of $100 has been lost! The final account balance should be $800 (instead of just setting $900 twice).



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Transfer 1

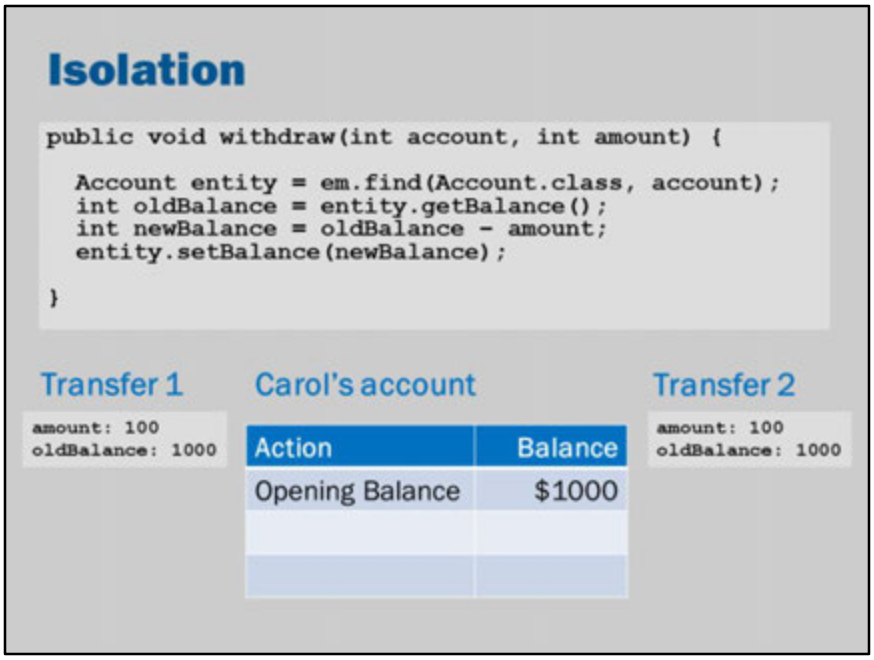
```
amount: 100
```

Carol's account

Action	Balance
Opening Balance	$1000

Transfer 2

```
amount: 100
```



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Transfer 1

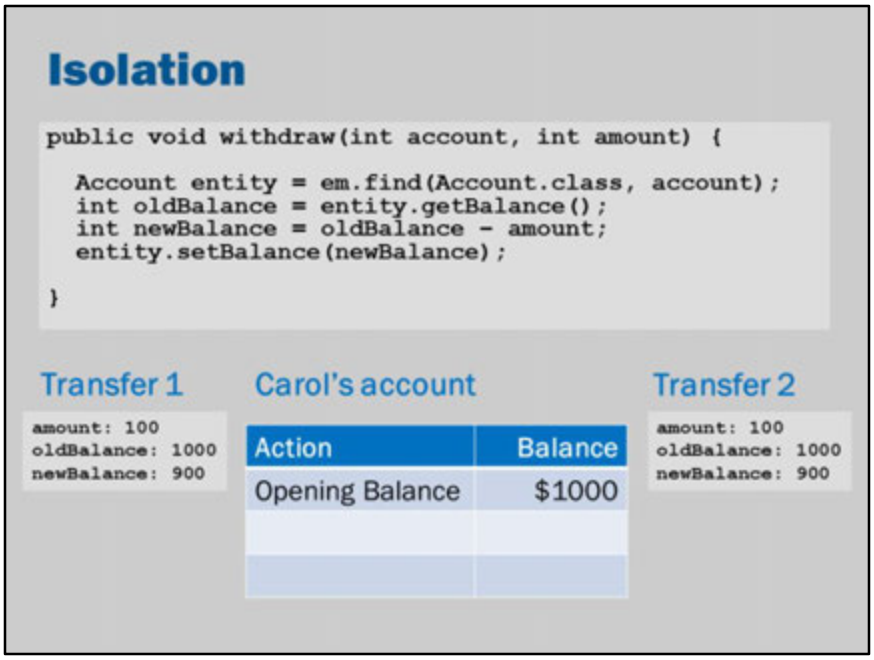
```
amount: 100
oldBalance: 1000
```

Carol's account

Action	Balance
Opening Balance	$1000

Transfer 2

```
amount: 100
oldBalance: 1000
```



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Transfer 1

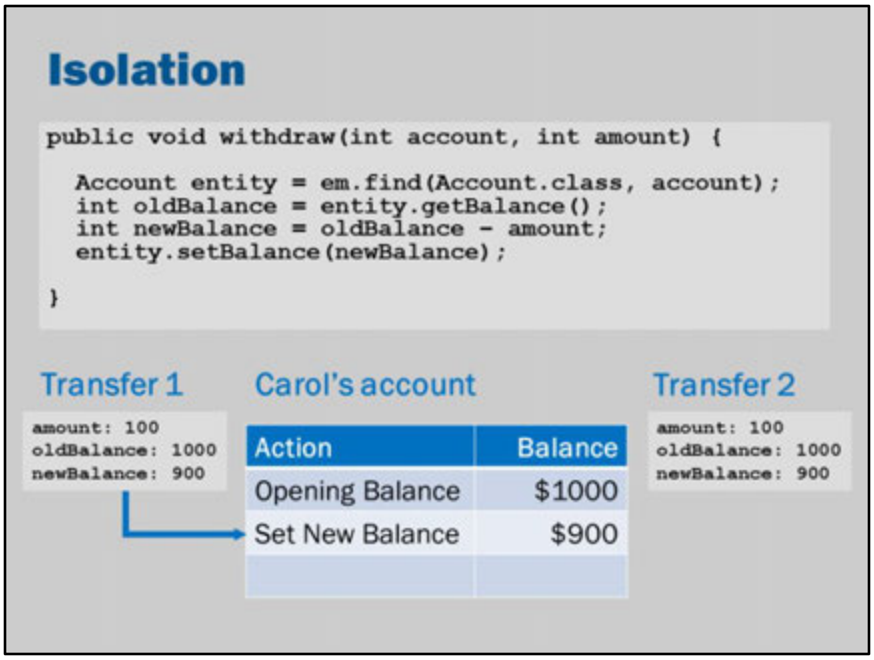
```
amount: 100
oldBalance: 1000
newBalance: 900
```

Carol's account

Action	Balance
Opening Balance	$1000

Transfer 2

```
amount: 100
oldBalance: 1000
newBalance: 900
```



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Transfer 1

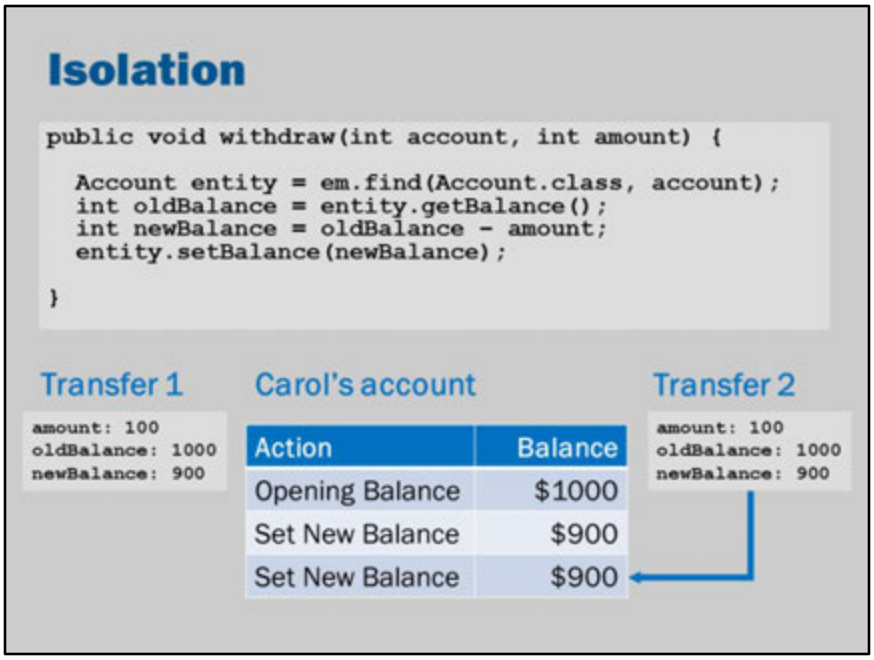
```
amount: 100
oldBalance: 1000
newBalance: 900
```

Carol's account

Action	Balance
Opening Balance	$1000
Set New Balance	$900

Transfer 2

```
amount: 100
oldBalance: 1000
newBalance: 900
```



Isolation

```
public void withdraw(int account, int amount) {

  Account entity = em.find(Account.class, account);
  int oldBalance = entity.getBalance();
  int newBalance = oldBalance - amount;
  entity.setBalance(newBalance);

}
```

Transfer 1

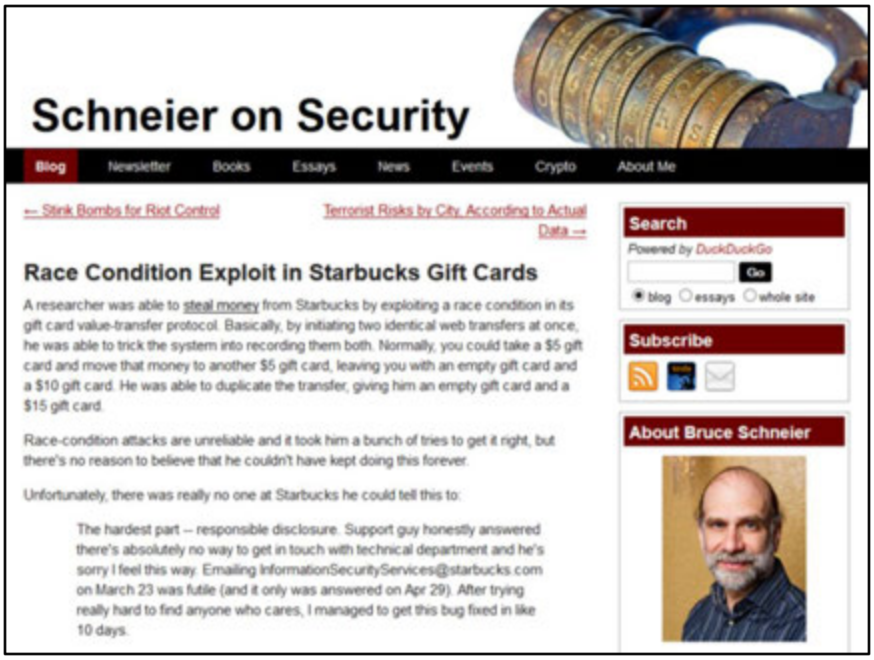
```
amount: 100
oldBalance: 1000
newBalance: 900
```

Carol's account

Action	Balance
Opening Balance	$1000
Set New Balance	$900
Set New Balance	$900

Transfer 2

```
amount: 100
oldBalance: 1000
newBalance: 900
```



Race Condition Exploit in Starbucks Gift Cards

A researcher was able to steal money from Starbucks by exploiting a race condition in its gift card value-transfer protocol. Basically, by initiating two identical web transfers at once, he was able to trick the system into recording them both. Normally, you could take a $5 gift card and move that money to another $5 gift card, leaving you with an empty gift card and a $10 gift card. He was able to duplicate the transfer, giving him an empty gift card and a $15 gift card.

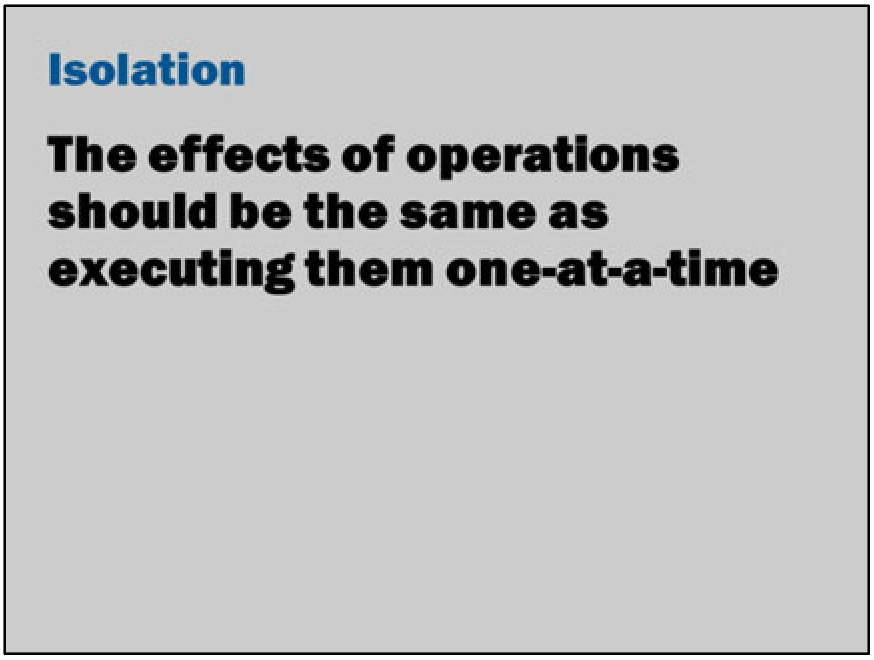
Race-condition attacks are unreliable and it took him a bunch of tries to get it right, but there's no reason to believe that he couldn't have kept doing this forever.

Unfortunately, there was really no one at Starbucks he could tell this to:

> The hardest part -- responsible disclosure. Support guy honestly answered there's absolutely no way to get in touch with technical department and he's sorry I feel this way. Emailing InformationSecurityServices@starbucks.com on March 23 was futile (and it only was answered on Apr 29). After trying really hard to find anyone who cares, I managed to get this bug fixed in like 10 days.

This isn't purely hypothetical.

Starbucks allowed transfers between their gift cards.
A security researcher discovered that they could steal money by executing two transfers at exactly the same time.
https://www.schneier.com/blog/archives/2015/05/race_condition_.html

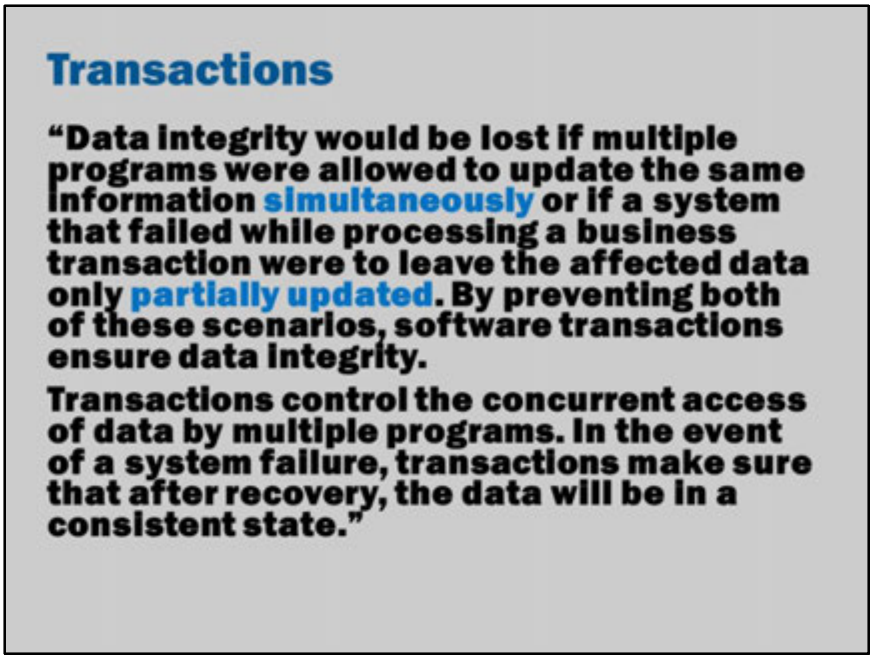


Isolation

The effects of operations should be the same as executing them one-at-a-time

Ideally what you want is that when you're executing multiple operations at once, then you want the outcome to be the same as if you were to execute them one-by-one.

Your code should be able to pretend that it is the only thing that is running in the entire system.



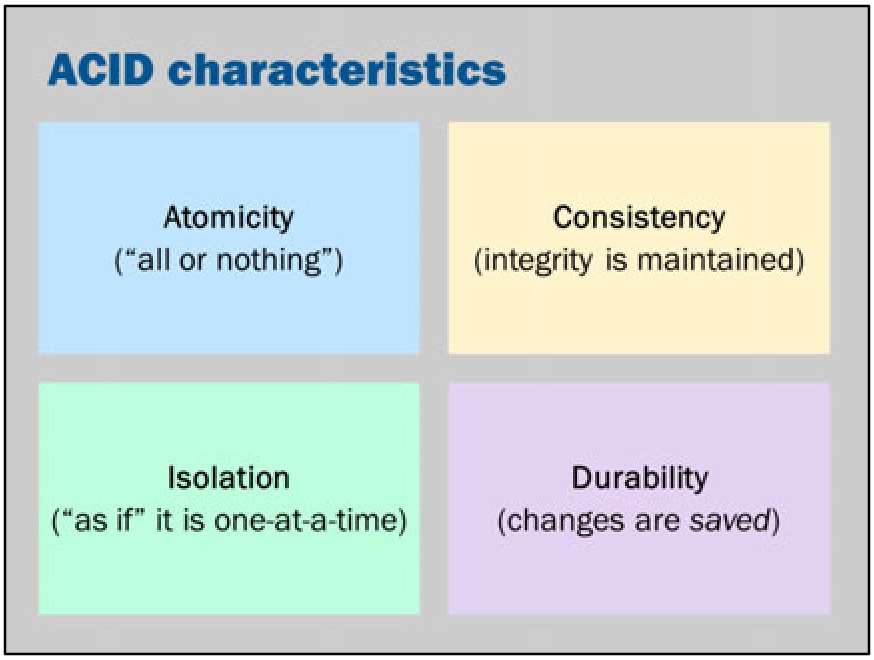
Transactions

"Data integrity would be lost if multiple programs were allowed to update the same information simultaneously or if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity.

Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery, the data will be in a consistent state."

-- http://docs.oracle.com/javaee/7/tutorial/doc/transactions.htm

The key point of transactions is that they:
ensure data is consistent with simultaneous access
and ensure that data is consistent if something goes wrong.



When we talk about transactions, what we mean is an a unit of work that will be executed subject to "ACID" characteristics: atomicity, consistency, isolation and durability.

Atomicity:

- All-or-nothing
- Either the whole operation works or the whole operation fails.
- If something goes wrong during an atomic operation, then every change should be "undone" (i.e., roll back)
- In the bank transfer situation, atomicity would mean that if the withdrawal fails, the database should also automatically reverse the deposit)

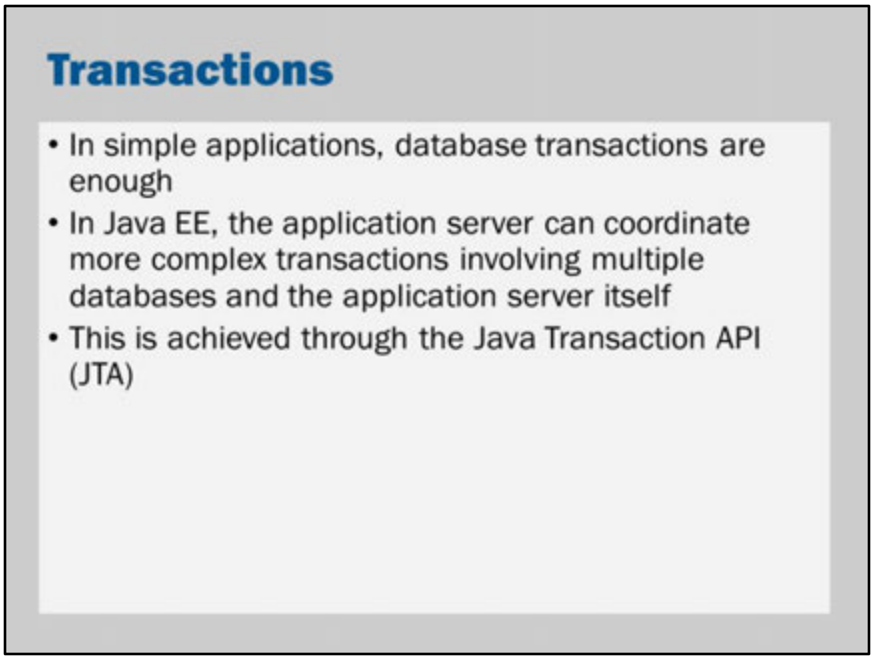
Consistency

- Every change maintains the application constraints: the database will go from one valid state to another valid state.
- Programming errors should not cause the database to enter a state that would violate its integrity constraints.
- Many databases allow you to temporarily violate consistency constraints within a

transaction as long as it is consistent at the end of the transaction: this is to allow you to deal with circular consistency constraints (e.g., if a person must have an office and an office must have a person, you can create the person first and then the office next so long as at the end of the transaction they both exist and refer to each other)

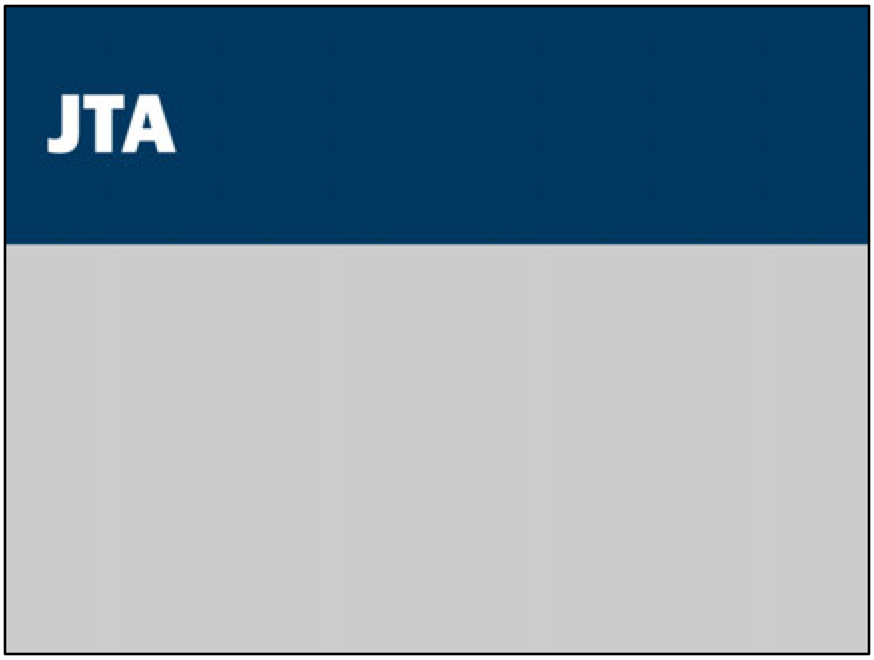
Isolation
- Any operation should work "as-if" it was running on its own
- The result of a sequence of actions should be equivalent to a sequence of actions that might occur if all the actions were performed one-by-one, in serial and without concurrency
- In the bank transfer situation, isolation would have prevented the withdrawal from overwriting the deposit that was made at exactly the same time

Durability
- When a transaction is committed, it stays committed, even if the power is lost or there is a computer crash
- In the bank transfer situation, this means that once the transfer is complete, it should be saved permanently on disk so that it if the computer loses power the transfer does not need to be performed again



Transactions

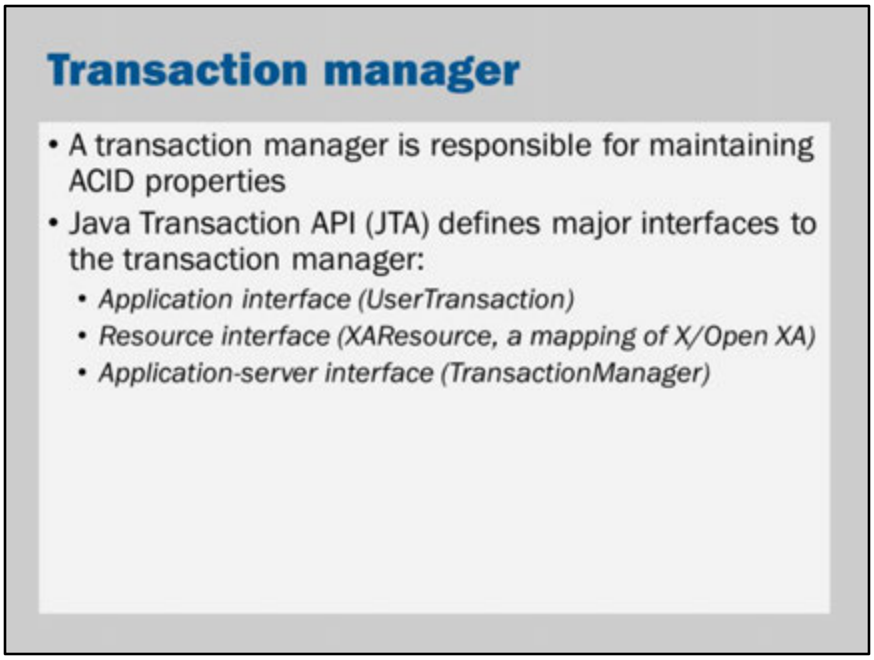
- In simple applications, database transactions are enough
- In Java EE, the application server can coordinate more complex transactions involving multiple databases and the application server itself
- This is achieved through the Java Transaction API (JTA)



JTA

Java EE provides support for transactions.
In fact, you've already been using transactions: by default, Enterprise JavaBeans always run within a transaction.

The Java Transaction API (JTA) is the underlying specification that defines the behavior and interfaces for managing transactions.



Transaction manager

- A transaction manager is responsible for maintaining ACID properties
- Java Transaction API (JTA) defines major interfaces to the transaction manager:
  - *Application interface (UserTransaction)*
  - *Resource interface (XAResource, a mapping of X/Open XA)*
  - *Application-server interface (TransactionManager)*

The core of JTA is the transaction manager.
A transaction manager is responsible for keeping track of the currently running transactions and ensuring that ACID properties are maintained.

The JTA spec defines a number of interfaces.

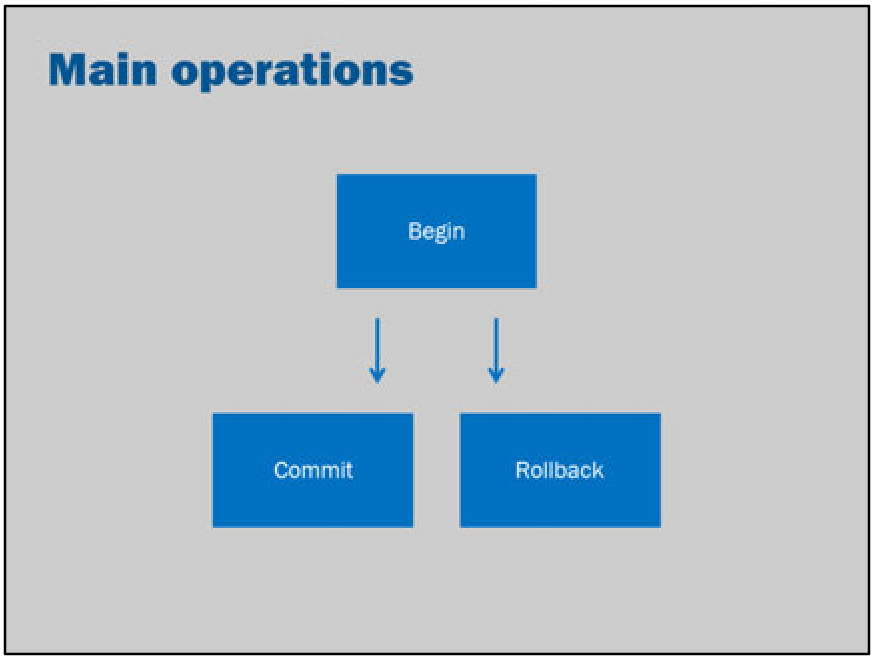
As an application developer, you are most likely to encounter UserTransaction. This is the interface you use when you want to create and manage transactions manually.

XAResource is used by developers who build components for application servers. For example, if you were to create your own Database Management System (e.g., a competitor to Oracle or Microsoft SQL Server), you would implement XAResource so that your database can cooperate with other databases running in the same transaction. XAResource allows the transaction manager to coordinate transactions across multiple systems.

TransactionManager is used by developers who build application servers. It provides a direct interface to the transaction manager.

XA stands for “eXtended Architecture” it is a standard for distributed transactions. It allows your Java Application to participate in complex transactions (involving non-Java systems) over a network.

Advanced reading:
The JTA specification is quite short!
https://java.net/projects/jta-spec/sources/spec-source-repository/content/jta-1_2-spec_v2.pdf?rev=14



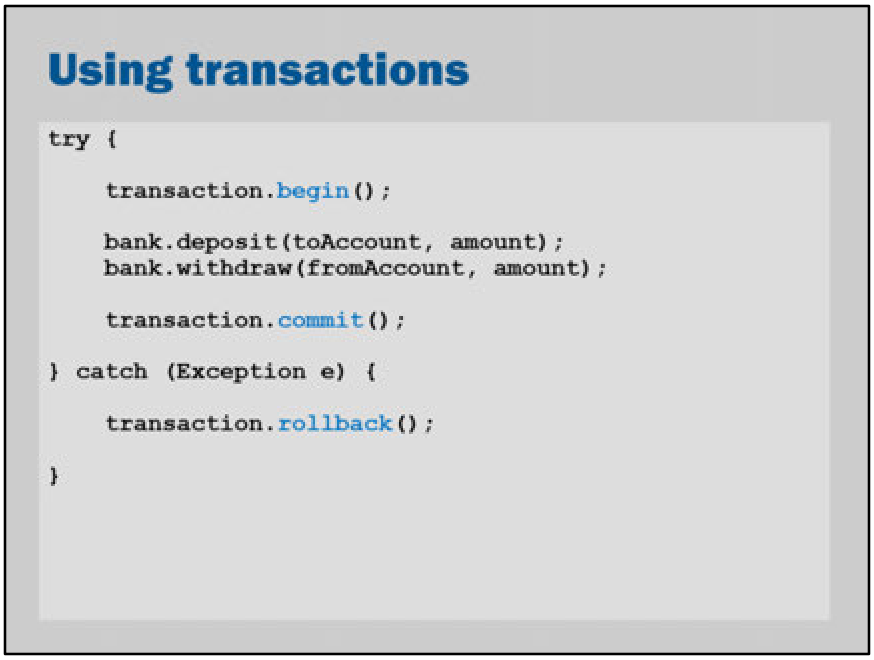
There are just three things you need to know when you're using a transaction:

Begin
This starts the transaction. Anything done between the begin and the commit or roll back should be transactional: it should comply with the ACID characteristics.

Commit
This tells the transaction manager to “save all the changes“. Once commit succeeds, the transaction is over and the effects should be stored durably.

If something goes wrong during commit (e.g., the commit would violate database consistency constraints) then the commit will fail and the changes will be rolled back.

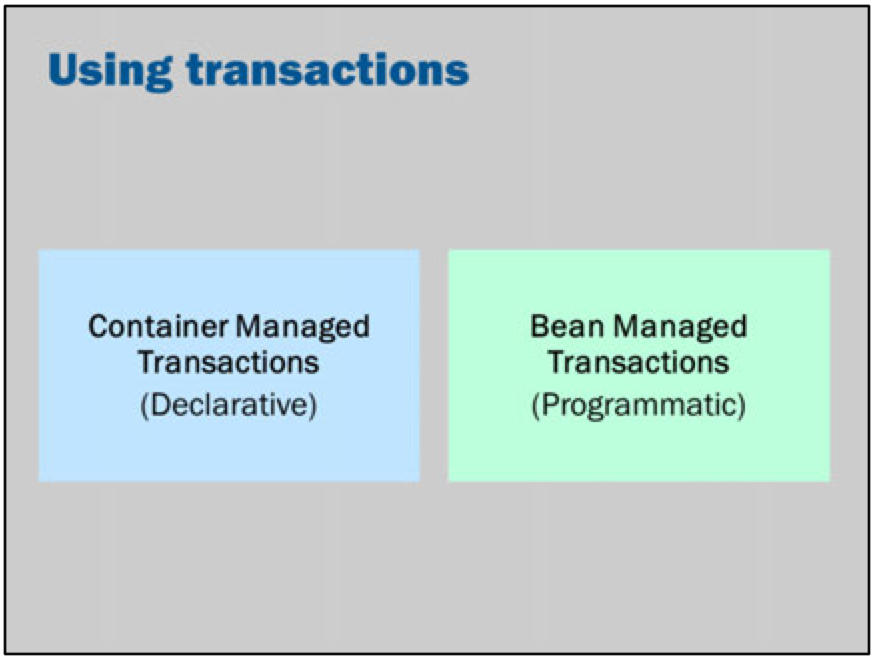
Rollback
This tells the transaction manager to "undo all the changes". The database should be returned back to the way it was before you called begin.



Using transactions

```
try {

    transaction.begin();

    bank.deposit(toAccount, amount);
    bank.withdraw(fromAccount, amount);

    transaction.commit();

} catch (Exception e) {

    transaction.rollback();

}
```

These three operations (Begin, Commit and Rollback) translate directly into methods on UserTransaction: begin(), commit() and rollback().



Using transactions

Container Managed Transactions
(Declarative)

Bean Managed Transactions
(Programmatic)

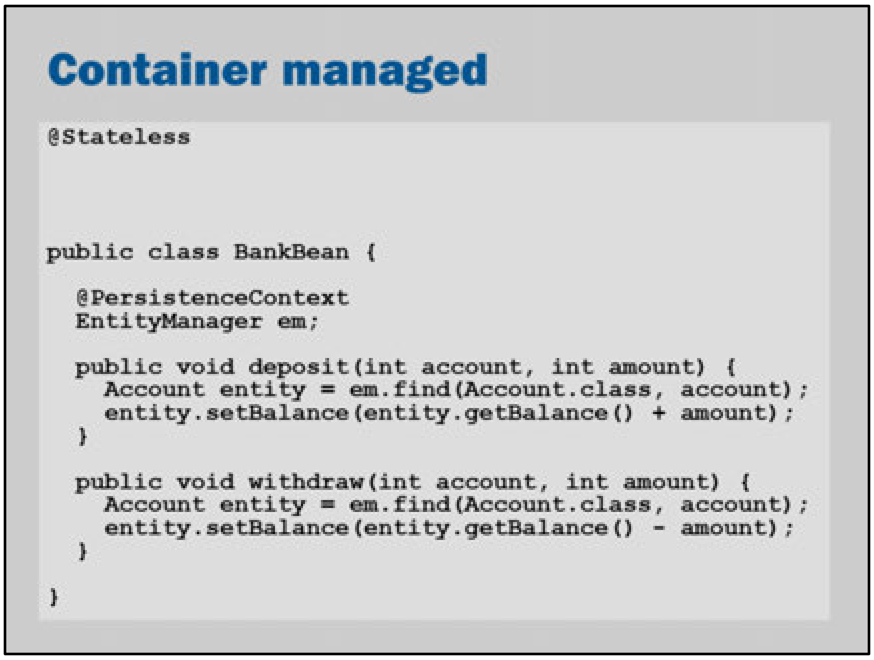
There are two main approaches to using transactions:

Container Managed Transactions
Let the application server (i.e., GlassFish) look after transactions for you. GlassFish will automatically begin the transaction and commit/rollback when you are finished.

This may be referred to as declarative because you "declare" your transaction requirements and the container handles the steps involved.

Bean Managed Transactions
Manually deal with transactions by yourself (i.e., use transaction.begin(), transaction.commit() and transaction.rollback() yourself).

This may be referred to as programmatic because you need to "program" it yourself.



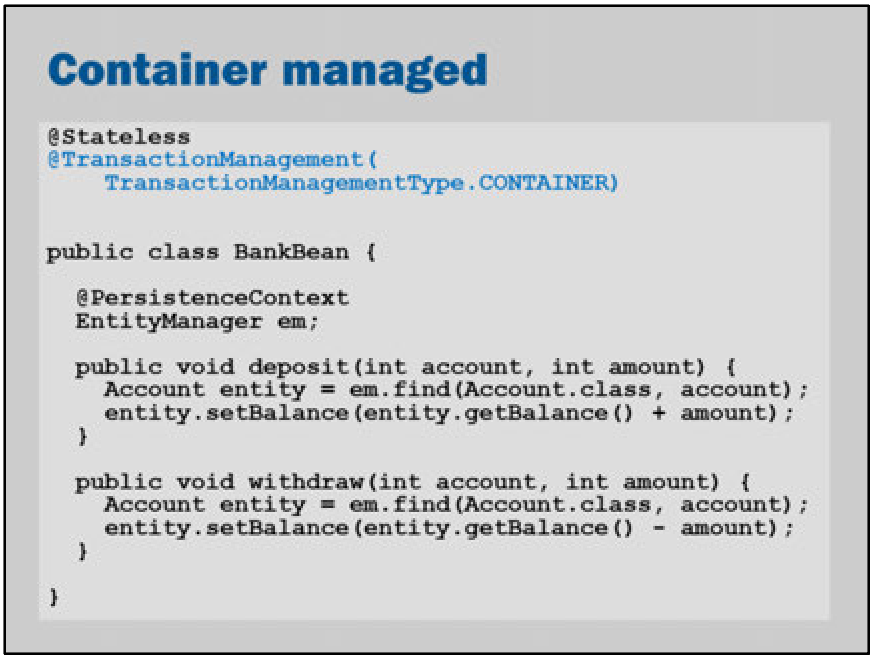
Container managed

```
@Stateless



public class BankBean {

  @PersistenceContext
  EntityManager em;

  public void deposit(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() + amount);
  }

  public void withdraw(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() - amount);
  }

}
```

This is an ordinary stateless session bean.

By default, a stateless session bean uses container managed transactions. This is the default behavior defined in the EJB specifications.

By default, every operation will run in a transaction. This is the default behavior defined in the EJB specifications.

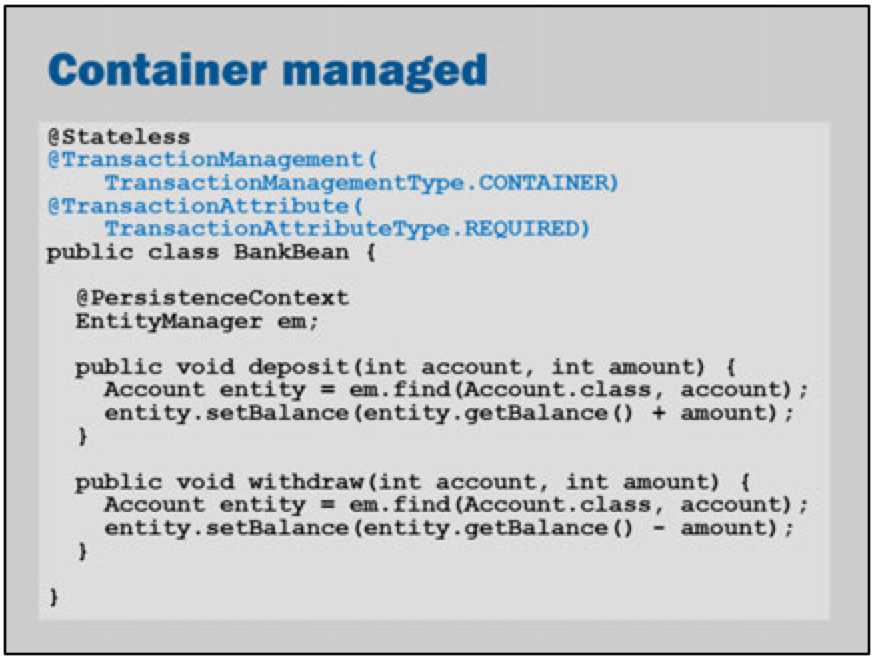


Container managed

```
@Stateless
@TransactionManagement(
    TransactionManagementType.CONTAINER)

public class BankBean {

  @PersistenceContext
  EntityManager em;

  public void deposit(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() + amount);
  }

  public void withdraw(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() - amount);
  }

}
```

In fact, we can explicitly tell the application server that this bean uses container managed transactions (or we can use this annotation to override the default).

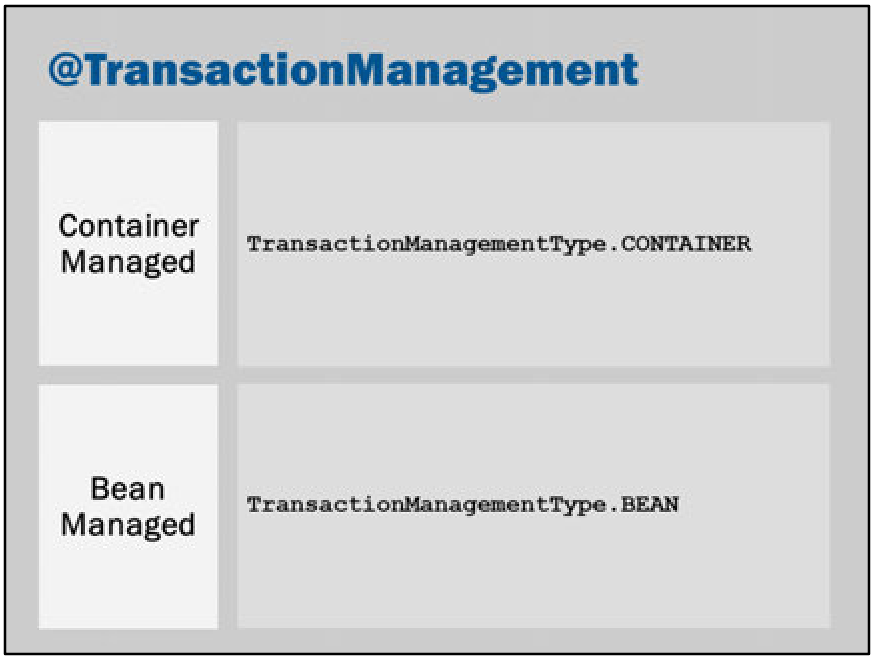
In practice, this annotation is not needed because it is the default.


Container managed

```
@Stateless
@TransactionManagement(
    TransactionManagementType.CONTAINER)
@TransactionAttribute(
    TransactionAttributeType.REQUIRED)
public class BankBean {

  @PersistenceContext
  EntityManager em;

  public void deposit(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() + amount);
  }

  public void withdraw(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() - amount);
  }

}
```

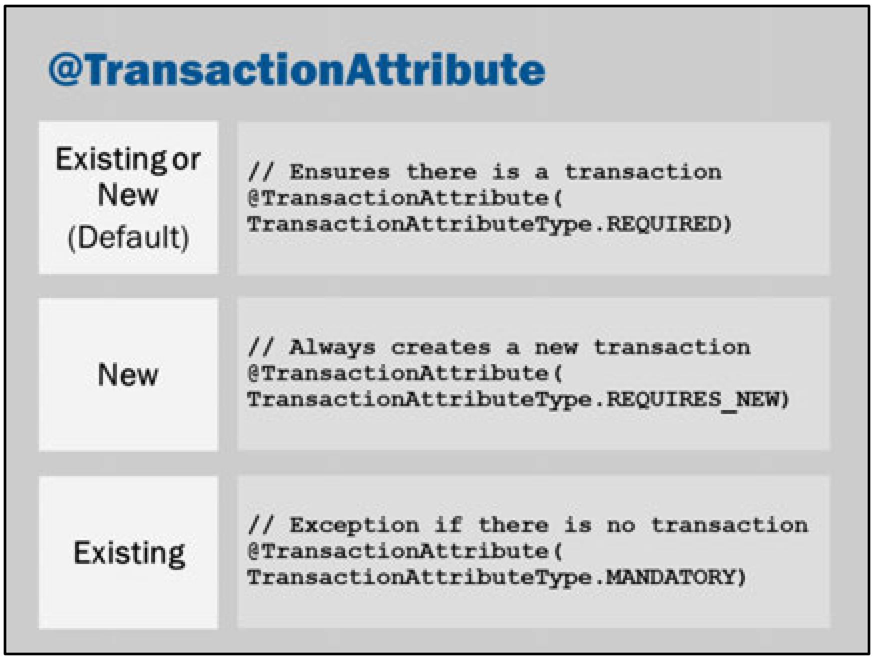
We can also explicitly tell the application server that every operation in this bean needs to run inside a transaction (or we can use this annotation to override the default).

In practice, this annotation is not needed because it is the default.



@TransactionManagement

Container Managed	`TransactionManagementType.CONTAINER`
Bean Managed	`TransactionManagementType.BEAN`



@TransactionAttribute

Existing or New (Default)	`// Ensures there is a transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.REQUIRED)`

New	`// Always creates a new transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.REQUIRES_NEW)`

Existing	`// Exception if there is no transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.MANDATORY)`

The following is an excerpt from https://openejb.apache.org/transaction-annotations.html

“**MANDATORY**

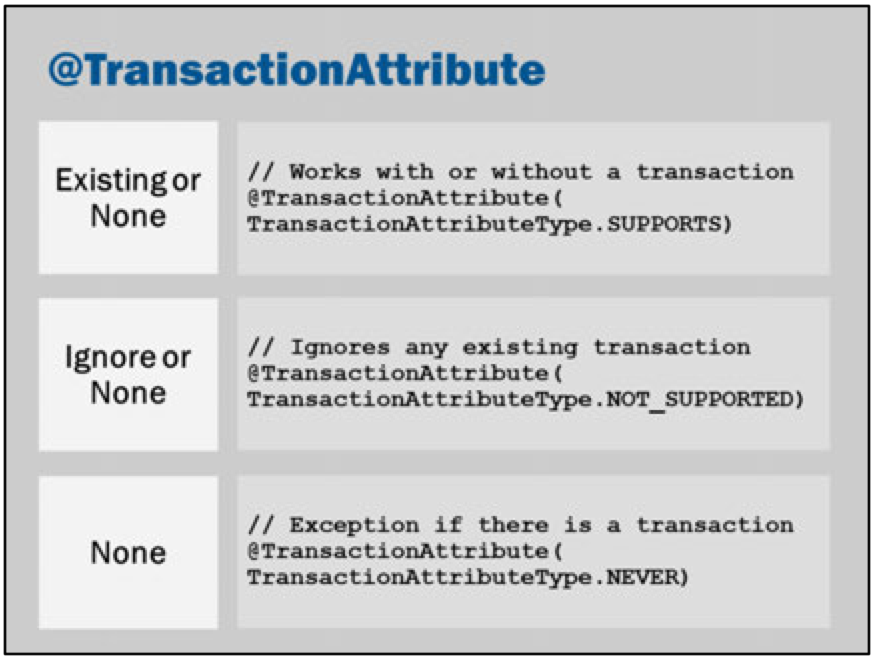
A MANDATORY method is guaranteed to always be executed in a transaction. However, it's the caller's job to take care of supplying the transaction. If the caller attempts to invoke the method outside of a transaction, then the container will block the call and throw them an exception.

REQUIRED

A REQUIRED method is guaranteed to always be executed in a transaction. If the caller attempts to invoke the method outside of a transaction, the container will start a transaction, execute the method, then commit the transaction.

REQUIRES_NEW

A REQUIRES_NEW method is guaranteed to always be executed in a transaction. If the caller attempts to invoke the method inside or outside of a transaction, the container will still start a transaction, execute the method, then commit the transaction. Any transaction the caller may have in progress will be suspended before the method execution then resumed afterward."



@TransactionAttribute

Existing or None	`// Works with or without a transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.SUPPORTS)`

Ignore or None	`// Ignores any existing transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.NOT_SUPPORTED)`

None	`// Exception if there is a transaction`
	`@TransactionAttribute(`
	`TransactionAttributeType.NEVER)`

The following is an excerpt from https://openejb.apache.org/transaction-annotations.html

"**NEVER**

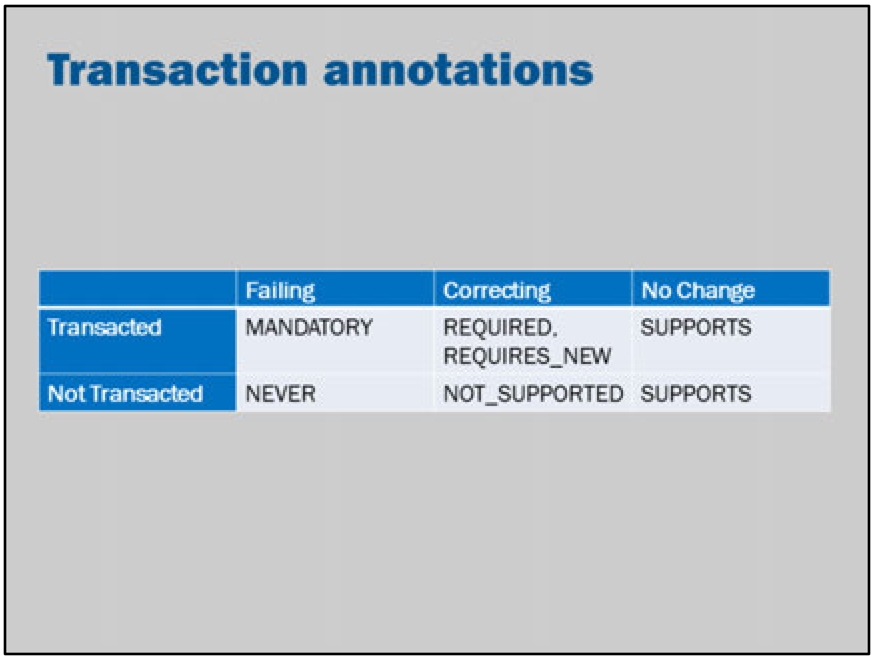
A NEVER method is guaranteed to never be executed in a transaction. However, it's the caller's job to ensure there is no transaction. If the caller attempts to invoke the method inside of a transaction, then the container will block the call and throw them an exception.

NOT_SUPPORTED

A NOT_SUPPORTED method is guaranteed to never be executed in a transaction. If the caller attempts to invoke the method inside of a transaction, the container will suspend the caller's transaction, execute the method, then resume the caller's transaction.

SUPPORTS

A SUPPORTS method is guaranteed to adopt the exact transactional state of the caller. These methods can be invoked by caller's inside or outside of a transaction. The container will do nothing to change that state.”



Transaction annotations

	Failing	Correcting	No Change
Transacted	MANDATORY	REQUIRED, REQUIRES_NEW	SUPPORTS
Not Transacted	NEVER	NOT_SUPPORTED	SUPPORTS

--https://openejb.apache.org/transaction-annotations.html

“The "Transacted" and "Not Transacted" categories represent the container guarantee, i.e. if the bean method will or will not be invoked in a transaction. The "Failing", "Correcting", and "No Change" categories represent the action take by the container to achieve that guarantee.

For example, Never and Mandatory are categorized as "Failing" and will cause the container to throw an exception to the caller if there is (Tx Never) or is not (Tx Mandatory) a transaction in progress when the method is called. The attributes Required, RequiresNew, and NotSupported are categorized as "Correcting" as they will cause the container to adjust the transactional state automatically as needed to match the desired state, rather than blocking the invocation by throwing an exception.”

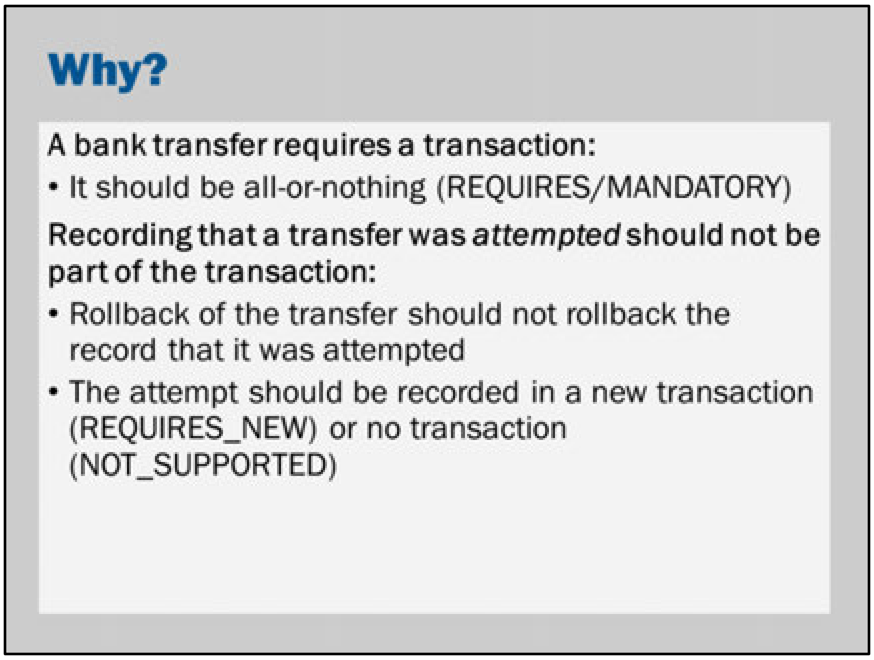
Why?

Why do we need these transaction annotations?

Because when we write our business logic, we may require different transactional guarantees. Some objects may require transactions, some may require no transactions and for others is does not matter.

For example, the bank transfer clearly needs to be in a transaction.

However, you may also like to record attempted bank transfers. If you try to record all attempts in the database, when the transaction fails the record of the attempt will also be removed. Thus, you might want to log the attempt in a different transaction (REQUIRES_NEW) or in no transaction (NOT_SUPPORTED) even though the main operation is performed in a transaction (REQUIRES or MANDATORY).



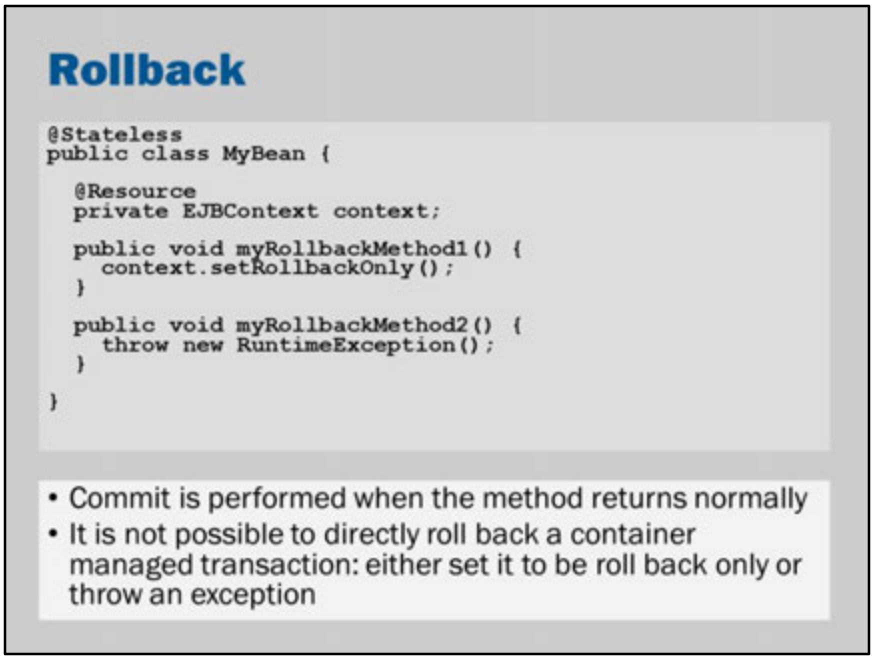
Why?

A bank transfer requires a transaction:

- It should be all-or-nothing (REQUIRES/MANDATORY)

Recording that a transfer was *attempted* should not be part of the transaction:

- Rollback of the transfer should not rollback the record that it was attempted
- The attempt should be recorded in a new transaction (REQUIRES_NEW) or no transaction (NOT_SUPPORTED)



Rollback

```
@Stateless
public class MyBean {

  @Resource
  private EJBContext context;

  public void myRollbackMethod1() {
    context.setRollbackOnly();
  }

  public void myRollbackMethod2() {
    throw new RuntimeException();
  }

}
```

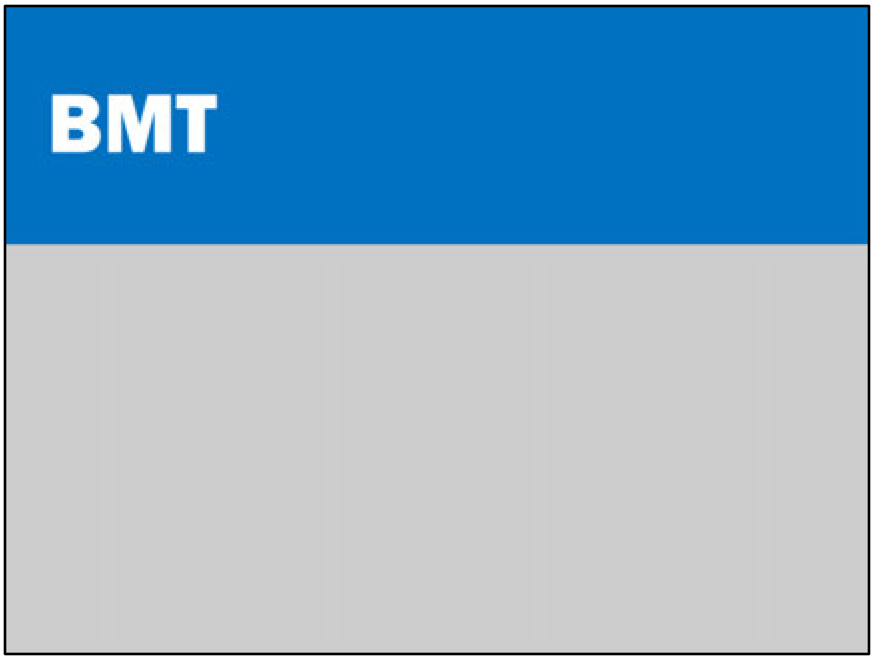
- Commit is performed when the method returns normally
- It is not possible to directly roll back a container managed transaction: either set it to be roll back only or throw an exception

You can roll back a container managed transaction in two ways:

setRollbackOnly
Calling this method will tell the container that the transaction must not be committed. It can only be rolled back. The container decides when and how to perform commit/rollback, so it is an error to directly roll back the transaction. However, setRollbackOnly tells the container that it may only “choose” to do a rollback.

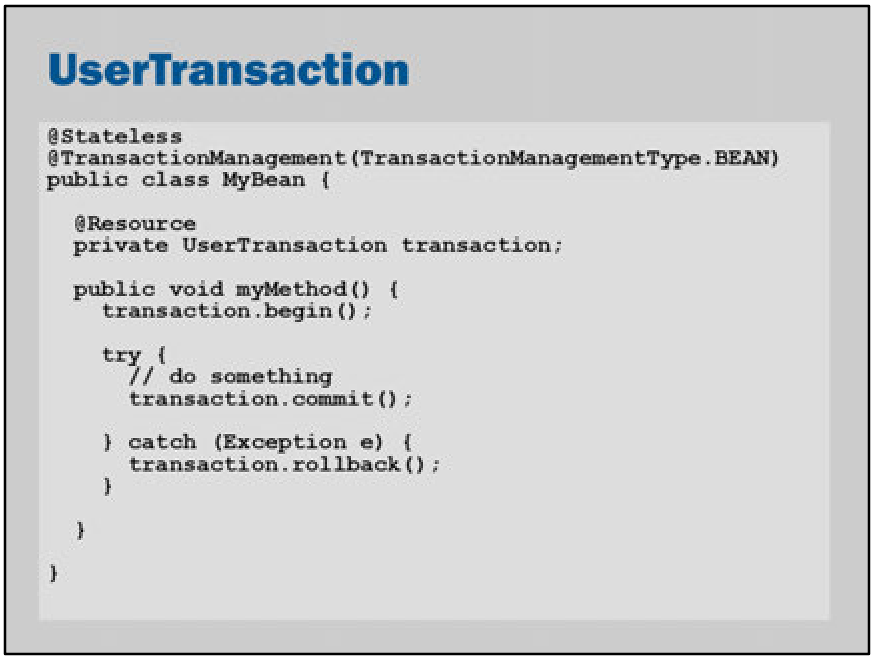
Note: setRollbackOnly should NOT be called in a bean managed transaction.

Throwing a System Exception
Throwing a RuntimeException or an EJBException will cause a rollback.
Throwing a user-defined application exception will not cause a rollback (unless the exception is annotated with @ApplicationException(rollback=true))



BMT

Bean Managed Transactions



UserTransaction

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class MyBean {

  @Resource
  private UserTransaction transaction;

  public void myMethod() {
    transaction.begin();

    try {
      // do something
      transaction.commit();

    } catch (Exception e) {
      transaction.rollback();
    }

  }

}
```

There are less things to know about when using bean managed transactions.

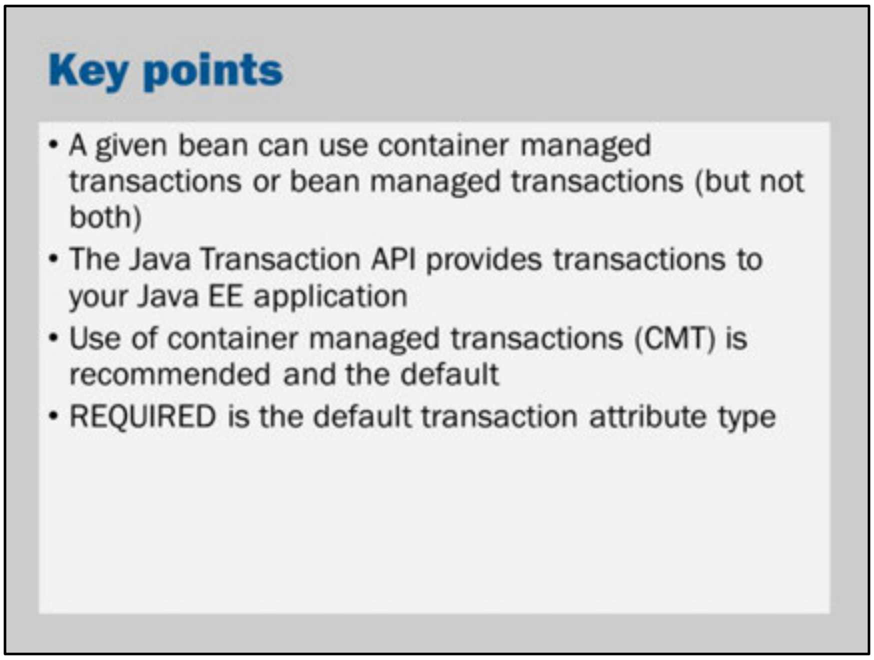
You simply tell the transaction manager when to begin, commit and rollback your transactions. To do this you use the interface provided by UserTransaction.

Be careful. Bean managed transactions may look simple. However, they are much more difficult to use because you need to be very careful about ensuring that you've handled every possible outcome.

Advanced notes:
Generally, it is better to stick to either container managed transactions or bean managed transactions, and not mix the two.

However, you can mix the two:
When bean managed transaction code calls code that uses container managed transactions, then the container managed transactions will reuse the bean-managed transactions.

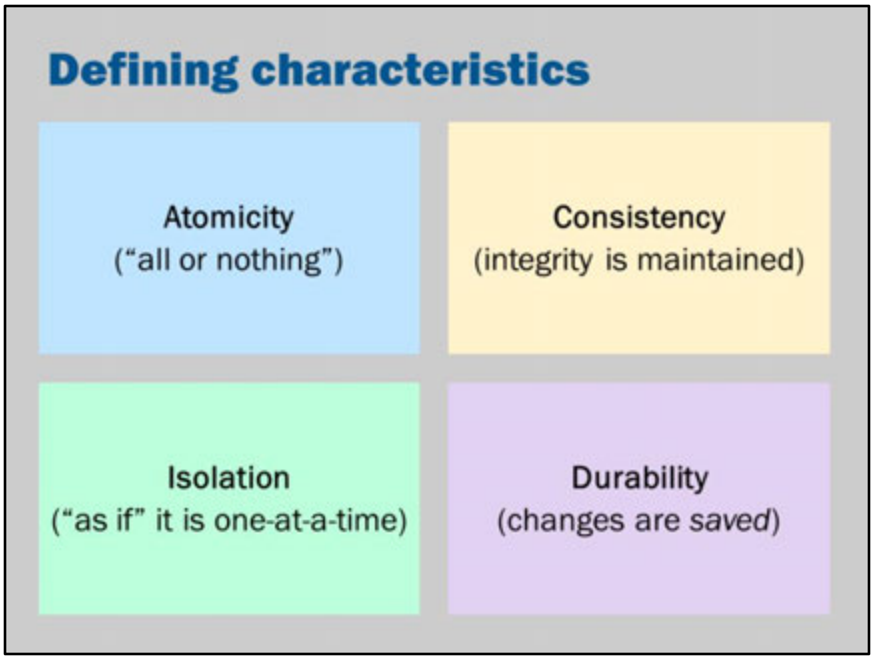
When container managed transaction code calls code that uses bean managed transactions, then the container will suspend any transactions that it created. The bean managed transaction code can't reuse the container managed transactions.



Key points

- A given bean can use container managed transactions or bean managed transactions (but not both)
- The Java Transaction API provides transactions to your Java EE application
- Use of container managed transactions (CMT) is recommended and the default
- REQUIRED is the default transaction attribute type

Weaker isolation


Defining characteristics

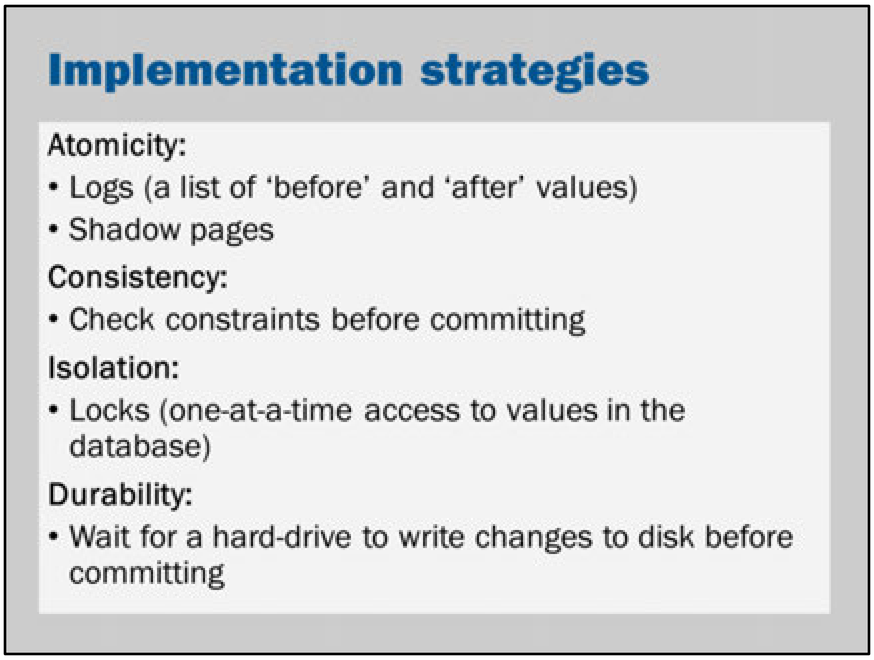
Atomicity ("all or nothing")	Consistency (integrity is maintained)
Isolation ("as if" it is one-at-a-time)	Durability (changes are *saved*)

So far, most of our discussion has been about atomicity.
However, transactions enforce all of the ACID characteristics.

Isolation refers to the idea that you can write your code "as-if" it was running on its own. In theory, when you write code that uses a transaction you can pretend that it is the only code running at a given point of time. The database and application server will use locks or other concurrency management techniques to ensure that the effects of any transactions that run simultaneously would be equivalent to those transactions being run in some sequential order.

The ideal goal of transactions is to achieve "serializability". You can have multiple transactions running at the same time, but the end result of the transactions running at the same time should be equivalent to some sequence of the same transactions running one after the other.

The easiest way to achieve serializability would be to enforce a one-transaction-at-a-time rule. However, this would be impractical. It would make the application extremely slow when you have many users. Instead, the application server allows multiple transactions at the same time – as long as the transactions are reading and writing different data, there is no problem. If more than one transaction needs to read/write data that another transaction is manipulating, then the application server may make one of the transactions wait until the other is finished.



Implementation strategies

Atomicity:
- Logs (a list of 'before' and 'after' values)
- Shadow pages

Consistency:
- Check constraints before committing

Isolation:
- Locks (one-at-a-time access to values in the database)

Durability:
- Wait for a hard-drive to write changes to disk before committing

Atomicity
Atomicity can be implemented using transactions logs.
A log keeps track of previous and updated values of every change.
This means that the log can be used to fix up any errors:
https://en.wikipedia.org/wiki/Transaction_log

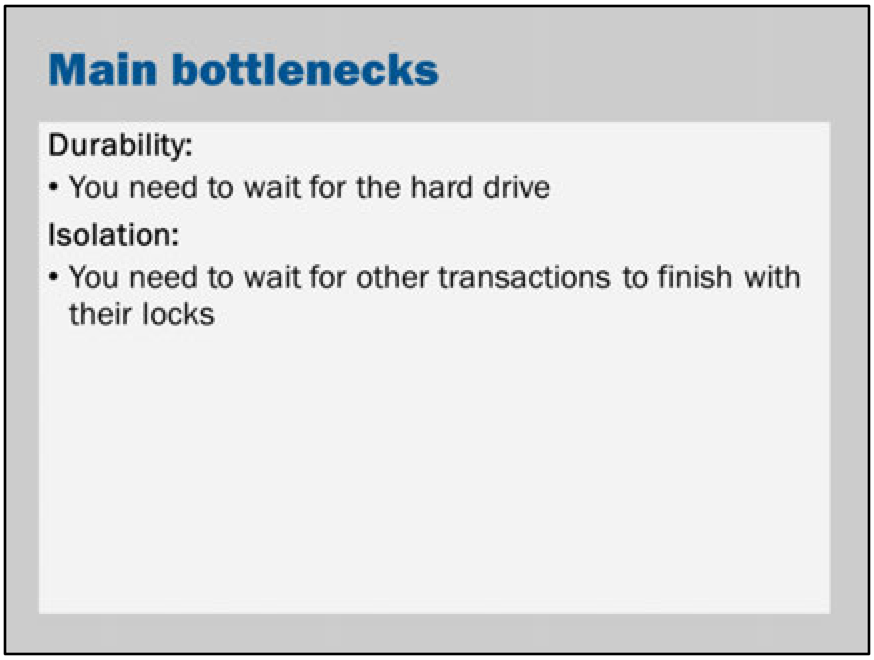
Atomicity can also be implemented using shadow pages:
https://en.wikipedia.org/wiki/Shadow_paging

Consistency
Consistency should be fairly straightforward: the database can just checks that constraints are satisfied before allowing a commit to occur.

Isolation
Isolation is typically implemented using locks:
https://en.wikipedia.org/wiki/Lock_%28database%29
A lock gives a transaction an exclusive right to read or write to a record in a database.
If another transaction needs to make changes to the same record, then it will need to wait until the locks are freed before it can obtain its own locks and then modify the

databse.

Durability
Durability can be achieved by waiting to be sure that the hard disk has written the transaction log before confirming that the transaction has been committed.



Main bottlenecks

Durability:

- You need to wait for the hard drive

Isolation:

- You need to wait for other transactions to finish with their locks

Atomicity
If you think about it, atomicity shouldn't be much of a problem for performance.
To get atomicity you just write any changes to a log.
There's nothing in writing to a log that would inherently slow down a database.

If you've got multiple servers involved in a transaction, then there will be some overhead in getting those servers to agree to a commit (using a 2-phase commit) but there's no fundamental bottleneck here:
https://en.wikipedia.org/wiki/Two-phase_commit_protocol

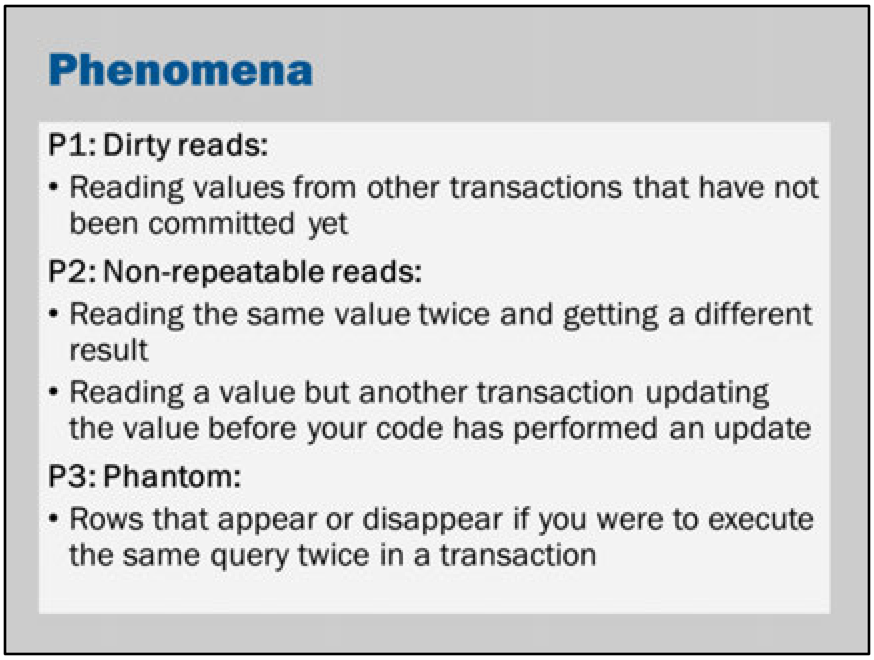
Consistency
Consistency requires checking integrity constraints.
There's nothing fundamentally blocking here.
With many servers in your database, you could even speed up consistency checks by running different checks on different servers.

Isolation
Locks are fundamentally about forcing transactions to wait until another transaction has finished.

The database just has to wait until the transaction is finished.
Adding extra servers doesn't make a database "wait faster".
The delays come from outside the database.
So this is a big bottleneck for performance.

Durability
To ensure durability, you need to wait until the hard-disk has confirmed that the data has been written to disk.
Hard disks tend to be quite slow but they can write a large amount of data in bulk.
This means that you could durably commit many transactions simultaneously, but there is some latency waiting for the hard-disk to confirm the write.
So, durability is a small bottleneck that will affect response time, but it doesn't have a significant impact on the overall performance or throughput of the system.

Advanced note:
Actually, database will use different types of locks. It may use read and write locks.
A read lock allows multiple transactions to read, but none to write.
A write lock only allows one transaction to read or write.
These locks may be at the row level, query level or the table level.
Table level locks create more performance problems because they apply to the entire table.



Phenomena

P1: Dirty reads:
- Reading values from other transactions that have not been committed yet

P2: Non-repeatable reads:
- Reading the same value twice and getting a different result
- Reading a value but another transaction updating the value before your code has performed an update

P3: Phantom:
- Rows that appear or disappear if you were to execute the same query twice in a transaction

Ensuring isolation can create significant performance problems.
This isn't because ensuring serializability with locks fundamentally requires waiting on another transaction to complete.
It isn't because the GlassFish application server is slow.

Isolation creates performance problems because serializability causes “bottlenecks”.
Most databases use locks to ensure serialiability.
Locks (like the Java keyword “synchronized”), only allow one operation at a time.
If data in the database has been locked, then other transactions will need to wait if they are using the same data.

Sometimes, however, you may not need full isolation.
You may be using transactions for atomicity, consistency and durability.
However, in your business application, you may not need to ensure isolation.
For example, if you were building a social network, it might not matter much if the number of "likes" is slightly wrong.

So one solution is to just reduce the amount of isolation.
Rather than guaranteeing that every transaction works perfectly "as though" it were

running on its own, with weaker isolation the database might only provide some guarantees.

If you don't have full isolation (i.e., if you don't use locks), then there are a number of problems that can occur when running database queries…

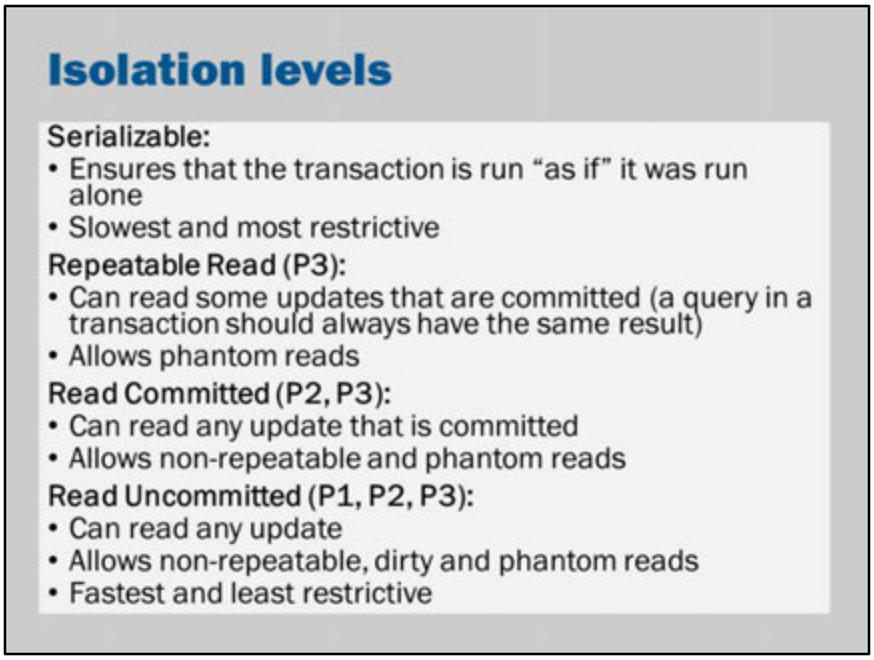
This is what the SQL 1992 standard says:

The isolation level specifies the kind of phenomena that can occur during the execution of concurrent SQL-transactions. The following phenomena are possible:

1) P1 ("Dirty read"): SQL-transaction T1 modifies a row. SQL-transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed.

2) P2 ("Non-repeatable read"): SQL-transaction T1 reads a row. SQL-transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.

3) P3 ("Phantom"): SQL-transaction T1 reads the set of rows N that satisfy some <search condition>. SQL-transaction T2 then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.


Isolation levels

Serializable:
- Ensures that the transaction is run "as if" it was run alone
- Slowest and most restrictive

Repeatable Read (P3):
- Can read some updates that are committed (a query in a transaction should always have the same result)
- Allows phantom reads

Read Committed (P2, P3):
- Can read any update that is committed
- Allows non-repeatable and phantom reads

Read Uncommitted (P1, P2, P3):
- Can read any update
- Allows non-repeatable, dirty and phantom reads
- Fastest and least restrictive

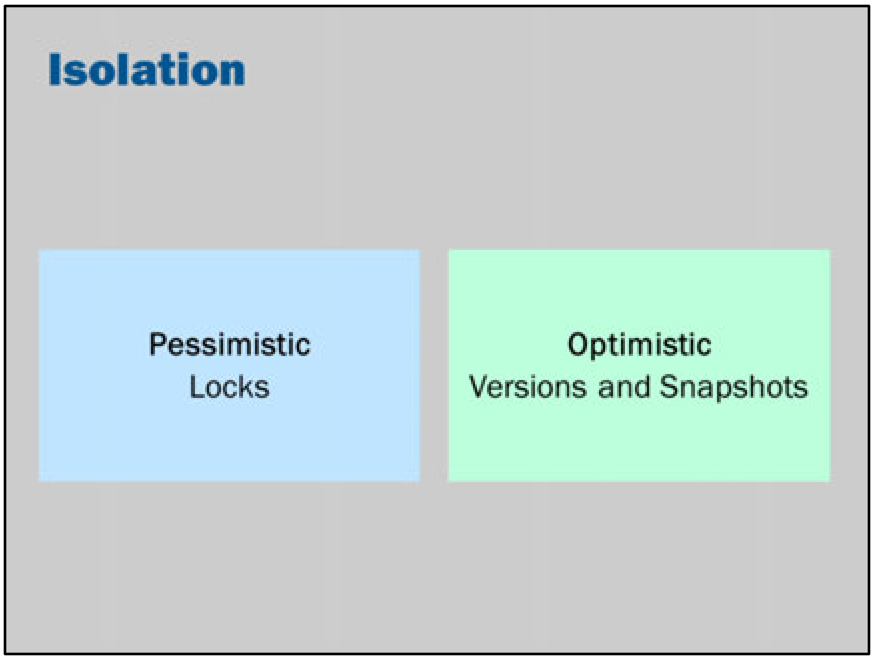
So, you might decide to accept some of those phenomena.
That is, you might decide to reduce the amount of isolation to improve performance.

In fact, JavaDB and many other JDBC databases allow you to configure the transaction isolation level. You can set this in GlassFish from the JDBC Connection Pool settings.

Using Read Uncommitted provides the lowest level of guarantees – it allows data to be updated at maximum performance.

Serializable provides the highest level of guarantees. However, these guarantees have a performance impact.

In standard SQL, the transaction isolation level can be set as follows:
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SET TRANSACTION ISOLATION LEVEL READ REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL READ SERIALIZABLE



Isolation

Pessimistic
Locks

Optimistic
Versions and Snapshots

Isolation can be achieved in two ways.

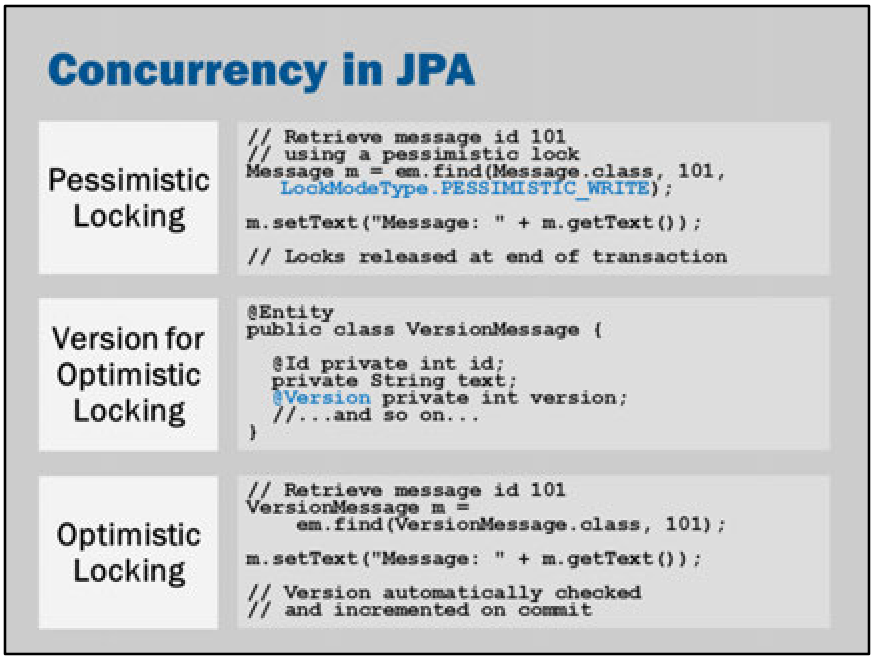
Most major databases (JavaDB, Oracle, Microsoft SQL Server, Sybase) use locking strategies by default.
This is the “classic” approach.
A lock prevents multiple transactions from modify the same data at the same time.
When a transaction needs to modify data, it “acquires” a lock associated with the data.
When the transaction commits (or rolls back) it “releases” that lock.
If another transaction needs to modify the same data, it will attempt to “acquire” that same lock.
However, it will be made to wait (it is said to *block*) until the first transaction releases its lock.
This is referred to as a pessimistic strategy because it assumes the worst: it ensures that non-serializable transactions cannot occur.

Other database systems, use an optimistic strategy.
They optimistically allow updates to occur immediately; non-serializable transactions

can occur, but the container/database will roll them back when it detects it.
They allow any updates to occur during a transaction, without any delay due to locking.
When the transaction is about to commit, the database or transaction manager
Thus, optimistic strategies typically have much higher performance.
They work best when transactions modify different data/records so that they are already isolated from each other – when the transactions they do not require the container to provide isolation.
However, they are prone to causing lots of roll backs if many transactions simultaneously modify the same data/records.

When you use JavaDB in GlassFish, a pessimistic strategy is used by default.

It is possible to use optimistic strategies when using JPA. This is the purpose of the @Version annotation on a JPA Entity.



Concurrency in JPA

Pessimistic Locking

```
// Retrieve message id 101
// using a pessimistic lock
Message m = em.find(Message.class, 101,
    LockModeType.PESSIMISTIC_WRITE);

m.setText("Message: " + m.getText());

// Locks released at end of transaction
```

Version for Optimistic Locking

```
@Entity
public class VersionMessage {

  @Id private int id;
  private String text;
  @Version private int version;
  //...and so on...
}
```

Optimistic Locking

```
// Retrieve message id 101
VersionMessage m =
    em.find(VersionMessage.class, 101);

m.setText("Message: " + m.getText());

// Version automatically checked
// and incremented on commit
```

JPA assumes that it is running on a database with READ COMMITTED isolation. Even if you reconfigure your database to use higher isolation, JPA will still access the database in ways that causes it to work as though you have READ COMMITTED isolation.

You can override this in two ways:

By requesting pessimistic locks. This will force JPA to obtain locks when it reads or writes data.

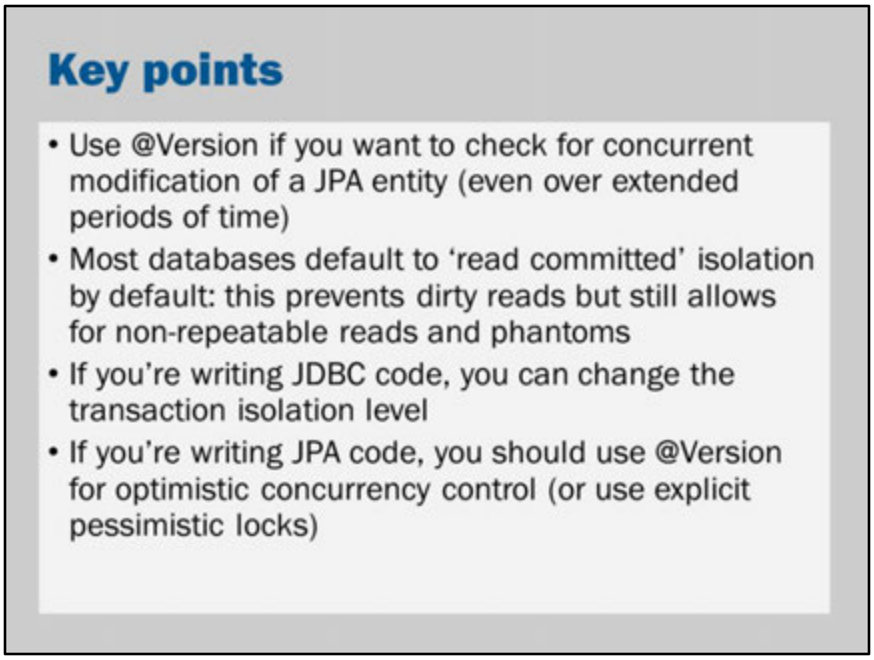
By using @Version fields. This will use optimistic concurrency control strategies. Before changing data, JPA will check the version field to see if another transaction has modified the data in the database. This will enforce repeatable reads because if another transaction has modified the database, then an exception will be thrown when you try to commit the new data with an incorrect version.

The @Version annotation works by increasing the value of the field by one every time the record is changed.

Here's an example:
Suppose X reads a record from the database with version 1.
Suppose Y also reads the same record with version 1.

Y updates the record.
Y saves the change to the database: it expects the database record to be 1 and the database is currently at version 1.
So the database is updated and the version is set to 2.

X now updates the record.
X tries to save the change to the database: it expects the database record to be 1 but the database is currently at version 2.
So there is an error. There has been a concurrent change. An exception will be thrown and X's will not be committed.



Key points

- Use @Version if you want to check for concurrent modification of a JPA entity (even over extended periods of time)
- Most databases default to 'read committed' isolation by default: this prevents dirty reads but still allows for non-repeatable reads and phantoms
- If you're writing JDBC code, you can change the transaction isolation level
- If you're writing JPA code, you should use @Version for optimistic concurrency control (or use explicit pessimistic locks)

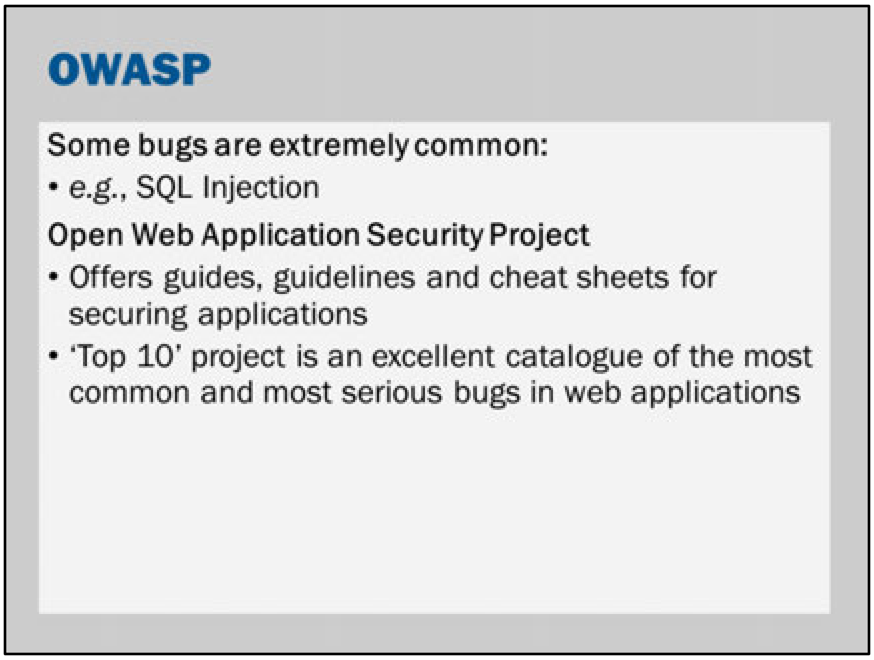
Security

The internet is an incredibly “hostile” place. It doesn't take long from when your computer is connected to the internet until you will start getting people attempting to break in.

Some of the situations I've encountered in my career include:

- Passwords guessed by brute force attempts (trying many username/password combinations until one works)
- Vendor-supplied equipment having an insecure account/password that we did not know about
- Successful SQL injection attacks
- An insecure administration section of a website: the address was eventually guessed by some ‘hackers’
- An internal system without much security being accidentally connected to the internet instead of remaining behind a firewall
- A system getting disconnected from an LDAP server, causing the accounts to revert to insecure default passwords

A few mistakes have been my own fault. Some have been caused by people on the same team as me. I can't deny all responsibility as, ultimately, computer security is everybody's responsibility.



OWASP

Some bugs are extremely common:

- *e.g.*, SQL Injection

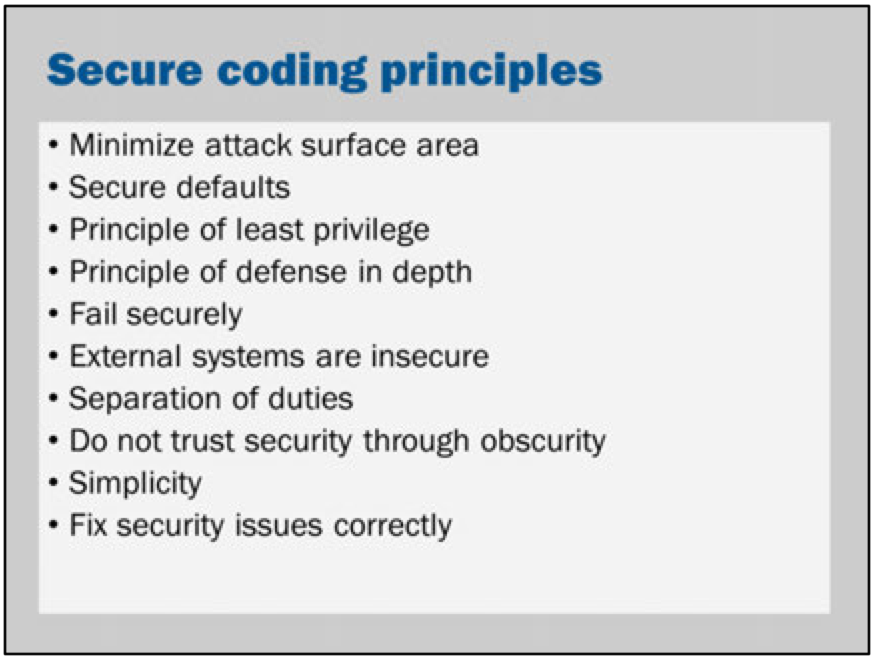
Open Web Application Security Project

- Offers guides, guidelines and cheat sheets for securing applications
- 'Top 10' project is an excellent catalogue of the most common and most serious bugs in web applications

One excellent resource for securing web applications is the OWASP project.

I very much recommend reading their guides.

If you end up working with a government department as one of your customers, you may very well be asked to comply with an OWASP guide.


Secure coding principles

- Minimize attack surface area
- Secure defaults
- Principle of least privilege
- Principle of defense in depth
- Fail securely
- External systems are insecure
- Separation of duties
- Do not trust security through obscurity
- Simplicity
- Fix security issues correctly

As web developers, we need to imagine every way that the features we add to web applications could be abused.

For example, a “password reminder” facility, if not created correctly could be abused by spammers who need to test whether an email address is valid (e.g., if you enter an email to send an account/password reminder, but the system reports that there is no matching email). It could also be used to steal an account if owner uses an email address that has been taken over by somebody else (e.g., a Hotmail email address that has expired and been taken by somebody else).

The following is an excerpt from the OWASP secure coding principles: (https://www.owasp.org/index.php/Secure_Coding_Principles)

Minimize attack surface area

Every feature that is added to an application adds a certain amount of risk to the overall application. The aim for secure development is to reduce the overall risk by reducing the attack surface area.

For example, a web application implements online help with a search function. The search function may be vulnerable to SQL injection attacks. If the help feature was limited to authorized users, the attack likelihood is reduced. If the help feature's search function was gated through centralized data validation routines, the ability to perform SQL injection is dramatically reduced. However, if the help feature was re-written to eliminate the search function (through better user interface, for example), this almost eliminates the attack surface area, even if the help feature was available to the Internet at large.

Establish secure defaults

There are many ways to deliver an "out of the box" experience for users. However, by default, the experience should be secure, and it should be up to the user to reduce their security – if they are allowed.

For example, by default, password aging and complexity should be enabled. Users might be allowed to turn these two features off to simplify their use of the application and increase their risk.

Principle of Least privilege

The principle of least privilege recommends that accounts have the least amount of privilege required to perform their business processes. This encompasses user rights, resource permissions such as CPU limits, memory, network, and file system permissions.

For example, if a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted. Under no circumstances should the middleware be granted administrative privileges.

Principle of Defense in depth

The principle of defense in depth suggests that where one control would be reasonable, more controls that approach risks in different fashions are better. Controls, when used in depth, can make severe vulnerabilities extraordinarily difficult to exploit and thus unlikely to occur.

With secure coding, this may take the form of tier-based validation, centralized auditing controls, and requiring users to be logged on all pages.

For example, a flawed administrative interface is unlikely to be vulnerable to anonymous attack if it correctly gates access to production management networks, checks for administrative user authorization, and logs all access.

Fail securely

Applications regularly fail to process transactions for many reasons. How they fail can determine if an application is secure or not.

For example:

```
isAdmin = true;
try {
  codeWhichMayFail();
  isAdmin = isUserInRole( “Administrator” );
}
catch (Exception ex) {
  log.write(ex.toString());
}
```

If either codeWhichMayFail() or isUserInRole fails or throws and exception, the user is an admin by default. This is obviously a security risk.

Don’t trust services

Many organizations utilize the processing capabilities of third party partners, who more than likely have differing security policies and posture than you. It is unlikely that you can influence or control any external third party, whether they are home users or major suppliers or partners.

Therefore, implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.

For example, a loyalty program provider provides data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items. However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large.

Separation of duties

A key fraud control is separation of duties. For example, someone who requests a computer cannot also sign for it, nor should they directly receive the computer. This prevents the user from requesting many computers, and claiming they never arrived.

Certain roles have different levels of trust than normal users. In particular, administrators are different to normal users. In general, administrators should not be users of the application.

For example, an administrator should be able to turn the system on or off, set password policy but shouldn't be able to log on to the storefront as a super privileged user, such as being able to "buy" goods on behalf of other users.

Avoid security by obscurity

Security through obscurity is a weak security control, and nearly always fails when it is the only control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden.

For example, the security of an application should not rely upon knowledge of the source code being kept secret. The security should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

A practical example is Linux. Linux's source code is widely available, and yet when properly secured, Linux is a hardy, secure and robust operating system.

Keep security simple

Attack surface area and simplicity go hand in hand. Certain software engineering fads prefer overly complex approaches to what would otherwise be relatively straightforward and simple code.

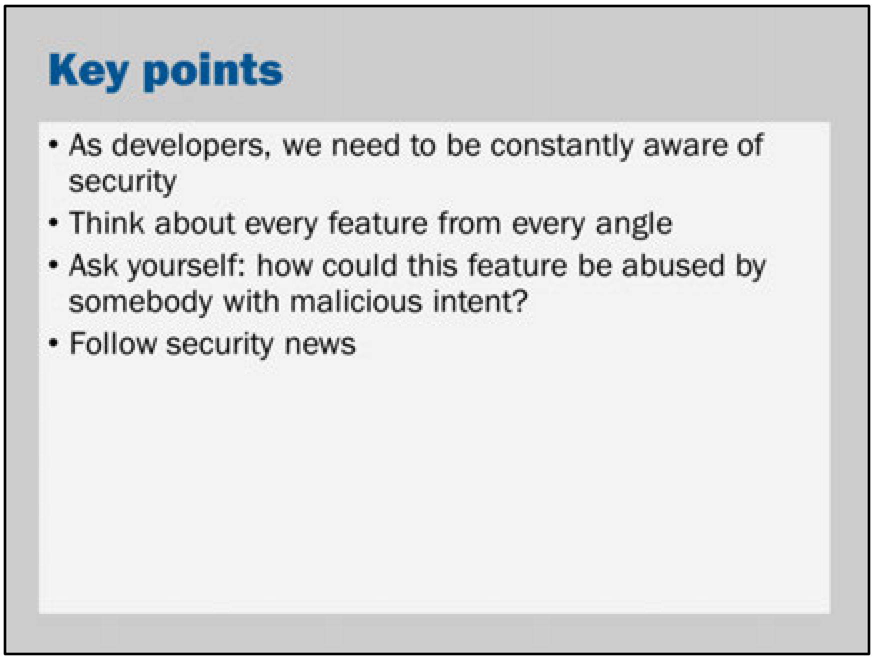
Developers should avoid the use of double negatives and complex architectures when a simpler approach would be faster and simpler.

For example, although it might be fashionable to have a slew of singleton entity beans running on a separate middleware server, it is more secure and faster to simply use global variables with an appropriate mutex mechanism to protect against race conditions.

Fix security issues correctly

Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue. When design patterns are used, it is likely that the security issue is widespread amongst all code bases, so developing the right fix without introducing regressions is essential.

For example, a user has found that they can see another user’s balance by adjusting their cookie. The fix seems to be relatively straightforward, but as the cookie handling code is shared among all applications, a change to just one application will trickle through to all other applications. The fix must therefore be tested on all affected applications.

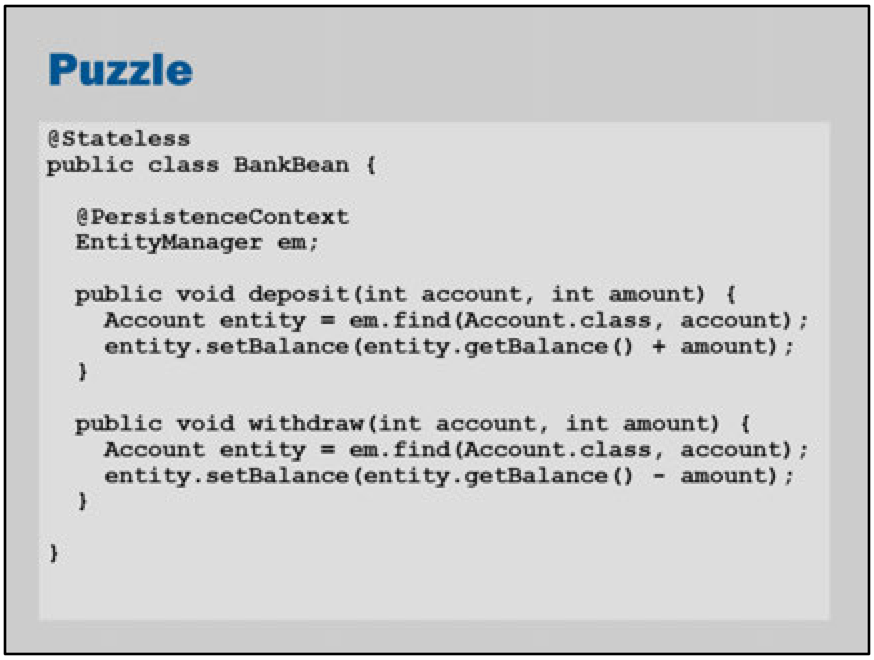


Key points

- As developers, we need to be constantly aware of security
- Think about every feature from every angle
- Ask yourself: how could this feature be abused by somebody with malicious intent?
- Follow security news

Some interesting sources of security news are:
- Crypto-Gram https://www.schneier.com/crypto-gram/
- RISKS http://catless.ncl.ac.uk/Risks/
- Krebs on Security: http://krebsonsecurity.com/

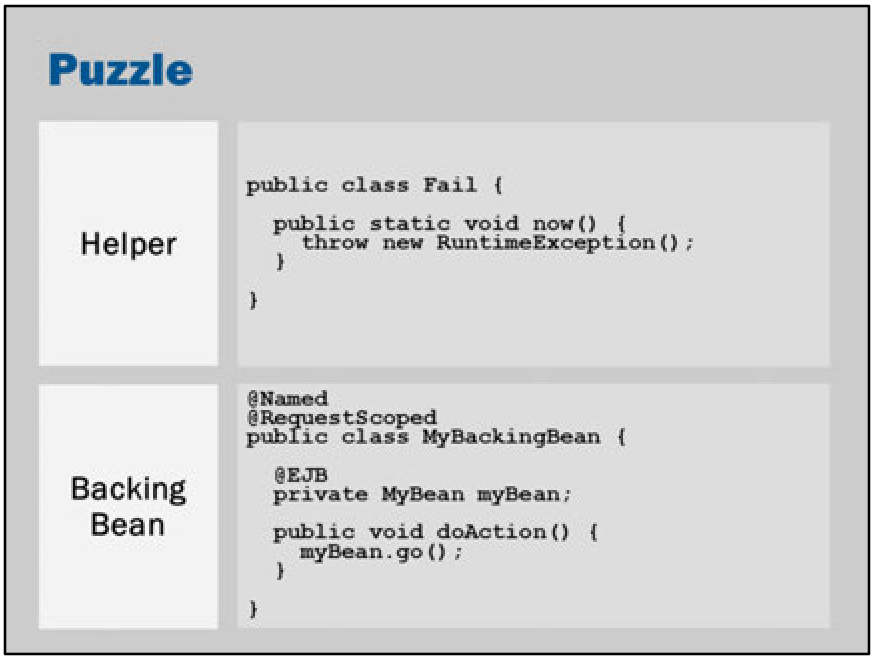
Bonus slides


Puzzle

```
@Stateless
public class BankBean {

  @PersistenceContext
  EntityManager em;

  public void deposit(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() + amount);
  }

  public void withdraw(int account, int amount) {
    Account entity = em.find(Account.class, account);
    entity.setBalance(entity.getBalance() - amount);
  }

}
```

We're going to look at a puzzle in a few slides. Note that these puzzles are very challenging – they are to test your understanding but you're not going to need to do anything this complex in your assignment.

This is one of the supporting classes. It provides operations to deposit and withdrawal money from an account number.



Puzzle

Helper

```
public class Fail {

  public static void now() {
    throw new RuntimeException();
  }

}
```

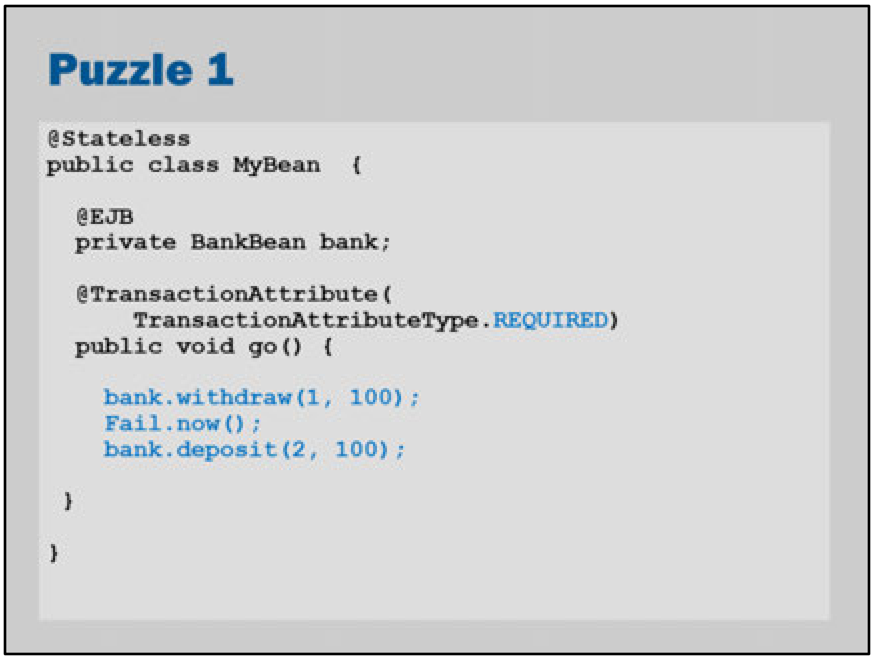
Backing Bean

```
@Named
@RequestScoped
public class MyBackingBean {

  @EJB
  private MyBean myBean;

  public void doAction() {
    myBean.go();
  }

}
```

These are also two supporting classes:

Fail.now() will throw a runtime exception… the effect of this would be to cause any current exception to be rolled back.

The backing bean isn't too important, it just happens to be the way that our puzzles will be called. The key thing to observe is that it is not an EJB. It is just an ordinary CDI backing bean. This means that there is NO transaction when the backing bean is called, and there is no transaction at the time that myBean.go() is called.


Puzzle 1

```
@Stateless
public class MyBean  {

  @EJB
  private BankBean bank;

  @TransactionAttribute(
      TransactionAttributeType.REQUIRED)
  public void go() {

    bank.withdraw(1, 100);
    Fail.now();
    bank.deposit(2, 100);

  }

}
```

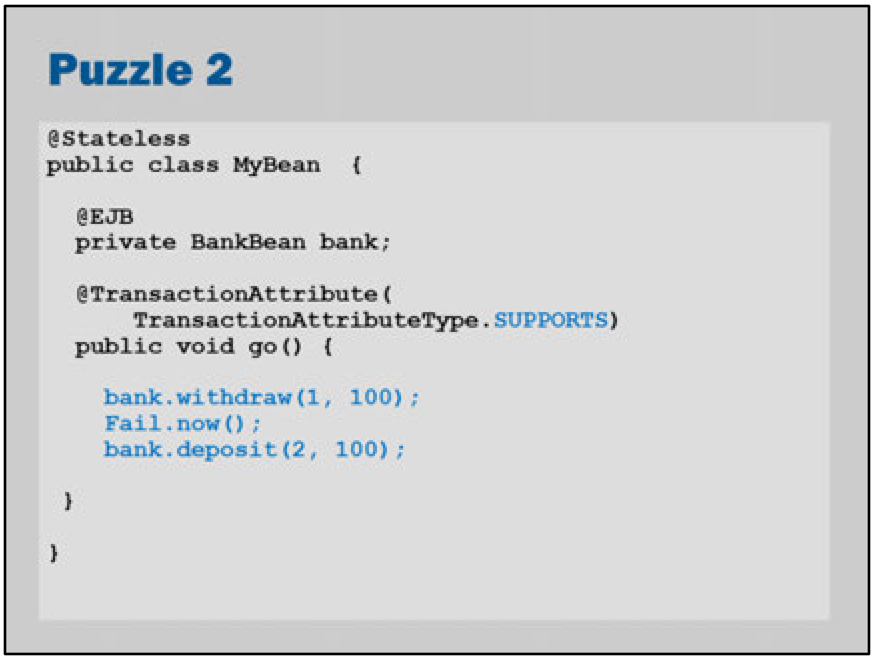
What happens here?

REQUIRED means that a transaction will be created if none exist.
Since no transaction exists at the time go() is called, a transaction will be started by the container.

Next, $100 will be withdrawn from bank account #1.

Next, the Fail.now() will cause a runtime exception to be thrown. When an exception is thrown, the method stops running and the container detects the runtime exception. The container will rollback the transaction. Rolling back the transaction will cause the $100 withdrawal to be "reversed" or "undone".

Thus, this method will have no effect. (Aside from an exception being thrown)



Puzzle 2

```
@Stateless
public class MyBean  {

  @EJB
  private BankBean bank;

  @TransactionAttribute(
      TransactionAttributeType.SUPPORTS)
  public void go() {

    bank.withdraw(1, 100);
    Fail.now();
    bank.deposit(2, 100);

 }

}
```

What happens here?

SUPPORTS means that the method will use a transaction if one exists, and it won't use a transaction if none exists.
The backing bean doesn't cause an transaction to start, so this method will run without a transaction.

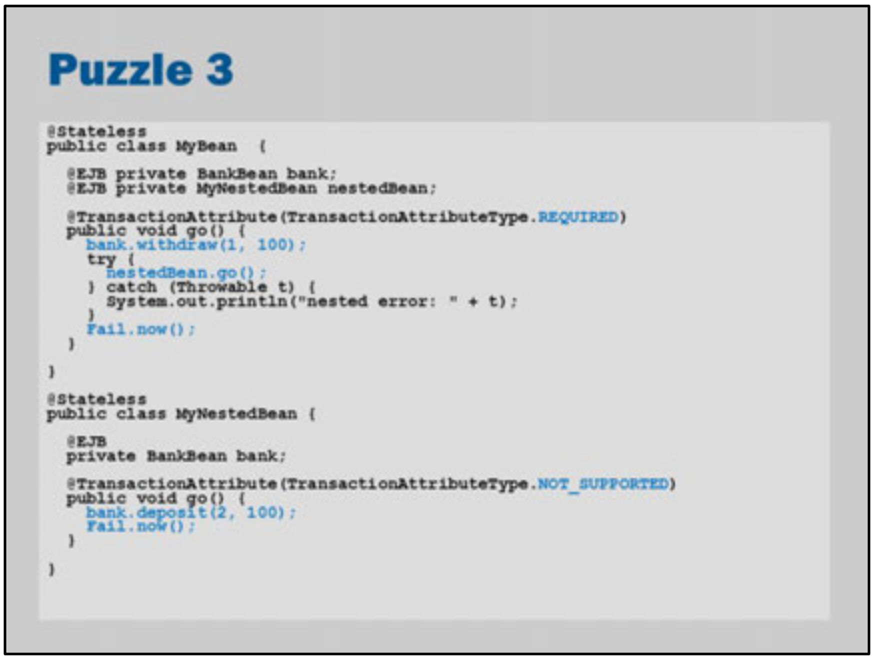
Calling bank.withdraw will withdraw $100 from account #1.

When it gets to Fail.now() the exception will stop the method from executing. However, since there is no transaction, the withdrawal will not be reversed or undone.

Thus, the effect of this method is that $100 will be withdrawn from account #1 and an exception will be thrown.

Advanced Note:
The BankBean is a stateless session bean. This means that its methods, by default,

are run with the REQUIRED transaction attribute. Thus, the withdrawal will actually be performed from within a transaction. The transaction begins at the start of the withdrawal and commits at the end of the withdrawal. However, there is no "longer running" transaction that encompasses the entire go() method.



Puzzle 3

```
@Stateless
public class MyBean  {

  @EJB private BankBean bank;
  @EJB private MyNestedBean nestedBean;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void go() {
    bank.withdraw(1, 100);
    try {
      nestedBean.go();
    } catch (Throwable t) {
      System.out.println("nested error: " + t);
    }
    Fail.now();
  }

}
@Stateless
public class MyNestedBean {

  @EJB
  private BankBean bank;

  @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
  public void go() {
    bank.deposit(2, 100);
    Fail.now();
  }

}
```

What happens here?

We have a REQUIRED method calling a NOT_SUPPORTED method.

Recall that REQUIRED ensures that there is a transaction, while NOT_SUPPORTED ensures that there isn't a transaction. Both are "correcting". This means that if there is no transaction when the REQUIRED method is called, then one will be created. If there is a transaction when the NOT_SUPPORTED method is called, then the existing transaction will be suspended.

Ok, so this is what happens:

The REQUIRED attribute is checked by the container and a transaction is started because none exists yet.

Then $100 is withdrawn from the account.

The nestedBean.go() method is then called. This is annotated with NOT_SUPPORTED so the container will *"suspend"* the current transaction and run the method outside

the existing transaction.
NOTE: The existing transaction is NOT cancelled, committed or rolled-back. It is still there. It is just that anything inside NOT_SUPPORTED is not performed within the transactional scope.

The first step of the nestedBean.go() method is to deposit $100 into account 2.

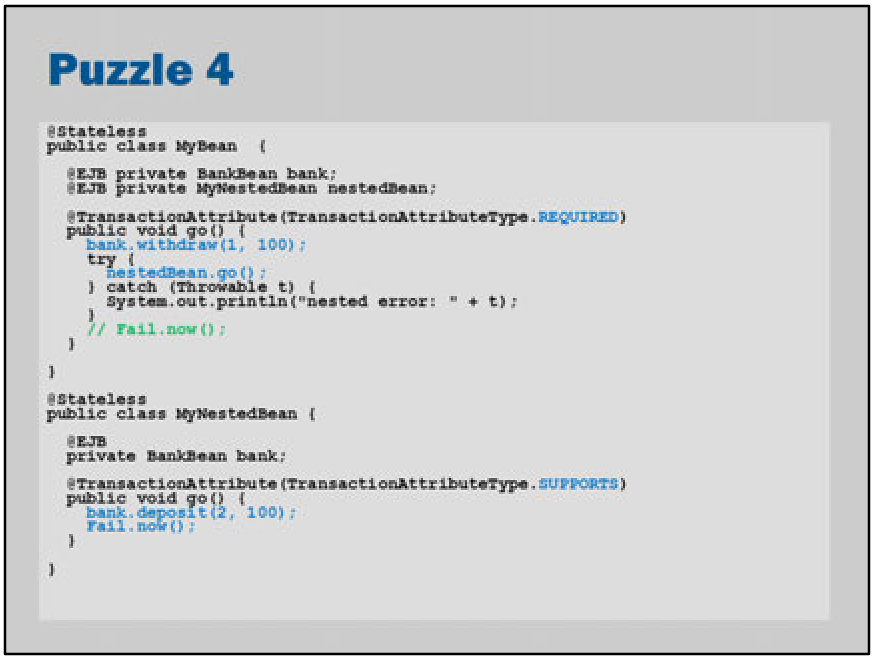
Next, Fail.now() causes an exception to be thrown. Since the deposit was NOT inside a transaction, it isn't going to be undone (i.e., rolled back).

The exception is then caught by the try-catch statement.

Then, the Fail.now() just after the try-catch is executed. This causes the method to stop executing. Since it is a RuntimeException, the current transaction will be rolled back. The withdrawal was part of the current transaction, so the withdrawal will be rolled back (i.e., "undone").

Therefore, the effect of this method is to deposit $100 in account 2 and throw an exception (but there is no withdrawal).

Advanced comment:
Actually, the story is a little bit more complicated. The deposit method runs as though it was annotated with REQUIRED. So, this means that when running nestedBean.go() there is one suspended transaction, and a new separate transaction for processing the deposit.



Puzzle 4

```
@Stateless
public class MyBean  {

  @EJB private BankBean bank;
  @EJB private MyNestedBean nestedBean;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void go() {
    bank.withdraw(1, 100);
    try {
      nestedBean.go();
    } catch (Throwable t) {
      System.out.println("nested error: " + t);
    }
    // Fail.now();
  }
}

@Stateless
public class MyNestedBean {

  @EJB
  private BankBean bank;

  @TransactionAttribute(TransactionAttributeType.SUPPORTS)
  public void go() {
    bank.deposit(2, 100);
    Fail.now();
  }
}
```

What happens here?

This time we have REQUIRED calling SUPPORTS.
SUPPORTS uses an existing transaction if there is one.

First, the transaction will be created because the method is annotated with REQUIRED.

$100 will be withdrawn from account #1.

Then the nestedBean.go() method will be called. Since it is SUPPORTS, the existing transaction will be used.

$100 will be deposited into account #2.

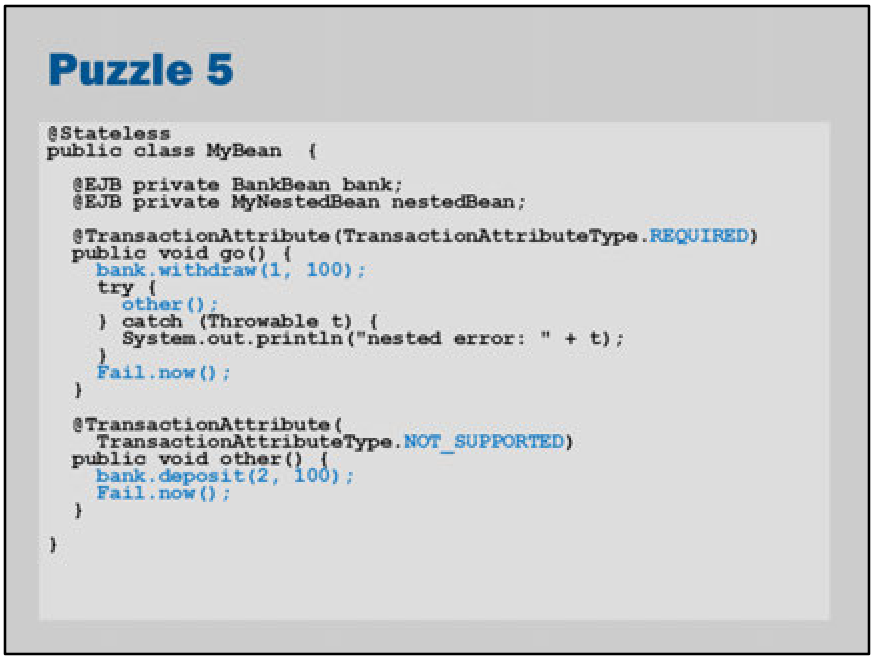
Next, an exception is thrown. Because the exception is a runtime exception, it will mark the transaction for rollback.

The exception will be caught, and a message will be displayed on the screen.

Even though the method returns normally, the exception in nestedBean.go() has caused the transaction to be rollback-only.

Thus, the container will rollback all of the effects of the transaction.

This code will therefore have no effect (aside from printing an error message to the server logs). The database will not be modified because all changes will be rolled back.


Puzzle 5

```
@Stateless
public class MyBean  {

  @EJB private BankBean bank;
  @EJB private MyNestedBean nestedBean;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void go() {
    bank.withdraw(1, 100);
    try {
      other();
    } catch (Throwable t) {
      System.out.println("nested error: " + t);
    }
    Fail.now();
  }

  @TransactionAttribute(
    TransactionAttributeType.NOT_SUPPORTED)
  public void other() {
    bank.deposit(2, 100);
    Fail.now();
  }

}
```

What happens here?

The code is the same as Puzzle 3, except it is all in one class instead of two.

This is actually a trick-question.

Calling other() is a "local" call. Java will call the method directly. The application server does not intercept direct invocations. It is only able to intercept calls to EJBs that you obtained by injection or by JNDI lookup.

So, the code above is equivalent to this:

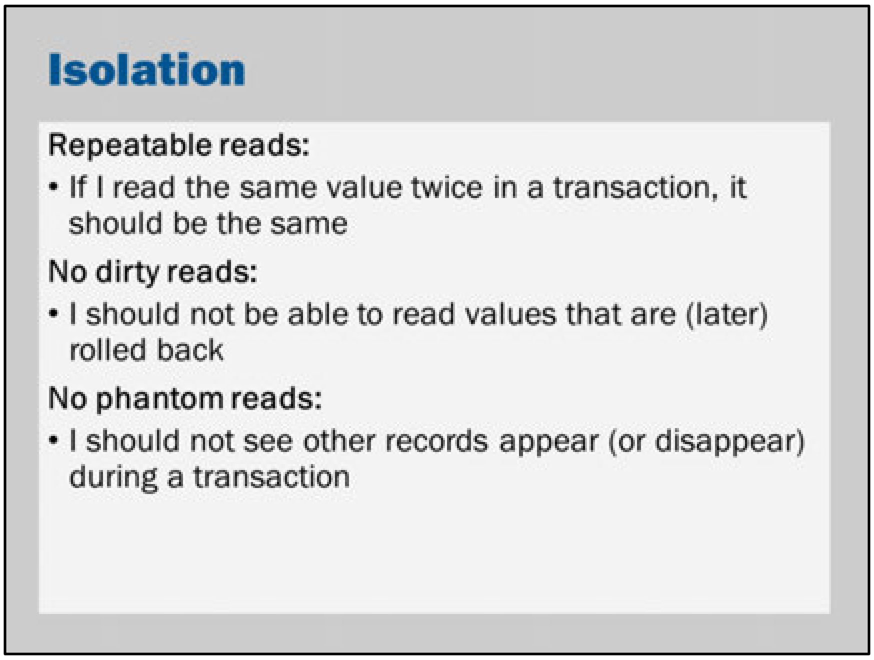
```
@Stateless
public class MyBean  {

  @EJB private BankBean bank;
  @EJB private MyNestedBean nestedBean;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void go() {
    bank.withdraw(1, 100);
    try {
      other();
    } catch (Throwable t) {
      System.out.println("nested error: " + t);
    }
    Fail.now();
  }

  private void other() {
    bank.deposit(2, 100);
    Fail.now();
  }

}
```

Which is, in turn, equivalent to this:

```
@Stateless
public class MyBean  {

  @EJB private BankBean bank;
  @EJB private MyNestedBean nestedBean;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void go() {
    bank.withdraw(1, 100);
    try {
      bank.deposit(2, 100);
      Fail.now();
    } catch (Throwable t) {
      System.out.println("nested error: " + t);
    }
    Fail.now();
  }

}
```

The code all runs in one transaction so when the Fail.now() occurs at the end, the transaction will be rolled back.

Thus, the code will not result in any change to the database but an exception will be thrown.


Isolation

Repeatable reads:

- If I read the same value twice in a transaction, it should be the same

No dirty reads:

- I should not be able to read values that are (later) rolled back

No phantom reads:

- I should not see other records appear (or disappear) during a transaction

Our objective, serializability, can be understood in terms of (at least) three constraints.

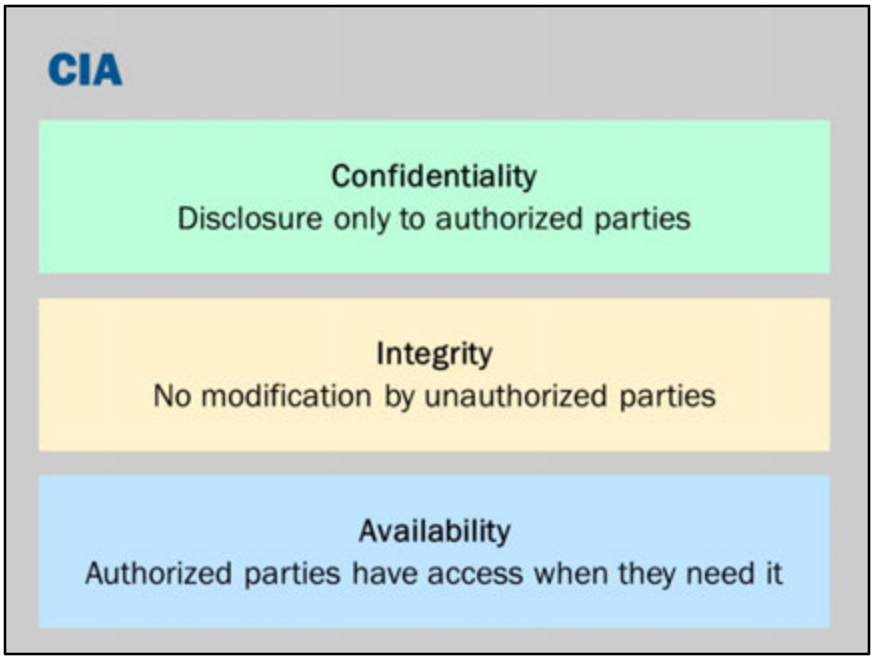
Imagine we have two transactions, A and B.

1. A reads the bank balance ($100).
2. B updates the bank balance (adds $200).
3. A reads the bank balance again ($300).
4. A updates the bank balance (adds $500, total is now $800).

In step 3, we would not have repeatable reads. If a transaction was running in isolation, then there would be no reason for a value to change between two identical read operations.

In step 3 and 4, the value of the bank balance is read and updated. However, what happens if transaction B is rolled back? In that case we would have to reverse the $200. However, this is not straightforward. We cannot simply set the value back to $100 because the $300 balance has already been used to calculate a new total of $800. In an ideal world, you could subtract $200 from the total ($800) to give a final balance of $600. We do not live in an ideal world and databases typically are not able to figure this out automatically.

Phantom reads is very similar to the issue of non-repeatable reads and dirty reads. However, they are treated separately because of a technical implementation detail of modern database systems (namely, if you have a database that uses row-level locking, it is complex to manage locks on rows that don't yet exist – the usual solution would be to use a table-level lock that causes problems because it does not allow as much concurrency).



CIA

Confidentiality
Disclosure only to authorized parties

Integrity
No modification by unauthorized parties

Availability
Authorized parties have access when they need it

The “CIA triad” is a classic model of computer security.
These are said to be three principles that a secure system should provide.

Confidentiality
Ensuring data is seen only seen by valid users. e.g., Having a password and log-in process.

Integrity
Protecting the system from unauthorized use or modification. e.g., Securing against hackers.

Availability
Ensuring the system is available. e.g., Protecting against DDoS (Distributed Denial of Service).

Availability is perhaps a surprising dimension that isn't typically associated with security.
However, it is important.
Sometimes disabling a system can be just as effective in creating harm as it is to hack

into or modify a system.
For example, if you can stop some important financial news from being published by the Australian Stock Exchange (or prevent traders from discovering that information), you could make a lot of money by trading on that news before anybody else.