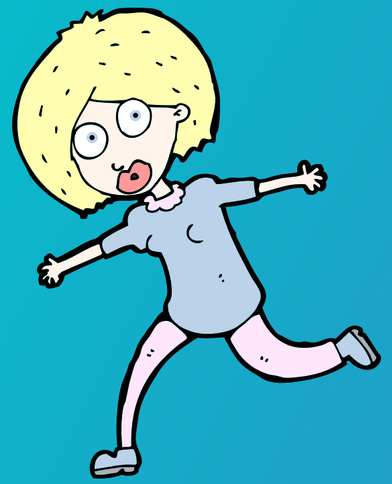
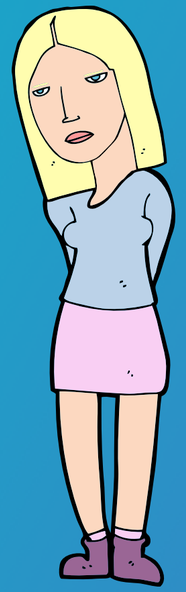
Flatmap your RPC!

Three States of JVM Thread



Running



Waiting
(blocked)

IO

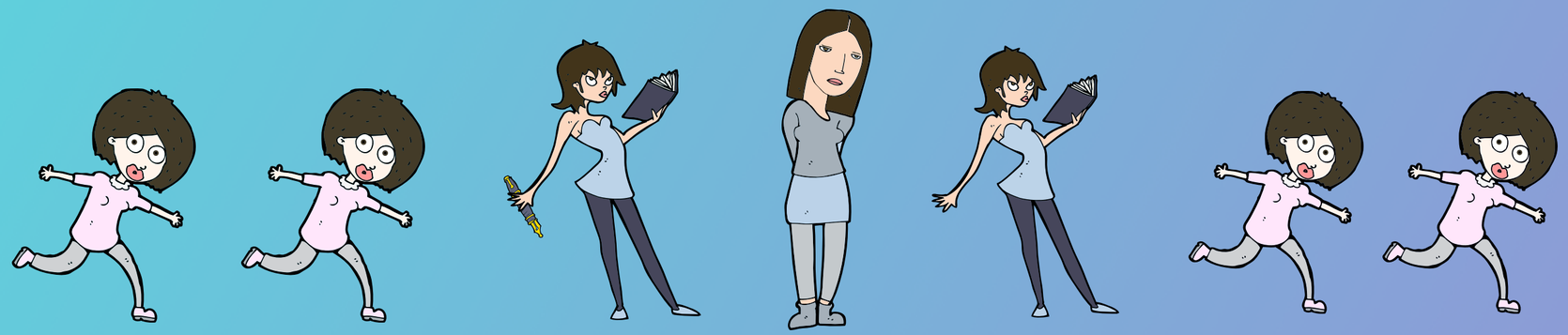
Mixing these States

- Remote procedure calls (RPC)

```
Document document = documentService.getDocument(ctx, GetDocumentRequest //
    .builder(req.document.id) //
    .withVersion(req.document.version) //
    .forReading() //
    .build()).document;
```

- Database transactions

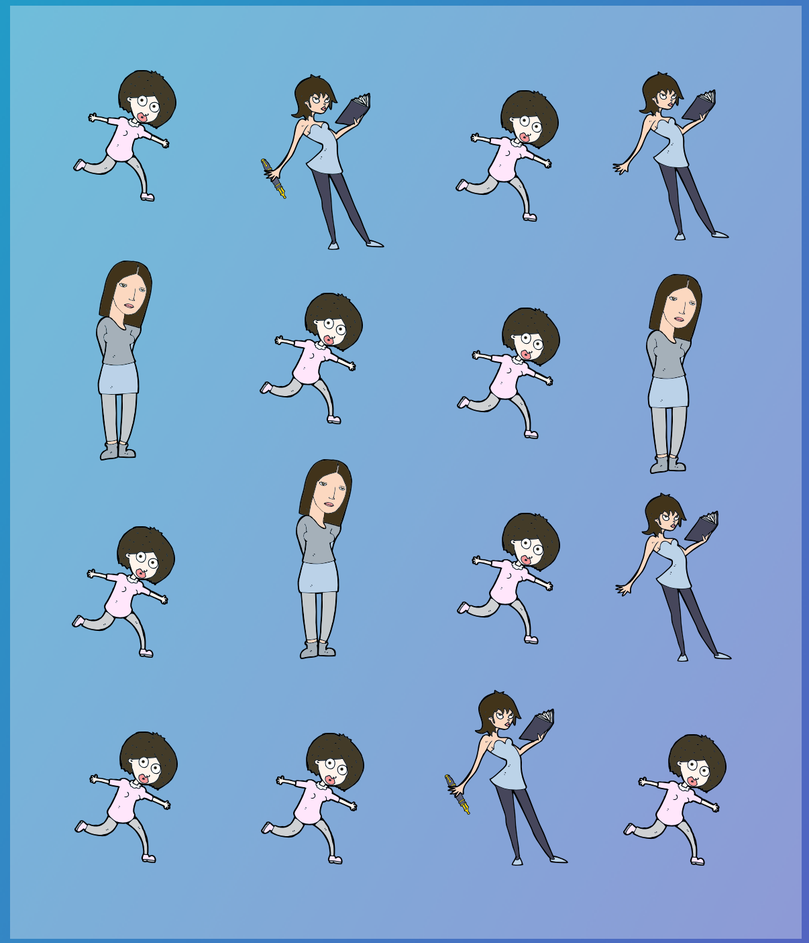
```
try (WriteTransaction txn = repo.beginWriteTransaction()) {
  ProductDto product = repo.getProduct(txn, req.productSku);
```



Execution Context

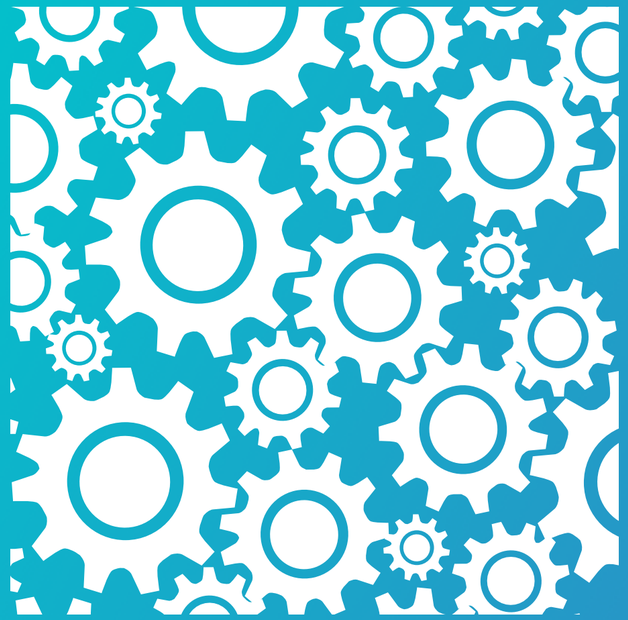
16 threads handling requests

```
profile.service.worker.threads=16
review.service.worker.threads=12
share.service.worker.threads=16
media.service.worker.threads=16
```



- Max number of threads RUNNING?
- Max number of threads doing IO?
- Min number of threads RUNNING?

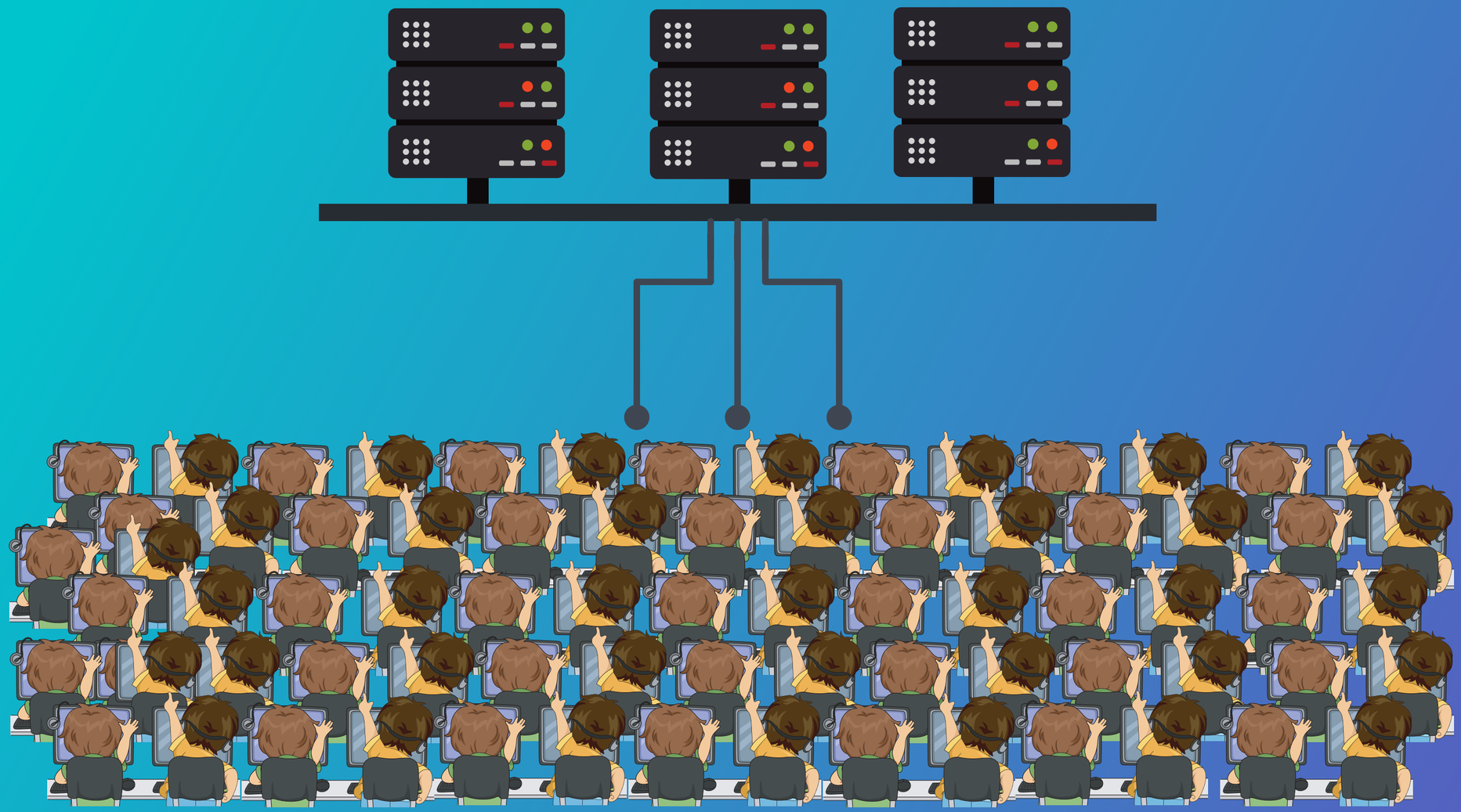
Higher Demands



Lots of computational jobs
(high CPU utilisation)

Highly loaded IO
(many connections)

MMO



Maximize CPU utilization

- Computational threads pool
- Don't block
- Don't IO

Example: AKKA

100'000 computational units

running in a pool of 32 threads

Maximize IO throughput

IO - one thread doing one IO job

NIO - many simultaneous IO operations managed in a small thread pool

Example:

5'000 active socket connections

living in a pool of 32 threads

Challenge: split the Source Code

```
public Report transfer(AccountID senderID, Password senderPassword, AccountID receiverID, Credits credits) {
    boolean senderAuthorized = authorize(senderID, senderPassword); // ← IO
    if (!senderAuthorized) {
      throw new NotAuthorizedExeption();
    }
    Account sender = getAccount(senderID); // ← IO?
    Account receiver = getAccount(receiverID); // ← IO?
    InvoiceID invoiceID = billing.transfer(sender.brandID(), receiver.brandID(), credits); // < IO
    Report report = generateBillingReport(invoiceID); // CPU
    send(sender.email(), report); // IO
    return report;
}
```

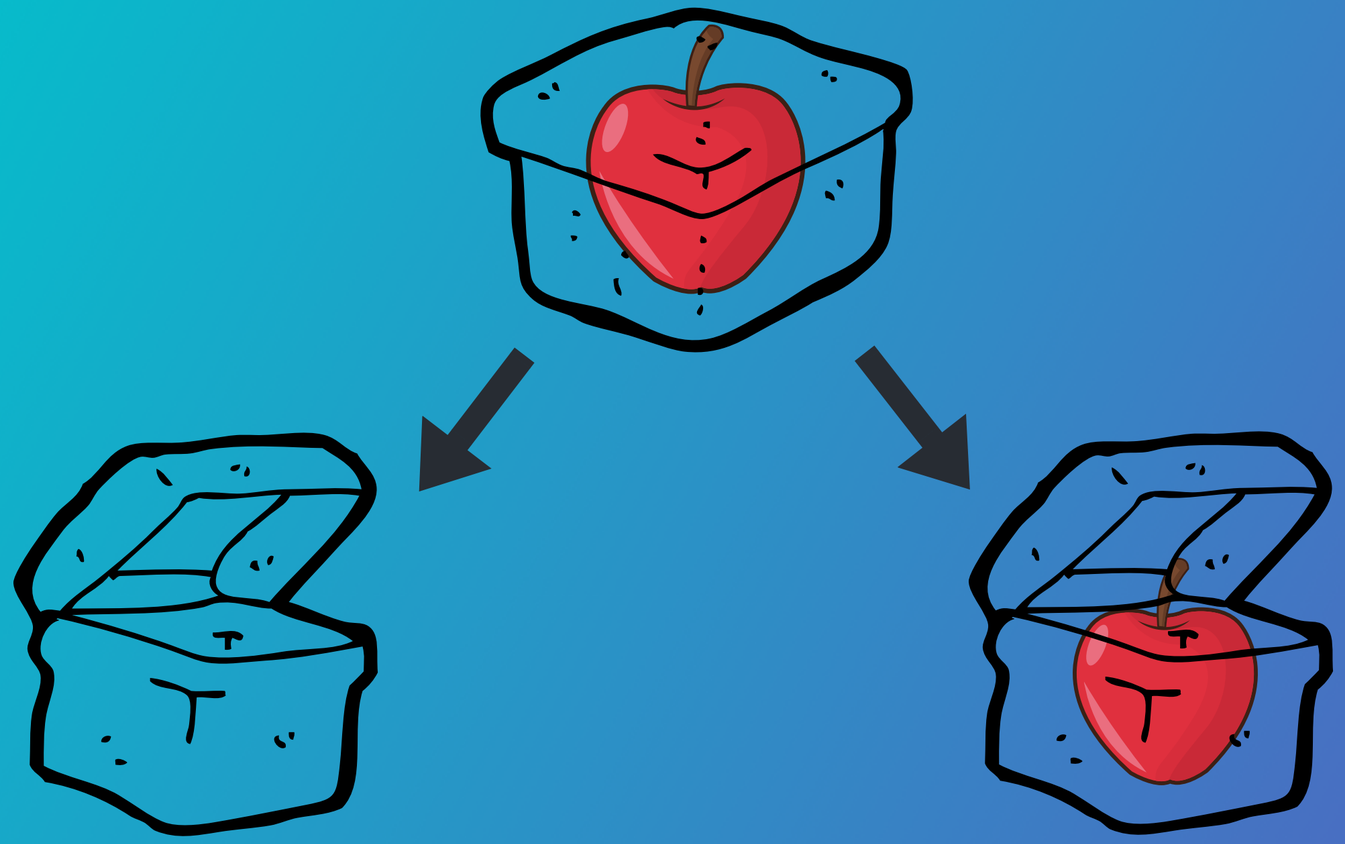
Callbacks? Callables? Actors? Compiler plugins?

Challenge: split the Source Code

Solution: flatmap that shit!

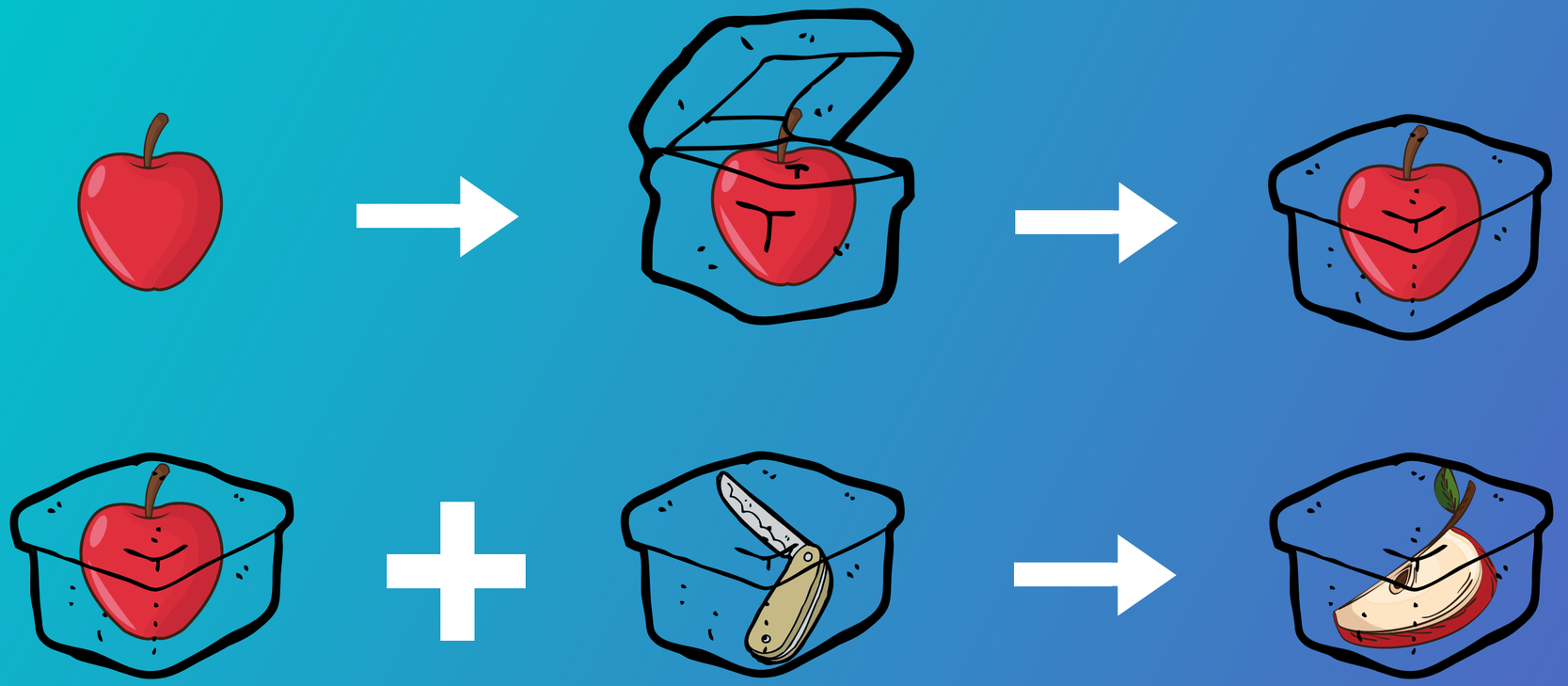
Introducing Future<T>

Future is like a box...



Future<T>

Fundamental Future properties



Code transformation

Declaration:

```
// Before:
Invoice transfer(BrandID sender, BrandID receiver, Credits amount);


// After:
Future<Invoice> transfer(BrandID sender, BrandID receiver, Credits amount);
```

Invocation:

```
// Berofe:
InvoiceID invoiceID = billing.transfer(sender.brandID(), receiver.brandID(), credits);


// After:
billing.transfer(sender.brandID(), receiver.brandID(), credits) .flatMap(invoiceID →


// Scala version:
invoideID ← billing.transfer(sender.brandID(), receiver.brandID(), credits)
```

Code Transformation

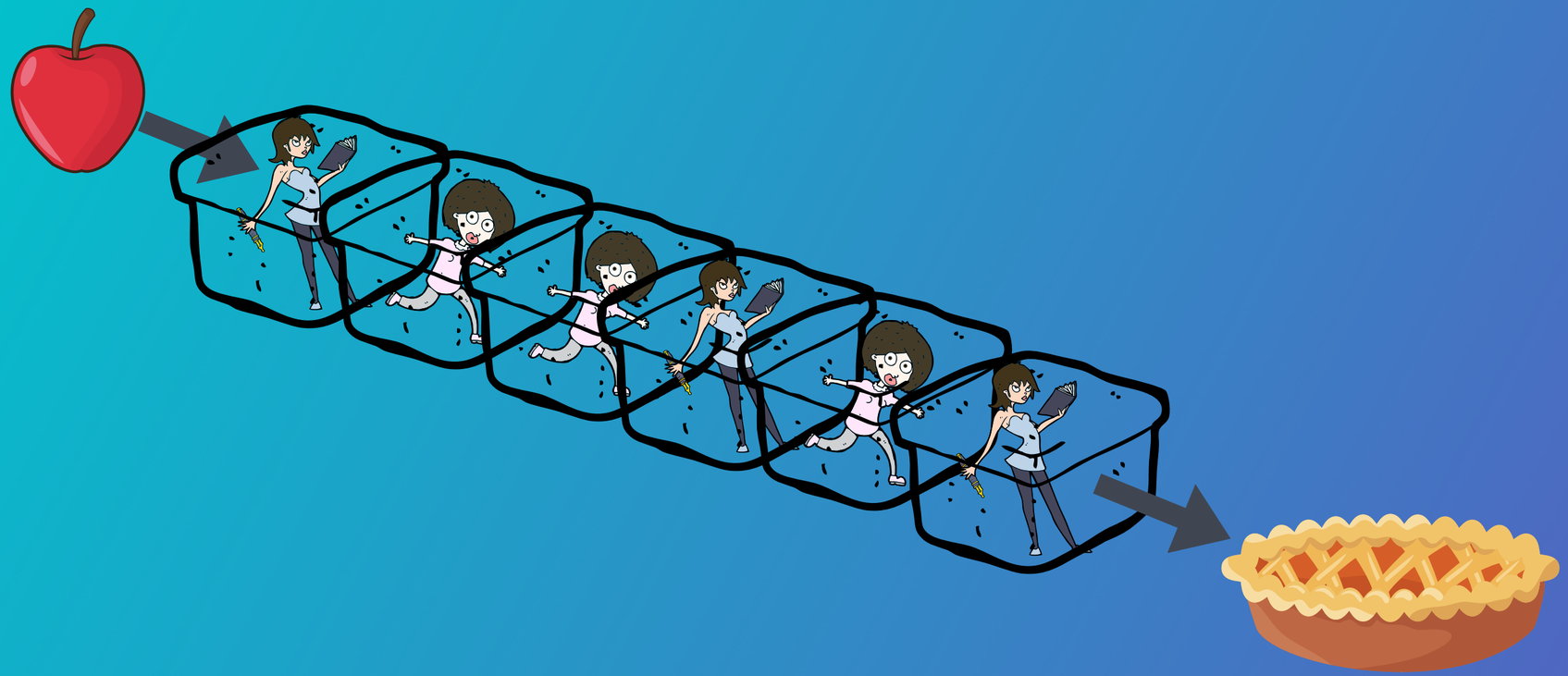
Before:

```
@Override
public void transfer(AccountID senderID, Password senderPassword, AccountID receiverID, Credits credits) {
  boolean senderAuthorized = securityService.authorize(senderID, senderPassword);
  if (!senderAuthorized) {
    throw new RuntimeException("Not authorized");
  }
  Account sender = accountService.getAccount(senderID);
  Account receiver = accountService.getAccount(receiverID);
  InvoiceID invoiceID = billingService.transfer(sender.brandID(), receiver.brandID(), credits);
  ReportID reportID = reportsService.generateBillingReport(invoiceID);
  emailService.sendReport(sender.emailAddress(), reportID);
}
```

After:

```
@Override
public Task<Void> transfer(AccountID senderID, Password senderPassword, AccountID receiverID, Credits credits) {
  securityService.authorize(senderID, senderPassword) ... senderAuthorized → {
  if (!senderAuthorized) {
    return Task.fail(new RuntimeException("Not authorized"));
  } ...
  accountService.getAccount(senderID) ... sender →
  accountService.getAccount(receiverID) ... receiver →
  billingService.transfer(sender.brandID(), receiver.brandID(), credits) ... invoiceID →
  reportsService.generateBillingReport(invoiceID) ... reportID →
  emailService.sendReport(sender.emailAddress(), reportID)...
}
```

Each box works in its own execution context



The power of functional composability

Practical benefits

1. Improve server throughput by doing more IO with Java NIO.

2. Ensure the highest CPU capacity by never blocking computational threads.

3. Enjoy the composability of functional paradigms.

Thank you!