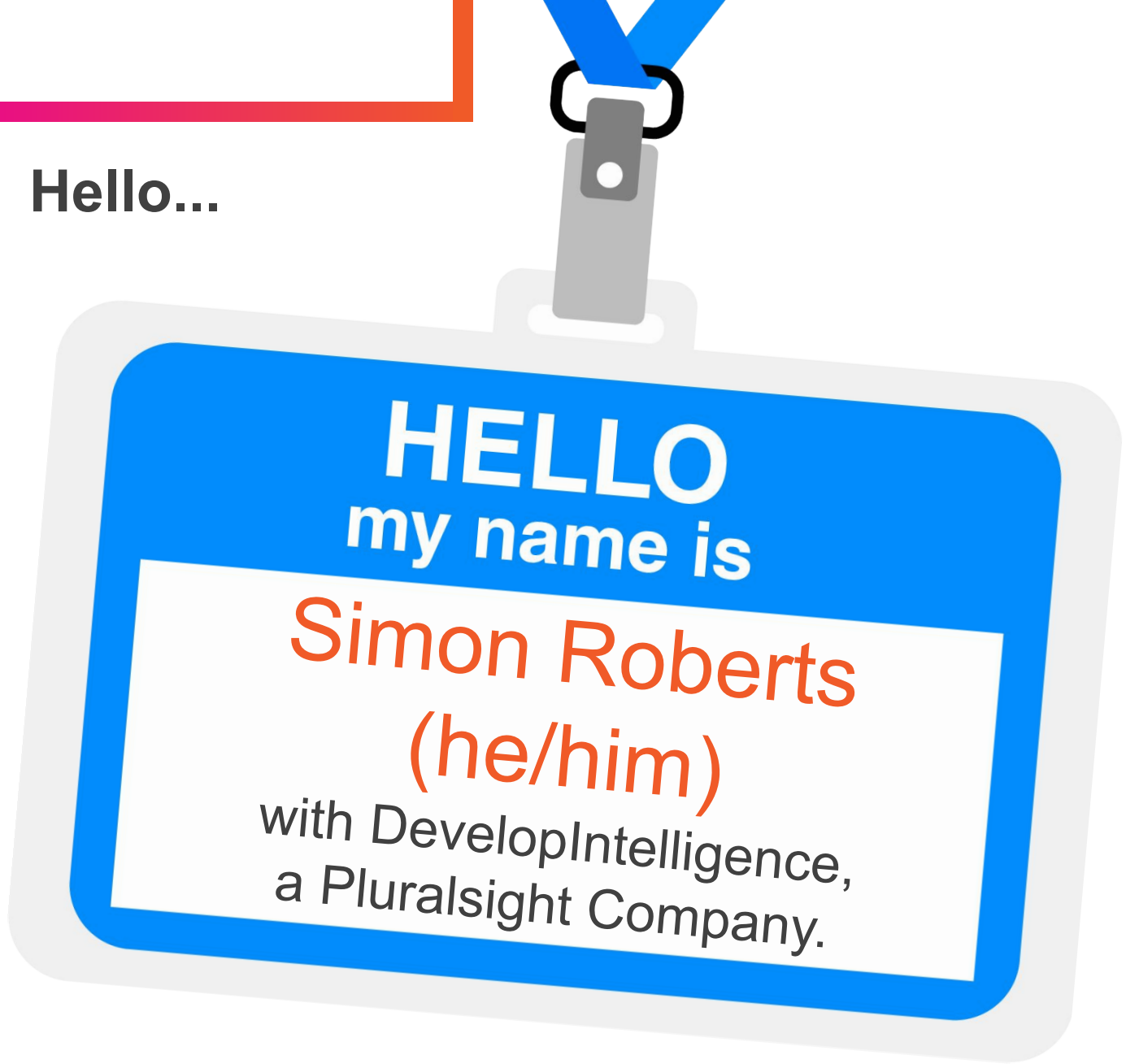
Welcome

Advanced Java TT Real World Design

DevelopIntelligence

A PLURALSIGHT COMPANY

Hello...



About me...

We teach over 400 technology topics

You experience our impact on a daily basis!

Prerequisites

This course assumes you

- Solid programming skills in an object oriented language
  - (any code examples will be presented in Java)
- Understanding of basic OO design concepts such as
- inheritance
- composition / association
- argument passing in methods
- Basic awareness of business pressures and non functional requirements in software projects

Why study this subject?

- Design is never "perfect"
- We rarely get enough time to think about this during the pressure of the working day

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Create an inclusive learning environment
- Use an on-screen timer for breaks

…also, if you have an accessibility need, please let me know

Objectives

At the end of this course you will be able to:

- Understand the need for a balanced response to conflicting pressures on a software project
- Describe how key aspects of software design have non-functional and business-related consequences
- Determine concerns of stakeholders that should be considered in the design process
- Describe the benefits and costs of several key design patterns

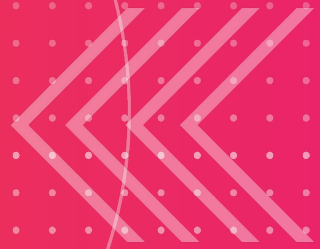
How we’re going to work together

- Discussions
- Diagrams
- Examples
- You'll have a copy of all the course materials shortly

Student Introductions

- Job title?
- Where are you based?
- Design skill evaluation
- Specific questions?
- Fun fact?

Thank you!



DevelopIntelligence
A PLURALSIGHT COMPANY

Design Concepts

Develop Intelligence

If a change from design A to design B is easy, and both satisfy the functional requirement, but a change from B to A is relatively hard, prefer design A unless there are other reasons to choose between them

- Make classes final until there is a determined need for inheritance, and the consequences are understood
- Prefer factories or builders over constructors
  - Constructors can only produce a new object of the exact type, or an exception
  - Any method can do this, but can do much more too
  - Removing constructors breaks clients, changing the implementation details of a method does not.

Design concepts

Develop Intelligence

Consider preferring immutable data and structures

- This can avoid "that can't happen" moments where other users of data altered them unexpectedly
- Immutable data can be shared, perhaps reducing the need for copying and compensating for the need to make new data to describe variations on old data
- Sometimes an immutable proxy (such as Collections.unmodifiableXxx) serves well to prevent clients changing something while allowing the "owner" to change it.

Design concepts

Develop Intelligence

Ensure that a business domain object is always "valid"

- Avoids unexpected states breaking client code
- Ensure any/all mutable fields are private
- Ensure that validity is verified prior to completing construction and before all changes
- Represent adjunct concepts (such as wire transfer and database storage) in secondary classes (e.g. a Data Transfer Object). Ensure the secondary class is dependent on the primary but not the other way round.

Design concepts

Develop Intelligence

Keep in mind that a Set is intended to reject duplicates, and might be preferable to a List in some situations. However:

- Sets reject duplicates simply by returning false from their add methods. This is easy to overlook.
- Sets require objects that implement equals/hashcode or ordering, and it's not always clear which is required if you simply have a java.util.Set.
- Most (but not all) core Java objects do not implement equals/hashcode unless they're immutable (String, primitive wrappers, java.time classes, for example)

Key concerns of professional software projects

Develop Intelligence

Profit - costs now / costs later

Time to market - lost market share is hard to regain

Change - requirements, team members, deployment environments, supporting components, and much more

Limit the consequences of change

What's up with implementation inheritance?

Develop Intelligence

Benefit: (internal) code reuse - write the behavior for the general case, it's then automatically provided to the special cases

Benefit: generalization (external code reuse) - code written on the generalization, automatically works on specializations (maintain Liskov substitutability!)

Problem: *in class-based languages* behavior is fixed at instant of construction

Problem: *in single-inheritance languages* variations in behaviors are limited to a single dimension of change

Using inheritance can impede handling change

Generalization and code reuse without inheritance

Dynamically typed languages often avoid these issues. JavaScript's prototypal inheritance in particular bypasses all these issues, while losing the large-project advantages of static strong type checking

Interfaces provide generalization with no strings attached

Variables referring to behavior (methods/functions) provide for *variability* of behavior. This is often called delegation and is key to several core "Gang of Four" design patterns

Generalize using interfaces, vary behavior with variables

Simple principles limiting consequences of change

Keep together what belongs together

so you know where to look

Keep apart what belongs apart

don't put something into a class simply for lack of a better place

Keep together what changes together

packages, libraries, services; keep all the changes in one "unit"

Keep apart what changes independently

avoid confusion, accidental damage, and merge conflicts

Strategy pattern

Develop Intelligence

Clients see a single object type.

Behavior variations through member variables (fields) in the object, these are called **strategy objects.**

These member variables are each of an interface type, groups of strategies implement the same interface. Prefer more fine grained interfaces.

External behaviors can delegate to "strategy" objects and can change at runtime.

Based on domain requirements, the object can provide methods to change those strategies, and impose rules on what changes are permitted.

Command pattern

Develop Intelligence

An argument to a function or method exists primarily for the purpose of the behavior that argument provides, rather than for data / state.

The called function delegates to the behavior embedded in that argument.

Similar to strategy in that objects are used for the behavior they contain, and behavior can vary.

Differs from strategy in that the behavior that supports an operation is provided along with the request to perform that action, rather than being stored ready for future use.

Factories and builders

Invoking a constructor results in either a *new* object of exactly the *named type*, or an exception.

Multiple constructors must generally differ by argument type-list (overloads)

Other methods that return objects can return any assignment compatible object, new or old.

Changing from constructors to methods returning objects would be a consequential change, but there's never a reason to change in the opposite direction

Changes of approach often have asymmetric consequences

Creating behaviors

Develop Intelligence

Since objects can exist for the behavior they contain, functions can return behaviors.

Using closure, a function can return a new behavior based on behavior embedded in an argument to that function.

Such approaches might be called *behavior factories*, *function decorators*, or *function transformers*.

Iterator and iterable

Develop Intelligence

Data structure (the iterable) varies independently of the clients of that structure, hence iteration should be "kept with" the data structure.

Iteration progress varies with the client, so progress must be a per-client feature.

The Iterator class describes iteration progress, and has privileged knowledge about data structure, but is instantiated separately from the structure class.

The Iterable instantiates the Iterator in response to clients' requests.

Requires language features (e.g. inner classes) to provide privileged access to the data structure.

Functor

A container of data (such as a List) that provides a means of applying a transformation to every item it contains, and produces a new container of the same type that contains the results

Conventional method name is called map

Takes a function (behavior, or object that defines the operation) as an argument (as per the command pattern)

Should not change the input data in any way

Can hide (and therefore hide changes in) the means of applying the operation to the data. E.g. can execute in multiple concurrent, or distributed, threads.

Monad

Similar to a Functor, but provides a flatmap operation which accepts a behavior that produces a result embodied in another Monad (usually of the same type)

Allows a 1 to N relationship between input items and output results (contrast with Functor which has a 1 to 1 relationship)